ANNOUNCEMENTS

• Today (10/07): Colab 1 due, Colab 2 out
• Next Thursday (10/14): HW 1 due, HW 2 out
• Project proposals due on Tuesday 10/19
  o If you are looking for project partners, check out / add yourself to our pinned Ed post ("Project Partner Thread") -- reach out to each other!
  o We strongly encourage groups of 3, but groups of 1 or 2 are allowed
**Intuition:** Map nodes to \( d \)-dimensional embeddings such that similar nodes in the graph are embedded close together

\[
f(\text{Input graph}) = \text{2D node embeddings}
\]

**How to learn mapping function** \( f \)?
Goal: \( \text{similarity}(u, v) \approx z_v^T z_u \)

Need to define!
**Encoder:** Maps each node to a low-dimensional vector

\[ \text{ENC}(v) = \mathbf{z}_v \]

node in the input graph

**Similarity function:** Specifies how the relationships in vector space map to the relationships in the original network

\[ \text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u \]

Similarity of \( u \) and \( v \) in the original network

Decoder

dot product between node embeddings
Simplest encoding approach: **Encoder is just an embedding-lookup**

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

- Dimension/size of embeddings
- One column per node
- Embedding vector for a specific node
Limitations of shallow embedding methods:

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

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

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

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

- **Node classification**
  - Predict a 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 and Transformers
Stanford CS224W: Basics of Deep Learning
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: 

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

**Objective function**

- $\Theta$: 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 ...
One common loss for classification: **cross entropy (CE)**

- Label \( \mathbf{y} \) is a categorical vector (**one-hot encoding**)
  - e.g. \( \mathbf{y} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \) \( \mathbf{y} \) is of class “3”

\[
f(\mathbf{x}) = \text{Softmax}(g(\mathbf{x}))
\]

- Recall from lecture 3: \( f(\mathbf{x})_i = \frac{e^{g(\mathbf{x})_i}}{\sum_{j=1}^{C} e^{g(\mathbf{x})_j}} \)

  where \( C \) is the number of classes.
  - e.g. \( f(\mathbf{x}) = \begin{bmatrix} 0.1 & 0.3 & 0.4 & 0.1 & 0.1 \end{bmatrix} \)

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

  - \( y_i \) and \( f(\mathbf{x})_i \) are the **actual** and **predicted** values of the \( i \)-th class.

  - **Intuition**: the lower the loss, the closer the prediction is to one-hot

**Total loss over all training examples:**

\[
\mathcal{L} = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{T}} \text{CE}(\mathbf{y}, f(\mathbf{x}))
\]

- \( \mathcal{T} \): training set containing all pairs of data and labels \( (\mathbf{x}, \mathbf{y}) \)
Machine Learning as Optimization

- How to optimize the objective function?
- **Gradient vector:** Direction and rate of fastest increase

\[
\nabla_{\Theta} \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \ldots \right)
\]

- \(\Theta_1, \Theta_2 \ldots\) : components of \(\Theta\)
- Recall **directional derivative** of a multi-variable function (e.g. \(\mathcal{L}\)) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the direction of largest increase.

**Gradient Descent**

- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence

\[ \Theta \leftarrow \Theta - \eta \nabla \Theta \mathcal{L} \]

- **Training:** Optimize \(\Theta\) iteratively
  - **Iteration:** 1 step of gradient descent

- **Learning rate (LR) \(\eta\):**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)

- **Ideal termination condition:** gradient = 0
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training).
Problem with gradient descent:

- Exact gradient requires computing $\nabla_{\theta} \mathcal{L}(y, f(x))$, where $x$ is the entire dataset!
  - This means summing gradient contributions over all the points in the dataset
  - Modern datasets often contain billions of data points
  - Extremely expensive for every gradient descent step

Solution: Stochastic gradient descent (SGD)

- At every step, pick a different minibatch $\mathcal{B}$ containing a subset of the dataset, use it as input $x$
Minibatch SGD

- **Concepts:**
  - **Batch size:** the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration:** 1 step of SGD on a minibatch
  - **Epoch:** one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)

- **SGD is unbiased estimator of full gradient:**
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
  - Common optimizer that improves over SGD:
    - Adam, Adagrad, Adadelta, RMSprop ...
Objective: \( \min_{\theta} \mathcal{L}(y, f(x)) \)

In deep learning, function \( f \) can be very complex.

Example:

- To start simple, consider linear function
  \[ f(x) = W \cdot x, \quad \Theta = \{W\} \]
- Then, if \( f \) returns a scalar, then \( W \) is a learnable vector
  \[ \nabla_W f = \left( \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3}, \ldots \right) \]
- But, if \( f \) returns a vector, then \( W \) is the weight matrix
  \[ \nabla_W f = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_2}{\partial w_{11}} \\ \frac{\partial f_1}{\partial w_{12}} & \frac{\partial f_2}{\partial w_{12}} \\ \frac{\partial f_1}{\partial w_{21}} & \frac{\partial f_2}{\partial w_{22}} \end{bmatrix} \]

Jacobian matrix of \( f \)
Goal: \( \min \mathcal{L}(y, f(x)) \)

To minimize \( \mathcal{L} \), we need to evaluate the gradient:

\[
\nabla_W \mathcal{L} = \left( \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3}, \ldots \right)
\]

which means we need to derive derivative of \( \mathcal{L} \).

Overview of Back-propagation:

- \( \mathcal{L} \) is composed from some set of predefined building block functions \( g(\cdot) \)
- For each such \( g \) we also have its derivative \( g' \)
- Then we can automatically compute \( \nabla_{\theta} \mathcal{L} \) by evaluating appropriate funcs. \( g' \) on the minibatch \( \mathcal{B} \).
How about a more complex function:

\[ f(x) = W_2(W_1x), \ \Theta = \{W_1, W_2\} \]

Recall **chain rule**:

\[
\frac{df}{dx} = \frac{dg}{dh} \cdot \frac{dh}{dx} \quad \text{or} \quad f'(x) = g'(h(x))h'(x)
\]

**Example:**

\[
\nabla_x f = \frac{\partial f}{\partial (W_1x)} \cdot \frac{\partial (W_1x)}{\partial x}
\]

**Back-propagation:** Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of \( \mathcal{L} \) w.r.t. \( \Theta \).
**Back-propagation Example (1)**

- **Example**: Simple 2-layer linear network
  - \( f(x) = g(h(x)) = W_2(W_1x) \)

- \( \mathcal{L} = \sum_{(x,y) \in \mathcal{B}} \| (y, -f(x)) \|_2 \)
  - The loss \( \mathcal{L} \) sums the L2 loss in a minibatch \( \mathcal{B} \).

- **Hidden layer**:
  - Intermediate representation of input \( x \)
  - Here we use \( h(x) = W_1x \) to denote the hidden layer
  - \( f(x) = W_2h(x) \)
Back-propagation Example (2)

- **Forward propagation:**
  Compute loss starting from input
  - $x \rightarrow h \rightarrow g \rightarrow \mathcal{L}$
  Multiply $W_1$ Multiply $W_2$ Loss

- **Back-propagation to compute gradient of**
  $\Theta = \{W_1, W_2\}$
  Start from loss, compute the gradient
  $$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}, \quad \frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}$$

Remember:
- $f(x) = W_2(W_1x)$
- $h(x) = W_1x$
- $g(z) = W_2z$

Multiply $W_1$ Multiply $W_2$ Loss

Compute backwards

10/7/21
Note that in $f(x) = W_2(W_1x)$, $W_2W_1$ is another matrix (vector, if we do binary classification)

Hence $f(x)$ is still linear w.r.t. $x$ no matter how many weight matrices we compose.

We introduce non-linearity:

- Rectified linear unit (ReLU)
  $$ReLU(x) = \max(x, 0)$$

- Sigmoid
  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$
Each layer of MLP combines linear transformation and non-linearity:

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

- $W_l$ 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 $x$
- $\sigma$ is non-linearity function (e.g., sigmoid)

Suppose $x$ is 2-dimensional, with entries $x_1$ and $x_2$
Objective function:

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

$\mathcal{L}$ 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 and Transformers
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)
- $X \in \mathbb{R}^{m \times |V|}$ 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.
Permutation Invariance

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

Order plan 1
Permutation Invariance

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

Order plan 1

Order plan 2

Node features $X_1$

Adjacency matrix $A_1$

Node features $X_2$

Adjacency matrix $A_2$
Permutation Invariance

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

Order plan 1

Node feature $X_1$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency matrix $A_1$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order plan 2

Node feature $X_1$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency matrix $A_1$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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 $\mathbb{R}^d$.  
- 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 $m$ nodes, there are $m!$ different order plans.
Similar 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:** $A_2, X_2$

$$f(A_1, X_1) =$$

$$f(A_2, X_2) =$$
Similarly 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: $A_2, X_2$

For two order plans, the vector of node at the same position is the same!
Similarly 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:** $A_2, X_2$

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

- Consider we learn a function $f$ that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{m \times d}$
  - graph has $m$ nodes, each row is the embedding of a node.
- Similarly, if this property holds for any pair of order plan $i$ and $j$, we say $f$ is a permutation equivariant function.
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!
Graph Neural Network Overview

- 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 and Transformers
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
**Idea: Aggregate Neighbors**

- **Key idea:** Generate node embeddings based on local network neighborhoods
**Intuition:** Nodes aggregate information from their neighbors using neural networks.
Idea: Aggregate Neighbors

- **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.

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

TARGET NODE

INPUT GRAPH
**Basic approach:** Average neighbor messages and apply a neural network

- Initial 0-th layer embeddings are equal to node features:
  \[ h_\nu^0 = x_\nu \]

- Average of neighbor's previous layer embeddings:
  \[ h_\nu^{(k+1)} = \sigma(W_k \sum_{u \in N(\nu)} \frac{h_u^{(k)}}{|N(\nu)|}) + B_k h_\nu^{(k)}, \forall k \in \{0, \ldots, K - 1\} \]

- Non-linearity (e.g., ReLU):
  \[ z_\nu = h_\nu^{(K)} \]

- Embedding after L layers of neighborhood aggregation:
  \[ z_\nu = h_\nu^{(K)} \]

- Notice summation is a permutation invariant pooling/aggregation.

- Total number of layers:
  \[ \sum_{u \in N(\nu)} \frac{h_u^{(k)}}{|N(\nu)|} \]

- Non-linearity (e.g., ReLU):
  \[ \sigma(W_k \sum_{u \in N(\nu)} \frac{h_u^{(k)}}{|N(\nu)|}) \]

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

- Initial 0-th layer embeddings are equal to node features:
  \[ h_\nu^0 = x_\nu \]

- Average of neighbor's previous layer embeddings:
  \[ h_\nu^{(k+1)} = \sigma(W_k \sum_{u \in N(\nu)} \frac{h_u^{(k)}}{|N(\nu)|}) + B_k h_\nu^{(k)}, \forall k \in \{0, \ldots, K - 1\} \]

- Non-linearity (e.g., ReLU):
  \[ z_\nu = h_\nu^{(K)} \]

- Embedding after L layers of neighborhood aggregation:
  \[ z_\nu = h_\nu^{(K)} \]

- Notice summation is a permutation invariant pooling/aggregation.
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.

The target node (blue) has the same computation graph for different order plans.
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.

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

$h^{(0)}_v = x_v$

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

$z_v = h^{(K)}_v$

Final node embedding

$h^{k}_v$: 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

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

10/7/21
Many aggregations can be performed efficiently by (sparse) matrix operations

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

Then: $\sum_{u \in N_v} h^{(k)}_u = A_{v,:}H^{(k)}$

Let $D$ be diagonal matrix where

$D_{v,v} = \text{Deg}(v) = |N(v)|$

The inverse of $D$: $D^{-1}$ is also diagonal:

$D^{-1}_{v,v} = 1/|N(v)|$

Therefore,

$$H^{(k+1)} = D^{-1}AH^{(k)}$$
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
Node embedding $z_\nu$ is a function of input graph

**Supervised setting:** we want to minimize the loss $\mathcal{L}$ (see also Slide 15):

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

- $y$: node label
- $\mathcal{L}$ could be L2 if $y$ is a 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

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

- Where \( y_{u,v} = 1 \) when node \( u \) and \( v \) are similar
- \text{CE} is the cross entropy (Slide 16)
- \text{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
Directly train the model for a supervised task (e.g., node classification)

E.g., a drug-drug interaction network

Safe or toxic drug?
Directly train the model for a supervised task (e.g., node classification)

- Use cross entropy loss (Slide 16)

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

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

CNN formulation: 

$$h^{(l+1)}_{v} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^{u} h^{(l)}_{u}), \forall l \in \{0, ..., L - 1\}$$

$N(v)$ represents the 8 neighbor pixels of $v$. 

Image  

CNN weights  

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

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

- CNN formulation:
  \[
  h_{v}^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \forall l \in \{0, ..., L - 1\}
  \]
  \[
  h_{v}^{(l+1)} = \sigma(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_{v}^{(l)}), \forall l \in \{0, ..., L - 1\}
  \]
Convolutional neural network (CNN) layer with 3x3 filter:

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

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

**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), \ldots, (1,1)\}
GNN vs. CNN

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.

**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: ${(-1,-1), (-1,0), (-1, 1),..., (1, 1)}$. 
**GNN vs. CNN**

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

- CNN can be seen as a special GNN with fixed neighbor size and ordering.
- CNN is not permutation equivariant.
  - Switching the order of pixels will lead to different outputs.

**Key difference:** We can learn different $W^u_i$ 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: $\{(-1,-1), (-1,0), (-1,1), \ldots, (1,1)\}$
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.

I  am  a  Stanford  student
GNN vs. Transformer

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

- 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 and Transformer can be viewed as a special GNN