ANNOUNCEMENTS

• Next Thursday (10/12): Colab 1 due
Recap: Node Embeddings

- **Intuition:** Map nodes to $d$-dimensional embeddings such that similar nodes in the graph are embedded close together.

$$f() = \text{Input graph} \Rightarrow \text{2D node embeddings}$$

How to **learn** mapping function $f$?
Recap: Node Embeddings

Goal: \( \text{similarity}(u, v) \approx z_v^T z_u \)

Need to define!

Input network

\( \text{ENC}(u) \)

\( \text{ENC}(v) \)

d-dimensional embedding space
**Recap: Two Key Components**

- **Encoder**: Maps each node to a low-dimensional vector
  
  \[
  \text{ENC}(v) = z_v
  \]

  node in the input graph

- **Similarity function**: Specifies how the relationships in vector space map to the relationships in the original network
  
  \[
  \text{similarity}(u, v) \approx z_v^T z_u
  \]

  dot product between node embeddings

**Similarity of** \( u \) **and** \( v \) **in the original network**
Simplest encoding approach: **Encoder is just an embedding-lookup**

embedding matrix

\[ \mathbf{Z} = \]  

embedding vector for a specific node

Dimension/size of embeddings

one column per node
Limitations of shallow embedding methods:

- \( O(|V|d) \) parameters are needed:
  - No sharing of parameters between nodes
  - Every node has its own unique unique embedding

- Inherently “transductive”:
  - Cannot generate embeddings for nodes that are not seen during training

- Do not incorporate node features:
  - Nodes in many graphs have features that we can and should leverage
Today: We will now discuss deep learning methods based on graph neural networks (GNNs):

Note: All these deep encoders can be combined with node similarity functions defined in the Lecture 3.
Deep Graph Encoders

Graph convolutions

Regularization, e.g., dropout

Graph convolutions

Output: Node embeddings. Also, we can embed subgraphs, and graphs.
Tasks we will be able to solve:

- **Node classification**
  - Predict the type of a given node
- **Link prediction**
  - Predict whether two nodes are linked
- **Community detection**
  - Identify densely linked clusters of nodes
- **Network similarity**
  - How similar are two (sub)networks
Modern deep learning toolbox is designed for simple sequences & grids.
But networks are far more complex!

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)
- No fixed node ordering or reference point
- Often dynamic and have multimodal features
Outline of Today’s Lecture

1. Basics of deep learning
2. Deep learning for graphs
3. Graph Convolutional Networks
4. GNNs subsume CNNs
Summary: Basics of Deep Learning

- **Loss function:**
  \[
  \min_{\Theta} \mathcal{L}(y, f_{\Theta}(x))
  \]

- \(f\) can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)

- Sample a minibatch of input \(x\)

- **Forward propagation:** Compute \(\mathcal{L}\) given \(x\)

- **Back-propagation:** Obtain gradient \(\nabla_{\Theta} \mathcal{L}\) using a chain rule.

- Use **stochastic gradient descent (SGD)** to optimize \(\mathcal{L}\) for \(\Theta\) over many iterations.
Stanford CS224W: Deep Learning for Graphs

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

- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs

- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training
Assume we have a graph $G$:
- $V$ is the vertex set
- $A$ is the adjacency matrix (assume binary)
- $X \in \mathbb{R}^{|V| \times m}$ is a matrix of node features
- $\nu$: a node in $V$; $N(\nu)$: the set of neighbors of $\nu$.

Node features:
- Social networks: User profile, User image
- Biological networks: Gene expression profiles, gene functional information
- When there is no node feature in the graph dataset:
  - Indicator vectors (one-hot encoding of a node)
  - Vector of constant 1: $[1, 1, \ldots, 1]$
A Naïve Approach

- Join adjacency matrix and features
- Feed them into a deep neural net:

- Issues with this idea:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering
Goal is to generalize convolutions beyond simple lattices.
Leverage node features/attributes (e.g., text, images).
Real-World Graphs

But our graphs look like this:

- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

Permutation Invariance

- Graph does not have a canonical order of the nodes!
- We can have many different order plans.
Graph does not have a canonical order of the nodes!

Order plan 1

Node features $X_1$

Adjacency matrix $A_1$
Graph does not have a canonical order of the nodes!

Order plan 1

Order plan 2

Node features $X_1$

Node features $X_2$

Adjacency matrix $A_1$

Adjacency matrix $A_2$
Permutation Invariance

- Graph does not have a canonical order of the nodes!

Order plan 1

Graph and node representations should be the same for Order plan 1 and Order plan 2.
What does it mean by “graph representation is same for two order plans”?

- Consider we learn a function $f$ that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$ then
  \[ f(A_1, X_1) = f(A_2, X_2) \]

In other words, $f$ maps a graph to a $d$-dim embedding.

Order plan 1: $A_1, X_1$

Order plan 2: $A_2, X_2$

For two order plans, output of $f$ should be the same!
What does it mean by “graph representation is same for two order plans”? 

Consider we learn a function $f$ that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$. $A$ is the adjacency matrix, $X$ is the node feature matrix.

Then, if $f(A_i, X_i) = f(A_j, X_j)$ for any order plan $i$ and $j$, we formally say $f$ is a permutation invariant function.

For a graph with $|V|$ nodes, there are $|V|!$ different order plans.

Definition: For any graph function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^d$, $f$ is permutation-invariant if $f(A, X) = f(PAP^T, PX)$ for any permutation $P$.

$m$... each node has a $m$-dim feature vector associated with it.

Examples: $(A, B, C) \rightarrow (B, C, A)$
For node representation: We learn a function $f$ that maps nodes of $G$ to a matrix $\mathbb{R}^{|V| \times d}$. In other words, each node in $V$ is mapped to a $d$-dim embedding.

Order plan 1: $A_1, X_1$

Order plan 2: $A_2, X_2$

$f(A_1, X_1) = \begin{bmatrix}
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
\end{bmatrix}$

$f(A_2, X_2) = \begin{bmatrix}
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
A & B & C & D & E & F \\
\end{bmatrix}$
For node representation: We learn a function $f$ that maps nodes of $G$ to a matrix $\mathbb{R}^{|V| \times d}$.

Order plan 1: $A_1, X_1$

Order plan 2: $A_2, X_2$

For two order plans, the vector of node at the same position in the graph is the same!
For node representation: We learn a function \( f \) that maps nodes of \( G \) to a matrix \( \mathbb{R}^{\mid V \mid \times d} \).

Order plan 1: \( A_1, X_1 \)

Order plan 2: \( A_2, X_2 \)

For two order plans, the vector of node at the same position in the graph is the same!

\[
f(A_1, X_1) = \text{Representation vector of the green node C}
\]

\[
f(A_2, X_2) = \text{Representation vector of the green node D}
\]
For node representation:

- Consider we learn a function $f$ that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{|V| \times d}$.

- If the output vector of a node at the same position in the graph remains unchanged for any order plan, we say $f$ is permutation equivariant.

**Definition:** For any node function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^{|V| \times d}$, $f$ is permutation-equivariant if $Pf(A, X) = f(PAP^T, PX)$ for any permutation $P$. $f$ maps each node in $V$ to a $d$-dim embedding.
Summary: Invariance and Equivariance

- **Permutation-invariant**
  \[ f(A, X) = f(PAP^T, PX) \]
  
  Permute the input, the output stays the same.
  (map a graph to a vector)

- **Permutation-equivariant**
  \[ Pf(A, X) = f(PAP^T, PX) \]
  
  Permute the input, output also permutes accordingly.
  (map a graph to a matrix)

- **Examples:**
  - \( f(A, X) = 1^TX \) : Permutation-invariant
    - Reason: \( f(PAP^T, PX) = 1^TPX = 1^TX = f(A, X) \)
  - \( f(A, X) = X \) : Permutation-equivariant
    - Reason: \( f(PAP^T, PX) = PX = Pf(A, X) \)
  - \( f(A, X) = AX \) : Permutation-equivariant
    - Reason: \( f(PAP^T, PX) = PAP^TPX = Pax = Pf(A, X) \)
Graph neural networks consist of multiple permutation equivariant / invariant functions.
Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

- No.

Switching the order of the input leads to different outputs!
Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

- No.

This explains why the naïve MLP approach fails for graphs!
Are any neural network architectures, e.g.,

Next: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors!
Outline of Today’s Lecture

1. Basics of deep learning ✓
2. Deep learning for graphs ✓
3. Graph Convolutional Networks
4. GNNs subsume CNNs
Idea: Node’s neighborhood defines a computation graph

- Determine node computation graph
- Propagate and transform information

Learn how to propagate information across the graph to compute node features
- **Key idea**: Generate node embeddings based on local network neighborhoods
**Intuition:** Nodes aggregate information from their neighbors using neural networks
**Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!
Deep Model: Many Layers

- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node $v$ is its input feature, $x_v$
  - Layer-$k$ embedding gets information from nodes that are $k$ hops away
Neighborhood aggregation: Key distinctions are in how different approaches aggregate information across the layers.

What is in the box?
**Basic approach:** Average information from neighbors and apply a neural network

1. Average messages from neighbors
2. Apply neural network
**Basic approach:** Average neighbor messages and apply a neural network

Initial 0-th layer embeddings are equal to node features

Embedding of \( \nu \) at layer \( k \)

Total number of layers

Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.
What are the invariance and equivariance properties for a GCN?

- **Given a node**, the GCN that computes its embedding is permutation invariant.

![Diagram showing GCN architecture and invariance properties.]
Considering all nodes in a graph, GCN computation is permutation equivariant.

Order plan 1

Order plan 2

Permute the input, the output also permutes accordingly - permutation equivariant.
GCN: Invariance and Equivariance

- Considering all nodes in a graph, GCN computation is permutation equivariant

Detailed reasoning:
1. The rows of input node features and output embeddings are aligned
2. We know computing the embedding of a given node with GCN is invariant.
3. So, after permutation, the location of a given node in the input node feature matrix is changed, and the output embedding of a given node stays the same (the colors of node feature and embedding are matched)

This is permutation equivariant

Permute the input, the output also permutes accordingly - permutation equivariant
How do we train the GCN to generate embeddings?

Need to define a loss function on the embeddings.
Model Parameters

$h_v^{(0)} = x_v$

$h_v^{(k+1)} = \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}), \forall k \in \{0..K - 1\}$

$z_v = h_v^{(K)}$

Trainable weight matrices (i.e., what we learn)

Final node embedding

We can feed these embeddings into any loss function and run SGD to train the weight parameters

$h_v^k$: the hidden representation of node $v$ at layer $k$
- $W_k$: weight matrix for neighborhood aggregation
- $B_k$: weight matrix for transforming hidden vector of self
Many aggregations can be performed efficiently by (sparse) matrix operations

Let \( H^{(k)} = [h_1^{(k)} \ldots h_{|\mathcal{V}|}^{(k)}]^T \)

Then:
\[
\sum_{u \in \mathcal{N}_v} h_u^{(k)} = A_v : H^{(k)}
\]

Let \( D \) be diagonal matrix where
\( D_{v,v} = \text{Deg}(v) = |N(v)| \)

The inverse of \( D \): \( D^{-1} \) is also diagonal:
\[ D_{v,v}^{-1} = 1/|N(v)| \]

Therefore,
\[
\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \quad \Rightarrow \quad H^{(k+1)} = D^{-1} A H^{(k)}
\]
**Matrix Formulation (2)**

- Re-writing update function in matrix form:
  \[
  H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)
  \]
  where \( \tilde{A} = D^{-1}A \)

  - Red: neighborhood aggregation
  - Blue: self transformation

- In practice, this implies that efficient sparse matrix multiplication can be used (\( \tilde{A} \) is sparse)

- **Note**: not all GNNs can be expressed in a simple matrix form, when aggregation function is complex
How to Train A GNN

- Node embedding $z_v$ is a function of input graph
- **Supervised setting**: We want to minimize loss $\mathcal{L}$:
  \[
  \min_{\theta} \mathcal{L}(y, f_{\theta}(z_v))
  \]
  - $y$: node label
  - $\mathcal{L}$ could be L2 if $y$ is real number, or cross entropy if $y$ is categorical
- **Unsupervised setting**:  
  - No node label available
  - Use the graph structure as the supervision!
Unsupervised Training

- **One possible idea:** “Similar” nodes have similar embeddings:

  \[
  \min_{\theta} \mathcal{L} = \sum_{z_u,z_v} CE(y_{u,v}, DEC(z_u, z_v))
  \]

  - where \(y_{u,v} = 1\) when node \(u\) and \(v\) are similar
  - \(z_u = f_\theta(u)\) and \(DEC(\cdot, \cdot)\) is the dot product
  - \(CE\) is the cross entropy loss:

    \[
    CE(y, f(x)) = -\sum_{i=1}^{C} (y_i \log f_\theta(x)_i)
    \]

    - \(y_i\) and \(f_\theta(x)_i\) are the actual and predicted values of the \(i\)-th class.
    - **Intuition:** the lower the loss, the closer the prediction is to one-hot

  - **Node similarity** can be anything from Lecture 2, e.g., a loss based on:
    - **Random walks** (node2vec, DeepWalk, struc2vec)
    - **Matrix factorization**
Directly train the model for a supervised task (e.g., node classification)

E.g., a drug-drug interaction network
Directly train the model for a supervised task (e.g., node classification)

- Use cross entropy loss (Slide 53)

\[ \mathcal{L} = - \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta)) \]

Encoder output:
node embedding

Classification weights

Node class label

Safe or toxic drug?
(1) Define a neighborhood aggregation function

(2) Define a loss function on the embeddings
(3) Train on a set of nodes, i.e., a batch of compute graphs
Model Design: Overview

(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!
The same aggregation parameters are shared for all nodes:

- The number of model parameters is sublinear in $|V|$ and we can generalize to unseen nodes!
Inductive node embedding

Train on one graph

Generalize to new graph

Inductive node embedding  Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B
Inductive Capability: **New Nodes**

- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
  - Need to generate new embeddings “on the fly”
Outline of Today’s Lecture

1. Basics of deep learning
2. Deep learning for graphs
3. Graph Convolutional Networks
4. GNNs subsume CNNs
Architecture Comparison

- How do GNNs compare to prominent architectures such as Convolutional Neural Nets?
Convolutional neural network (CNN) layer with 3x3 filter:

CNN formulation: $h^{(l+1)}_v = \sigma(\sum_{u \in N(v) \cup \{v\}} W^u_l h^{(l)}_u), \quad \forall l \in \{0, ..., L - 1\}$

$N(v)$ represents the 8 neighbor pixels of $v$. 
**GNN vs. CNN**

Convolutional neural network (CNN) layer with 3x3 filter:

![Image of a CNN layer](image)

- **GNN formulation:**
  \[ h^{(l+1)}_v = \sigma(W_l \sum_{u \in N(v)} \frac{h^{(l)}_u}{|N(v)|} + B_l h^{(l)}_v), \forall l \in \{0, ..., L - 1\} \]

- **CNN formulation:** (previous slide)
  \[ h^{(l+1)}_v = \sigma(\sum_{u \in N(v) \cup \{v\}} W^u_l h^{(l)}_u), \forall l \in \{0, ..., L - 1\} \]
  \[ h^{(l+1)}_v = \sigma(\sum_{u \in N(v)} W^u_l h^{(l)}_u + B_l h^{(l)}_v), \forall l \in \{0, ..., L - 1\} \]
GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:

**Image**

**Graph**

GNN formulation:
\[
h^{(l+1)}_v = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, ..., L - 1\}
\]

CNN formulation:
\[
h^{(l+1)}_v = \sigma(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, ..., L - 1\}
\]

**Key difference:** We can learn different \( W_l^u \) for different “neighbor” \( u \) for pixel \( v \) on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: \{(-1,-1), (-1,0), (-1, 1), ..., (1, 1)\}
Convolutional neural network (CNN) layer with 3x3 filter:

- CNN can be seen as a special GNN with fixed neighbor size and ordering:
  - The size of the filter is pre-defined for a CNN.
  - The advantage of GNN is it processes arbitrary graphs with different degrees for each node.
- CNN is not permutation invariant/equivariant.
  - Switching the order of pixels leads to different outputs.

**Key difference:** We can learn different $W_i^u$ for different “neighbor” $u$ for pixel $v$ on the image. The reason is we can pick an order for the 9 neighbors using relative position to the center pixel: {(-1,-1), (-1,0), (-1, 1), ..., (1, 1)}
Transformer

Transformer is one of the most popular architectures that achieves great performance in many sequence modeling tasks.

Key component: self-attention
- Every token/word attends to all the other tokens/words via matrix calculation.
A general definition of attention:

Given a set of vector \textit{values}, and a vector \textit{query}, \textit{attention} is a technique to compute a weighted sum of the values, dependent on the query.

Each token/word has a \textbf{value vector} and a \textbf{query vector}. The value vector can be seen as the representation of the token/word. We use the query vector to calculate the attention score (weights in the weighted sum).
Transformer layer can be seen as a special GNN that runs on a fully-connected “word” graph!

Since each word attends to all the other words, the computation graph of a transformer layer is identical to that of a GNN on the fully-connected “word” graph.
In this lecture, we introduced

- Idea for Deep Learning for Graphs
  - Multiple layers of embedding transformation
  - At every layer, use the embedding at previous layer as the input
  - Aggregation of neighbors and self-embeddings

- Graph Convolutional Network
  - Mean aggregation; can be expressed in matrix form

- GNN is a general architecture
  - CNN can be viewed as a special GNN