ANNOUNCEMENTS

- Homework 1 due on Thursday (2/2)
- Based on course feedback, we will hold in-person OHs every week on Wednesday 9-11 AM PT. Location will be updated on the OH calendar.
Recap: A General GNN Framework

(1) Message

(2) Aggregation

(3) Layer connectivity

(4) Graph augmentation

(5) Learning objective

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020
Recap: GNN Training Pipeline

**Implementation resources:**
- **PyG** provides core modules for this pipeline
- **GraphGym** further implements the full pipeline to facilitate GNN design
How powerful are GNNs?

- Many GNN models have been proposed (e.g., GCN, GAT, GraphSAGE, design space).

- What is the expressive power (ability to distinguish different graph structures) of these GNN models?

- How to design a maximally expressive GNN model?
We focus on message passing GNNs:

1. **Message**: each node computes a message
   \[ m_u^{(l)} = \text{MSG}^{(l)} \left( h_u^{(l-1)} \right), u \in \{ N(v) \cup v \} \]

2. **Aggregation**: aggregate messages from neighbors
   \[ h_v^{(l)} = \text{AGG}^{(l)} \left( \{ m_u^{(l)}, u \in N(v) \}, m_v^{(l)} \right) \]
Many GNN models have been proposed:
- GCN, GraphSAGE, GAT, Design Space etc.

Different GNN models use different neural networks in the box.
GNN Model Example (1)

- GCN (mean-pool) [Kipf and Welling ICLR 2017]

Element-wise mean pooling + Linear + ReLU non-linearity
GraphSAGE (max-pool) [Hamilton et al. NeurIPS 2017]

MLP + element-wise max-pooling
We use node same/different colors to represent nodes with same/different features.

- For example, the graph below assumes all the nodes share the same feature.

Key question: How well can a GNN distinguish different graph structures?
We specifically consider **local neighborhood structures** around each node in a graph.

- **Example**: Nodes 1 and 5 have **different** neighborhood structures because they have different node degrees.
We specifically consider local neighborhood structures around each node in a graph.

Example: Nodes 1 and 4 both have the same node degree of 2. However, they still have different neighborhood structures because their neighbors have different node degrees.

Node 1 has neighbors of degrees 2 and 3. Node 4 has neighbors of degrees 1 and 3.
We specifically consider **local neighborhood structures** around each node in a graph.

- **Example:** Nodes 1 and 2 have the **same** neighborhood structure because **they are symmetric within the graph.**

Node 1 has neighbors of degrees 2 and 3.
Node 2 has neighbors of degrees 2 and 3.
And even if we go a step deeper to 2^{nd} hop neighbors, both nodes have the same degrees (Node 4 of degree 2)
Key question: Can GNN node embeddings distinguish different node’s local neighborhood structures?
- If so, when? If not, when will a GNN fail?

Next: We need to understand how a GNN captures local neighborhood structures.
- Key concept: Computational graph
In each layer, a GNN aggregates neighboring node embeddings.

A GNN generates node embeddings through a computational graph defined by the neighborhood.

Ex: Node 1’s computational graph (2-layer GNN)
Ex: Nodes 1 and 2’s computational graphs.
**Ex:** Nodes 1 and 2’s computational graphs.

But GNN only sees node features (not IDs):
A GNN will generate the same embedding for nodes 1 and 2 because:

- Computational graphs are the same.
- Node features (colors) are identical.

Note: GNN does not care about node ids, it just aggregates features vectors of different nodes.
In general, different local neighborhoods define different computational graphs.
Computational graphs are identical to **rooted subtree structures** around each node.

**Rooted subtree structures** (defined by recursively unfolding neighboring nodes from the root nodes)
- GNN’s node embeddings capture rooted subtree structures.
- Most expressive GNN maps different rooted subtrees into different node embeddings (represented by different colors).
Recall: Injective Function

- **Function** \( f : X \rightarrow Y \) is **injective** if it maps different elements into different outputs.
- **Intuition:** \( f \) retains all the information about input.
Most expressive GNN should map subtrees to the node embeddings \textit{injectively}.
**Key observation**: Subtrees of the same depth can be recursively characterized from the leaf nodes to the root nodes.

From leaves to the root:
- (2 neighbors, 3 neighbors)

From leaves to the root:
- (1 neighbor, 3 neighbors)

Input features are uniform.
If each step of GNN’s aggregation can fully retain the neighboring information, the generated node embeddings can distinguish different rooted subtrees.

- Fully retain neighboring information

```
1 (2 neighbors, 3 neighbors) ≠ (1 neighbor, 3 neighbors)
```

- Input features are uniform

```
2 neighbors: 2 neighbors, 3 neighbors
1 5 1 2 4

3 neighbors: 1 neighbor, 3 neighbors
4 1 2 4
```

In other words, most expressive GNN would use an **injective neighbor aggregation** function at each step.

- Maps different neighbors to different embeddings.
**Summary so far**

- To generate a node embedding, GNNs use a computational graph corresponding to a **subtree** rooted around each node.

- GNN can fully distinguish different subtree structures if **every step of its neighbor aggregation** is injective.
Stanford CS224W: Designing the Most Powerful Graph Neural Network

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
Key observation: Expressive power of GNNs can be characterized by that of neighbor aggregation functions they use.

- A more expressive aggregation function leads to a more expressive a GNN.
- **Injective aggregation function** leads to the most expressive GNN.

Next:

- Theoretically analyze expressive power of aggregation functions.
Observation: Neighbor aggregation can be abstracted as a **function over a multi-set** (a set with repeating elements).

Examples of multi-set

Same color indicates the same features.
Next: We analyze aggregation functions of two popular GNN models

- **GCN (mean-pool)** [Kipf & Welling, ICLR 2017]
  - Uses **element-wise** mean pooling over neighboring node features
    \[
    \text{Mean}(\{x_u\}_{u \in N(v)})
    \]

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - Uses **element-wise** max pooling over neighboring node features
    \[
    \text{Max}(\{x_u\}_{u \in N(v)})
    \]
GCN (mean-pool) [Kipf & Welling ICLR 2017]

- Take element-wise mean, followed by linear function and ReLU activation, i.e., \( \max(0, x) \).

**Theorem** [Xu et al. ICLR 2019]

- GCN’s aggregation function cannot distinguish different multi-sets with the same color proportion.

**Failure case**

**Why?**
For simplicity, we assume node features (colors) are represented by one-hot encoding.

**Example:** If there are two distinct colors:

\[
\begin{align*}
\text{\textbullet} & = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
\text{\textbullet} & = \begin{pmatrix} 0 \\ 1 \end{pmatrix}
\end{align*}
\]

This assumption is sufficient to illustrate how GCN fails.
**Neighbor Aggregation: Case Study**

- **GCN** (mean-pool) [Kipf & Welling ICLR 2017]
  - **Failure case illustration**

![Same outputs!](image)

**Element-wise-mean-pool**

\[
\begin{pmatrix}
0.5 \\
0.5 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 \\
0 \\
0 \\
1 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
0.5 \\
0.5 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 \\
1 \\
0 \\
1 \\
0 \\
\end{pmatrix}
\]
**GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]

- Apply an MLP, then take *element-wise max*.
- **Theorem** [Xu et al. ICLR 2019]
  - GraphSAGE’s aggregation function cannot distinguish different multi-sets with the same set of distinct colors.

**Failure case**

**Why?**
**GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]

- **Failure case illustration**

The same outputs!

Element-wise-max-pool

For simplicity, assume the one-hot encoding after **MLP**.

Input:

1. **MLP**:
   - Input: \((1, 0, 1, 0, 1)\)
   - Output: \((1, 1, 1, 1, 1)\)

Output:

1. **MLP**:
   - Input: \((1, 0, 1, 0, 1)\)
   - Output: \((1, 1, 1, 1, 1)\)
We analyzed the **expressive power of GNNs.**

**Main takeaways:**

- Expressive power of GNNs can be characterized by that of the neighbor aggregation function.
- Neighbor aggregation is a function over multi-sets (sets with repeating elements)
- GCN and GraphSAGE’s aggregation functions fail to distinguish some basic multi-sets; hence not injective.
- Therefore, GCN and GraphSAGE are not maximally powerful GNNs.
Our goal: Design maximally powerful GNNs in the class of message-passing GNNs.

This can be achieved by designing injective neighbor aggregation function over multisets.

Here, we design a neural network that can model injective multiset function.
Theorem [Xu et al. ICLR 2019]

Any injective multi-set function can be expressed as:

\[ S : \text{multi-set} \]

\[ \Phi \left( \sum_{x \in S} f(x) \right) \]

Some non-linear function

Sum over multi-set

\[ f(\bullet) + f(\circ) + f(\circ) \]
Proof Intuition: [Xu et al. ICLR 2019]

$f$ produces one-hot encodings of colors. Summation of the one-hot encodings retains all the information about the input multi-set.

$$\Phi \left( \sum_{x \in S} f(x) \right)$$

Example:

$$\Phi \left[ f(\text{\color{orange}1}) + f(\text{\color{blue}0}) + f(\text{\color{blue}1}) \right]$$

One-hot

$$\begin{pmatrix}
1 \\
0 \\
0
\end{pmatrix} +
\begin{pmatrix}
0 \\
1 \\
1
\end{pmatrix} +
\begin{pmatrix}
0 \\
1 \\
1
\end{pmatrix} =
\begin{pmatrix}
1 \\
2
\end{pmatrix}$$
How to model $\Phi$ and $f$ in $\Phi(\sum_{x \in S} f(x))$?

We use a Multi-Layer Perceptron (MLP).

**Theorem: Universal Approximation Theorem**

[Hornik et al., 1989]

- 1-hidden-layer MLP with sufficiently-large hidden dimensionality and appropriate non-linearity $\sigma(\cdot)$ (including ReLU and sigmoid) can **approximate any continuous function to an arbitrary accuracy**.
We have arrived at a neural network that can model any injective multiset function.

In practice, MLP hidden dimensionality of 100 to 500 is sufficient.
Most Expressive GNN

- **Graph Isomorphism Network (GIN)** [Xu et al. ICLR 2019]
  - Apply an MLP, element-wise **sum**, followed by another MLP.
    
    $$\text{MLP}_\Phi \left( \sum_{x \in S} \text{MLP}_f(x) \right)$$

- **Theorem** [Xu et al. ICLR 2019]
  - GIN’s neighbor aggregation function is injective.
  - No failure cases!
  - **GIN is THE most expressive GNN** in the class of message-passing GNNs we have introduced!
So far: We have described the neighbor aggregation part of GIN.

We now describe the full model of GIN by relating it to WL graph kernel (traditional way of obtaining graph-level features).

We will see how GIN is a “neural network” version of the WL graph kernel.
Recall: Color refinement algorithm in WL kernel.

- **Given:** A graph $G$ with a set of nodes $V$.
  - Assign an initial color $c^{(0)}(v)$ to each node $v$.
  - Iteratively refine node colors by:
    \[
    c^{(k+1)}(v) = \text{HASH} \left( c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)} \right),
    \]
    where HASH maps different inputs to different colors.
  - After $K$ steps of color refinement, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood.
Example of color refinement given two graphs

- Assign initial colors

- Aggregate neighboring colors
Example of color refinement given two graphs

- **Aggregated colors:**

  ![Graph 1]

  ![Graph 2]

- **Injectively** HASH the aggregated colors

  ![HASH table]

  **HASH table:** *Injective!*

  - 1,1   -->   2
  - 1,11  -->   3
  - 1,111 -->   4
  - 1,1111 -->  5
Example of color refinement given two graphs

- Process continues until a stable coloring is reached
- Two graphs are considered **isomorphic** if they have the same set of colors.
GIN uses a neural network to model the injective HASH function.

\[ c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right) \]

Specifically, we will model the injective function over the tuple:

\[ (c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}) \]

Root node features | Neighboring node colors
Theorem (Xu et al. ICLR 2019)

Any injective function over the tuple

\[(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)})\]

of neighboring node features can be modeled as

\[
\text{MLP}_\Phi \left( (1 + \epsilon) \cdot \text{MLP}_f(c^{(k)}(v))) + \sum_{u \in N(v)} \text{MLP}_f(c^{(k)}(u)) \right)
\]

where \(\epsilon\) is a learnable scalar.
If input feature $c^{(0)}(v)$ is represented as one-hot, **direct summation is injective**.

Example:

$$\Phi \left( \begin{array}{c} 1 \\ 0 \\ 0 \end{array} \right) + \left( \begin{array}{c} 0 \\ 1 \\ 1 \end{array} \right) + \left( \begin{array}{c} 0 \\ 1 \end{array} \right) = \left( \begin{array}{c} 1 \\ 2 \end{array} \right)$$

- We only need $\Phi$ to ensure the injectivity.
GIN’s node embedding updates

**Given:** A graph $G$ with a set of nodes $V$.

- Assign an initial vector $c^{(0)}(v)$ to each node $v$.
- Iteratively update node vectors by

$$c^{(k+1)}(v) = \text{GINConv} \left( \left\{ c^{(k)}(v), \left\{ c^{(k)}(u) \right\}_{u \in N(v)} \right\} \right),$$

where $\text{GINConv}$ maps different inputs to different embeddings.

- After $K$ steps of GIN iterations, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood.
GIN and WL Graph Kernel

- **GIN** can be understood as differentiable neural version of the WL graph Kernel:

<table>
<thead>
<tr>
<th>Update target</th>
<th>Update function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL Graph Kernel</td>
<td>Node colors (one-hot)</td>
</tr>
<tr>
<td>GIN</td>
<td>Node embeddings (low-dim vectors)</td>
</tr>
</tbody>
</table>

- **Advantages of GIN over the WL graph kernel are:**
  - Node embeddings are **low-dimensional**; hence, they can capture the fine-grained similarity of different nodes.
  - Parameters of the update function can be **learned for the downstream tasks**.
Because of the relation between GIN and the WL graph kernel, their expressive is exactly the same.

- If two graphs can be distinguished by GIN, they can be also distinguished by the WL kernel, and vice versa.

How powerful is this?

- WL kernel has been both theoretically and empirically shown to distinguish most of the real-world graphs [Cai et al. 1992].

- Hence, GIN is also powerful enough to distinguish most of the real graphs!
Discussion: The Power of Pooling

Failure cases for mean and max pooling:

(a) Mean and Max both fail
(b) Max fails
(c) Mean and Max both fail

Colors represent feature values

Ranking by discriminative power:

Input sum - multiset mean - distribution max - set
Can the expressive power of GNNs be improved?

- There are basic graph structures that existing GNN framework cannot distinguish, such as difference in cycles.

- GNNs’ expressive power can be improved to resolve the above problem. [You et al. AAAI 2021, Li et al. NeurIPS 2020]

- Stay tuned for Lecture 15: Advanced Topics in GNNs
We design a neural network that can model injective multi-set function. We use the neural network for neighbor aggregation function and arrive at GIN---the most expressive GNN model. The key is to use element-wise sum pooling, instead of mean-/max-pooling. GIN is closely related to the WL graph kernel. Both GIN and WL graph kernel can distinguish most of the real graphs!
Stanford CS224W: When Things Don’t Go As Planned

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
General Tips

- **Data preprocessing is important:**
  - Node attributes can vary a lot! Use **normalization**
    - E.g. probability ranges (0,1), but some inputs could have much larger range, say (−1000, 1000)
- **Optimizer:** ADAM is relatively robust to learning rate
- **Activation function**
  - ReLU activation function often works well
  - Other good alternatives: [LeakyReLU](#), [PReLU](#)
  - No activation function at your output layer
  - Include bias term in every layer
- **Embedding dimensions:**
  - 32, 64 and 128 are often good starting points
Debug issues: Loss/accuracy not converging during training
• Check pipeline (e.g. in PyTorch we need `zero_grad`)
• Adjust hyperparameters such as learning rate
• Pay attention to weight parameter initialization
• Scrutinize loss function!

Important for model development:
• Overfit on (part of) training data:
  • With a small training dataset, loss should be essentially close to 0, with an expressive neural network
• Monitor the training & validation loss curve
GraphGym: Easy and flexible end-to-end GNN pipeline based on PyTorch Geometric (PyG)

Auto-differentiation frameworks

DGL

GraphNets

GNN frameworks:
Implements a variety of GNN architectures
Resources on Graph Neural Networks

Tutorials and overviews:
▪ Relational inductive biases and graph networks (Battaglia et al., 2018)
▪ Representation learning on graphs: Methods and applications (Hamilton et al., 2017)

Attention-based neighborhood aggregation:
▪ Graph attention networks (Hoshen, 2017; Velickovic et al., 2018; Liu et al., 2018)

Embedding entire graphs:
▪ Graph neural nets with edge embeddings (Battaglia et al., 2016; Gilmer et. al., 2017)
▪ Embedding entire graphs (Duvenaud et al., 2015; Dai et al., 2016; Li et al., 2018) and graph pooling (Ying et al., 2018, Zhang et al., 2018)
▪ Graph generation and relational inference (You et al., 2018; Kipf et al., 2018)
▪ How powerful are graph neural networks(Xu et al., 2017)

Embedding nodes:
▪ Varying neighborhood: Jumping knowledge networks (Xu et al., 2018), GeniePath (Liu et al., 2018)
▪ Position-aware GNN (You et al. 2019)

Spectral approaches to graph neural networks:
▪ Spectral graph CNN & ChebNet (Bruna et al., 2015; Defferrard et al., 2016)
▪ Geometric deep learning (Bronstein et al., 2017; Monti et al., 2017)

Other GNN techniques:
▪ Pre-training Graph Neural Networks (Hu et al., 2019)
▪ GNNExplainer: Generating Explanations for Graph Neural Networks (Ying et al., 2019)