ANNOUNCEMENTS

• Today (01/19): HW 1 out
• Monday (01/23): Recitation session for HW 1
• Next Thursday (01/26): Colab 1 due, Colab 2 out
**Intuition:** Map nodes to $d$-dimensional embeddings such that similar nodes in the graph are embedded close together.

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

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
  \[ \text{similarity}(u, v) \approx z_v^T z_u \]
  Similarity of \( u \) and \( v \) in the original network
  dot product between node embeddings

**Decoder**
Simplest encoding approach: **Encoder is just an embedding-lookup**

$$Z = \text{embedding matrix}$$

- **Dimension/size of embeddings**
- **one column per node**
- **embedding vector for a specific 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: Deep Graph Encoders

- **Today:** We will now discuss deep learning methods based on graph neural networks (GNNs):

  \[
  ENC(\nu) = \text{multiple layers of non-linear transformations based on graph structure}
  \]

- **Note:** All these deep encoders can be combined with node similarity functions defined in the Lecture 3.
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
Supervised learning: we are given input $x$, and the goal is to predict label $y$.

Input $x$ can be:
- Vectors of real numbers
- Sequences (natural language)
- Matrices (images)
- Graphs (potentially with node and edge features)

We formulate the task as an optimization problem.
Formulate the task as an optimization problem:

$$\min_{\Theta} \mathcal{L}(y, f(x))$$

- $\Theta$: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices ...
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)

- $\mathcal{L}$: **loss function**. Example: L2 loss
  $$\mathcal{L}(y, f(x)) = \|y - f(x)\|_2$$
  - Other common loss functions:
    - L1 loss, huber loss, max margin (hinge loss), cross entropy ...
Each layer of MLP combines linear transformation and non-linearity:

\[ x^{(l+1)} = \sigma(W_l x^{(l)} + b^l) \]

- where \( W_l \) is weight matrix that transforms hidden representation at layer \( l \) to layer \( l + 1 \)
- \( b^l \) is bias at layer \( l \), and is added to the linear transformation of \( x^{(l)} \)
- \( \sigma \) is non-linearity function (e.g., sigmoid)

Suppose \( x \) is 2-dimensional, with entries \( x_1 \) and \( x_2 \)

Every layer:
Linear transformation + non-linearity
Objective function:
\[
\min_{\Theta} \mathcal{L}(y, f(x))
\]

- \( f \) 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 \( \nabla_w \mathcal{L} \) using a chain rule.
- Use **stochastic gradient descent (SGD)** to optimize for \( \Theta \) over many iterations.
Outline of Today’s Lecture

1. Basics of deep learning
   ✓

2. Deep learning for graphs

3. Graph Convolutional Networks

4. GNNs subsume CNNs
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)
- $X \in \mathbb{R}^{|V| \times d}$ 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)
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
- 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

$$f(A_1, X_1) = f(A_2, X_2)$$

$A$ is the adjacency matrix

$X$ is the node feature matrix

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.

- **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 \( P \). Permutation \( P \): a shuffle of the node order.

Example: \( (A,B,C) \rightarrow (B,C,A) \).
For node representation: We learn a function $f$ that maps nodes of $G$ to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: $A_1, X_1$  

Order plan 2: $A_2, X_2$

$$f(A_1, X_1) = $$

$$f(A_2, X_2) = $$
For node representation: We learn a function $f$ that maps nodes of $G$ to a matrix $\mathbb{R}^{m \times d}$.

**Order plan 1: $A_1, X_1$**

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

**Order plan 2: $A_2, X_2$**

$\mathbf{f}(A_2, X_2) = \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \end{bmatrix}$

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 $\mathbb{R}^{m \times d}$.

**Order plan 1: $A_1, X_1$**

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

**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

- Consider we learn a function $f$ that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{m \times d}$
- 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 $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^{|V| \times m}$, $f$ is permutation-equivariant if $Pf(A, X) = f(PAP^T, PX)$ for any permutation $P$. 
### Summary: Invariance and Equivariance

- **Permutation-invariant**
  \[ f(A, X) = f(PAP^T, PX) \]

- **Permutation-equivariant**
  \[ Pf(A, X) = f(PAP^T, PX) \]

- **Examples:**
  - \[ f(A, X) = 1^T X : \text{Permutation-invariant} \]
    - Reason: \[ f(PAP^T, PX) = 1^T PX = 1^T X = f(A, X) \]
  - \[ f(A, X) = X : \text{Permutation-equivariant} \]
    - Reason: \[ f(PAP^T, PX) = PX = Pf(A, X) \]
  - \[ f(A, X) = AX : \text{Permutation-equivariant} \]
    - Reason: \[ f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X) \]
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 architecture, 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.

**Idea: Aggregate Neighbors**

- **TARGET NODE**
- **INPUT GRAPH**
- **Neural networks**
Intuition: Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!
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.
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.

\[ h_{v}^{0} = x_{v} \]

Embedding after \( K \) layers of neighborhood aggregation.

\[ h_{v}^{(k+1)} = \sigma(W_{k} \sum_{u \in N(v)} \frac{h_{u}^{(k)}}{|N(v)|}) + B_{k} h_{v}^{(k)}), \forall k \in \{0, \ldots, K - 1\} \]

Notice summation is a permutation invariant pooling/aggregation.

\[ z_{v} = h_{v}^{(K)} \]

Non-linearity (e.g., ReLU)
What are the **invariance** and **equivariance** properties for a GCN?

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

![Diagram of GCN](image)

- **Average** of neighbor’s previous layer embeddings - **Permutation invariant**
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.
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
How do we train the GCN to generate embeddings?

Need to define a loss function on the embeddings.
We can feed these embeddings into any loss function and run SGD to train the weight 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

\( h_v^k \): 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 $H^{(k)} = [h^{(k)}_1 \ldots h^{(k)}_{|V|}]^T$

Then: $\sum_{u \in N(v)} h^{(k)}_u = A \cdot v \cdot H^{(k)}$

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

- The inverse of $D$: $D^{-1}$ is also diagonal: $D^{-1}_{v,v} = 1/|N(v)|$

Therefore,

$$\sum_{u \in N(v)} \frac{h^{(k-1)}_u}{|N(v)|} \quad \rightarrow \quad H^{(k+1)} = D^{-1} A H^{(k)}$$
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 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 the loss $\mathcal{L}$ (see also Slide 15):
  \[
  \min_{\Theta} \mathcal{L}(y, f(z_v))
  \]
  - $y$: 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!
“Similar” nodes have similar embeddings

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

- Where \( y_{u,v} = 1 \) when node \( u \) and \( v \) are similar
- \( CE \) is the cross entropy (Slide 16)
- \( DEC \) is the decoder such as inner product (Lecture 4)

**Node similarity** can be anything from Lecture 3, e.g., a loss based on:

- **Random walks** (node2vec, DeepWalk, struc2vec)
- **Matrix factorization**
- **Node proximity in the graph**
Supervised Training

Directly train the model for a supervised task (e.g., node classification)

Safe or toxic drug?

E.g., a drug-drug interaction network
Supervised Training

Directly train the model for a supervised task (e.g., node classification)
- Use cross entropy loss (Slide 16)

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

Encoder output: node embedding

Safe or toxic drug?

Classification weights

Node class label
Model Design: Overview

(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
(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 Capability: New Graphs**

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

$$h_v^{(l+1)} = \sigma \left( \sum_{u \in N(v) \cup \{v\}} W_l^{u} h_u^{(l)} \right), \quad \forall l \in \{0, ..., L - 1\}$$

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

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

- **GNN formulation:** $h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$, $\forall l \in \{0, \ldots, L - 1\}$
- **CNN formulation:** (previous slide) $h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)})$, $\forall l \in \{0, \ldots, L - 1\}$
  
  if we rewrite:
  
  $h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)})$, $\forall l \in \{0, \ldots, L - 1\}$
Convolutional neural network (CNN) layer with 3x3 filter:

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

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

**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)\}.
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 will 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)}
In this lecture, we introduced:

- Basics of neural networks
  - Loss, Optimization, Gradient, SGD, non-linearity, MLP
- 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