Contents

Overview

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

- 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

Stepup: vertex set, adjacency matrix, node features

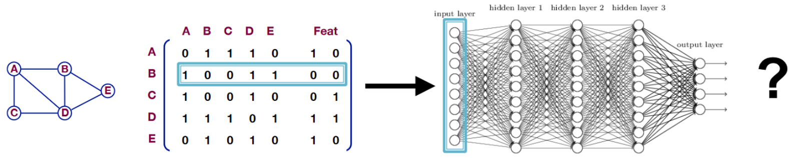
- **Assume we have a graph $\boldsymbol{G}$:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - 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, …, 1]

Naive Approach: append node feaetures to adjacency matrix

3 problems: parameter size, graph size, node ordering

1. Parameter size
   a. One training example per node but for each node there are N + X (node features) number of features
   b. Training unstable / easy to overfit
2. Graph with different size
   - E.g. If 5 nodes, hard to fit in input size of 7
3. Node ordering
   - If the column order change, then the adjacency matrix changes
     - Rows & cols permuted thou the info is the same
   - For images, the ordering can be top left pixel to bottom right
   - but for graphs, there is no fix node ordering i.e. unclear how to sort the graph to put them as input in the matrix
   - Has to be **invariant** to **node ordering**

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



	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0

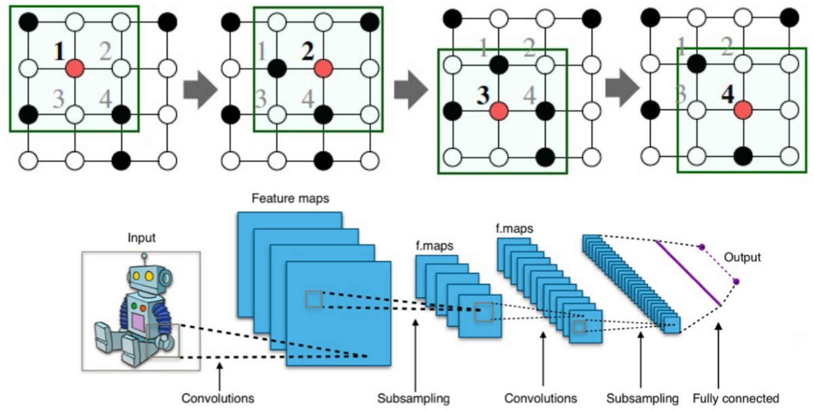
- Issues with this idea:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

Adopt CNN

Goal. generalize convolutions beyond simple lattices & Leverage node features/attributes (e.g., text, images)

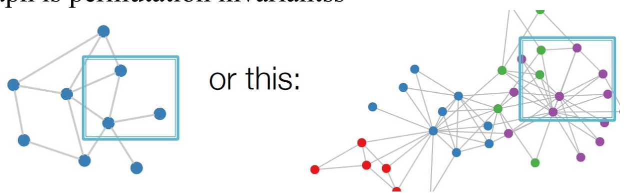
CNN = Sliding windows & locality.

CNN on an image:



Problem

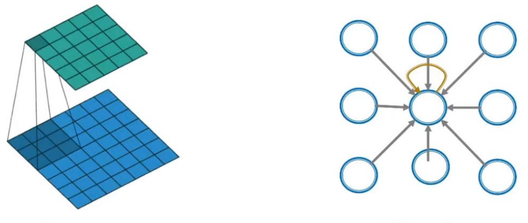
- no fixed notion of locality or sliding window on the graph
  - L: covers 3 nodes
  - R: covers more nodes
- Graph is permutation invariantss



Solutions

- Aggregate information about a node based on its neighbouring nodes

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



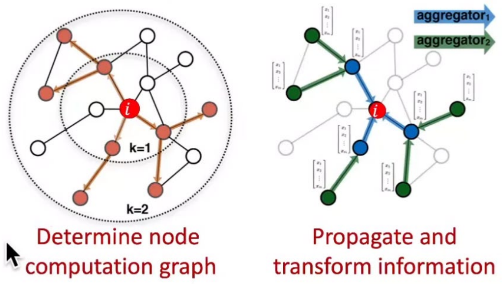
Idea: transform information at the neighbors and combine it:

- Transform "messages" h_i from neighbors: $W_i\ h_i$
- Add them up: $\sum_i W_i\ h_i$

Kipf & Welling, ICLR 2017

- Neighbour nodes takes the message from the node and propagate. Steps
  1. Determine node computation graph
  2. Propagate & transform information

Idea: Node's neighborhood defines a computation graph

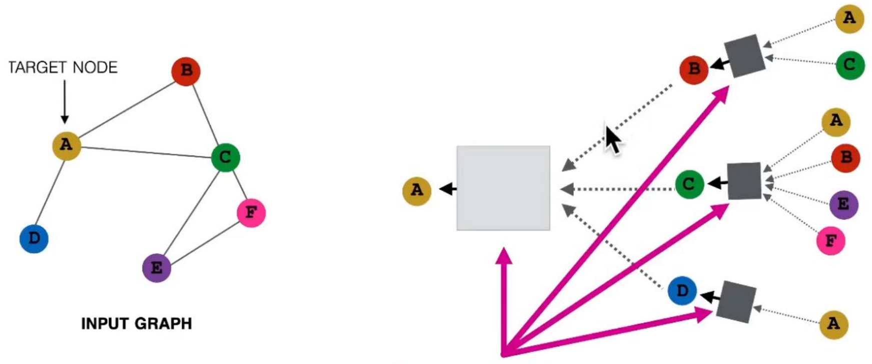

k=1
k=2
aggregator
aggregator
Determine node computation graph
Propagate and transform information

Learn how to propagate information across the graph to compute node features

Explain – transformation & aggregation

1. To decide node A informatino, we collect from its neighbours {B, C, D}, in which their info are based on their neighbours.
2. The message passing is then from the leaf to root
   a. First, transform the info from leaf
   b. Second, aggregate them in the parent node
   c. Repeat

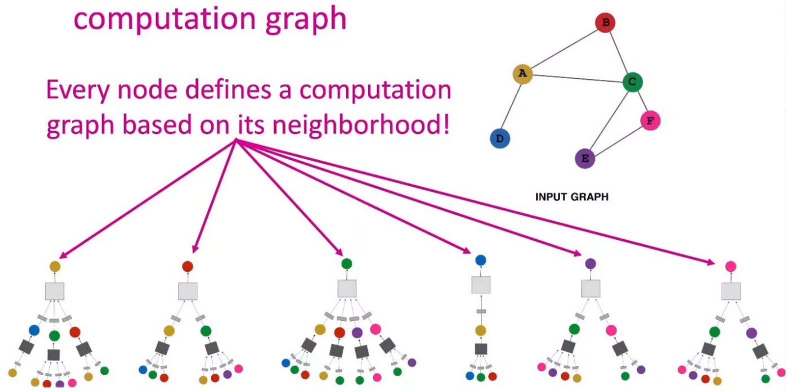
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



Explain – each node has a computational graph

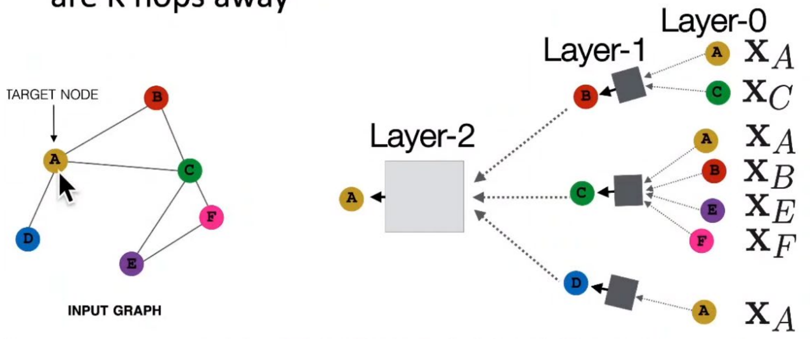
- Every node has its own computation graph / architecture
- The structure depends on other structure
- Different to classical DL

- **Intuition:** Network neighborhood defines a computation graph


Every node defines a computation graph based on its neighborhood!
INPUT GRAPH

Many layers – k-hops

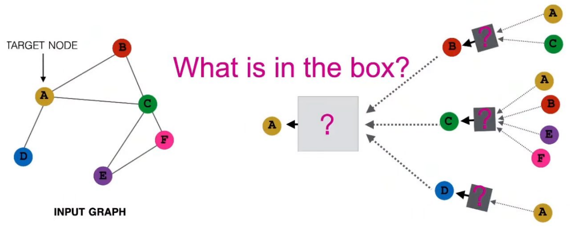
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node u is its input feature, x_u
  - Layer-k embedding gets information from nodes that are K hops away



Neighbourhood aggregation

- The aggregation result is the same regardless of the ordering

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



Transformation choice – average embedding

- Ordering invariant
- Examples
  - Average messages
  - Apply NN
    - Apply linear transformation
    - Follow by non-linearity

- Transform current layer features + aggregated previous child nodes messages

Initial 0-th layer embeddings are equal to node features

$$h_v^0 = x_v$$

embedding of v at layer k

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

Total number of layers

$$z_v = h_v^{(K)}$$

Embedding after L layers of neighborhood aggregation

Non-linearity (e.g., ReLU)

Average of neighbor's previous layer embeddings

Notice summation is a permutation invariant pooling/aggregation.

Model Parameters: W, B; neighborhood aggregation, transformation

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

$$h_v^{(0)} = x_v$$

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

$$z_v = h_v^{(K)}$$

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

Matrix Formulation - efficient sparse matrix

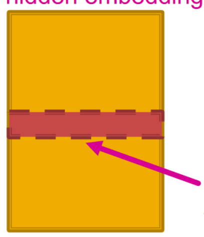
Averging of neighbour embedding

- Many aggregations can be performed **efficiently** by (sparse) matrix operations
- Then
  - Node embedding is the avage of the neighbour embedding
  - i.e. the adjacency matrix * embedding spaces at a layer
- in short
  - averging of neighbour embedding
    - summing and averagign can be rewritten as
    - **matrix multiplication (dot prodct)**
  - the drawing is what correspond to what

- Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^{\mathrm{T}}$
- Then: $\sum_{u \in N_v} h_u^{(k)} = A_{v,:} H^{(k)}$
- Let D be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
  - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
- **Therefore,**

$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \Rightarrow H^{(k+1)} = D^{-1} A H^{(k)}$$

Matrix of hidden embeddings $H^{(k-1)}$



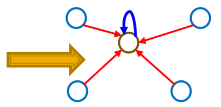
Re-writing update function in matrix form

- 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**

- Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A} H^{(k)} W_k^{\mathrm{T}} + H^{(k)} B_k^{\mathrm{T}})$$

where $\tilde{A} = D^{-1} A$



$H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$

- Red: neighborhood aggregation
- Blue: self transformation

How to train GNN: supervised & unsupervised setting

- 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!**

Unsupervised Training

- No labels

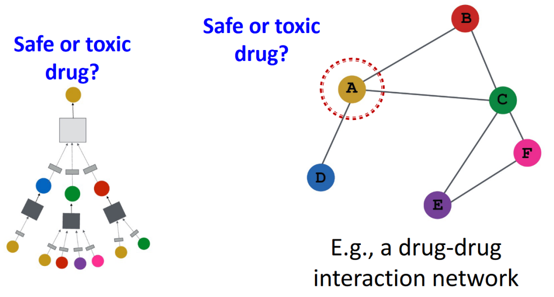
- “Similar” nodes have similar embeddings

$$\mathcal{L} = \sum_{z_u,z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

  - 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)



Cross entropy loss

- If label is 1, want output to be 1
- If 0, want 0

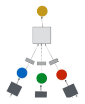
- Use cross entropy loss (Slide 16)

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

Encoder output: node embedding

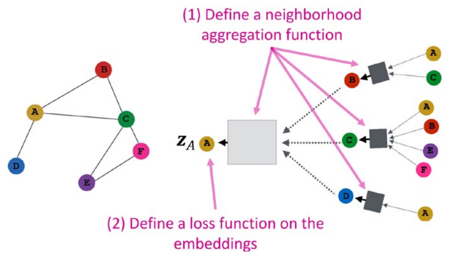
Classification weights

Node class label

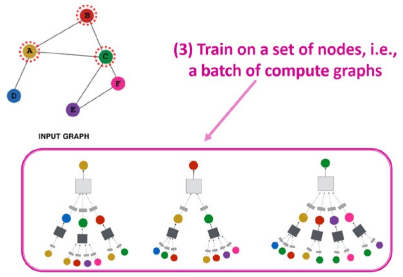


Model design

Step 1,2 - neighbour aggregation & loss function

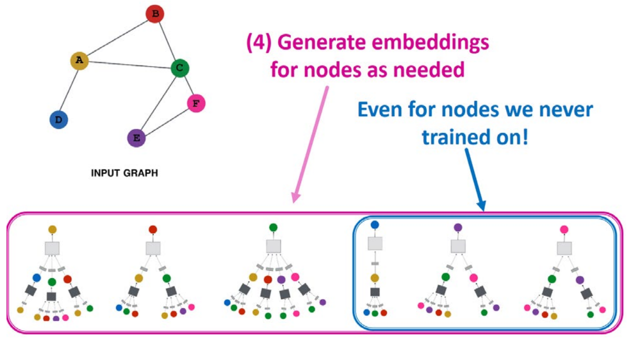

(1) Define a neighborhood aggregation function
z_A
A
B
C
D
E
F
(2) Define a loss function on the embeddings

Step 3 – train on a batch


INPUT GRAPH
(3) Train on a set of nodes, i.e., a batch of compute graphs

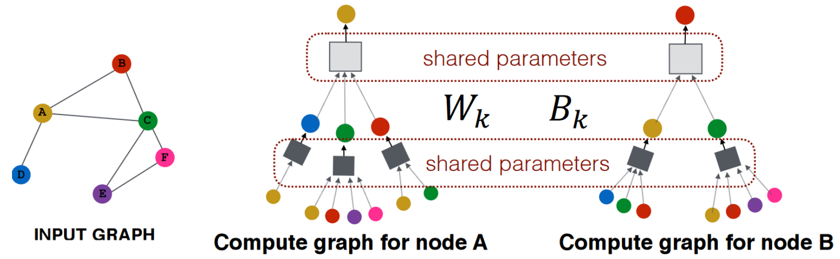
Step 4 - enerate embeddings for nodes as needed

- **Generalisability.** Train embedding for one graph and transfer to another graph

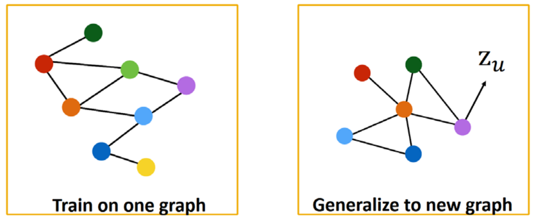

INPUT GRAPH
(4) Generate embeddings for nodes as needed
Even for nodes we never trained on!

Inductive capcability

- The same aggregation parameters are **shared** for all nodes
- The number of model parameters is **sublinear** in $|V|$
- # model parameters **W & B**, depends on
  - #features / embedding dimensationality (since shared)
  - not the size of graph (#nodes)
- Thus, generalize to **unseen** nodes



New graph

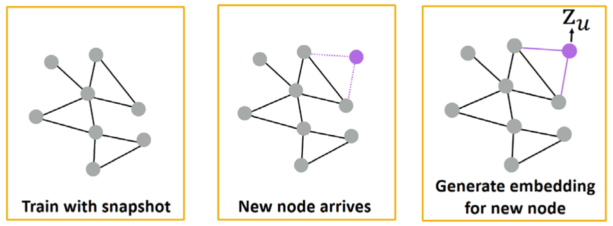


Inductive node embedding → Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

New nodes

- One forward pass can generate new embedding for the new node as the graph evoloving

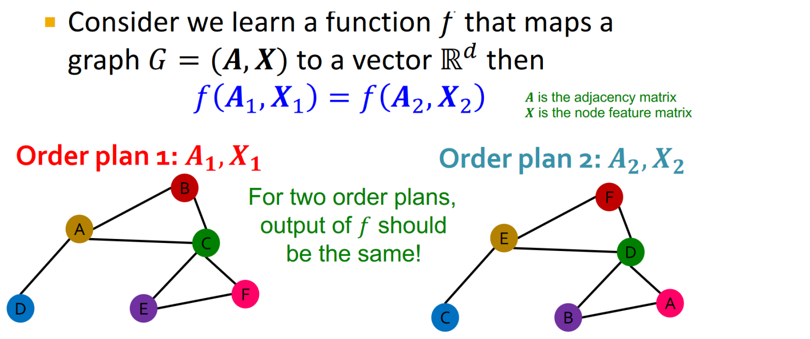


- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings “on the fly”

More

Permutation Invariance

- Graph does not have a canonical order of the nodes
- For a graph with m nodes, there are $m!$ different order plans.
- Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?
  - No
  - Switching the order of the input leads to different outputs!
  - This explains why the **naïve MLP** approach **fails** for **graphs**



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

$f(\boldsymbol{A}_1, \boldsymbol{X}_1) = f(\boldsymbol{A}_2, \boldsymbol{X}_2)$ — $\boldsymbol{A}$ is the adjacency matrix; $\boldsymbol{X}$ is the node feature matrix

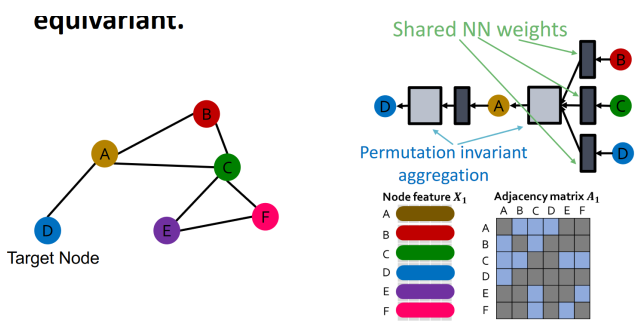
Order plan 1: $\boldsymbol{A}_1, \boldsymbol{X}_1$

For two order plans, output of f should be the same!

Order plan 2: $\boldsymbol{A}_2, \boldsymbol{X}_2$

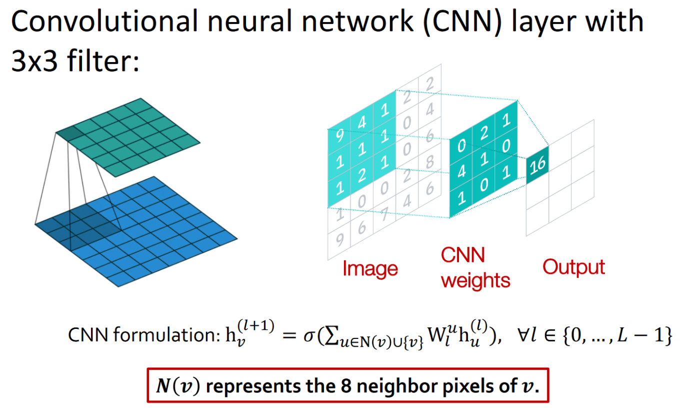
Equivariant Property

- 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**



equivariant.

GNNs subsume CNNs and Transformers



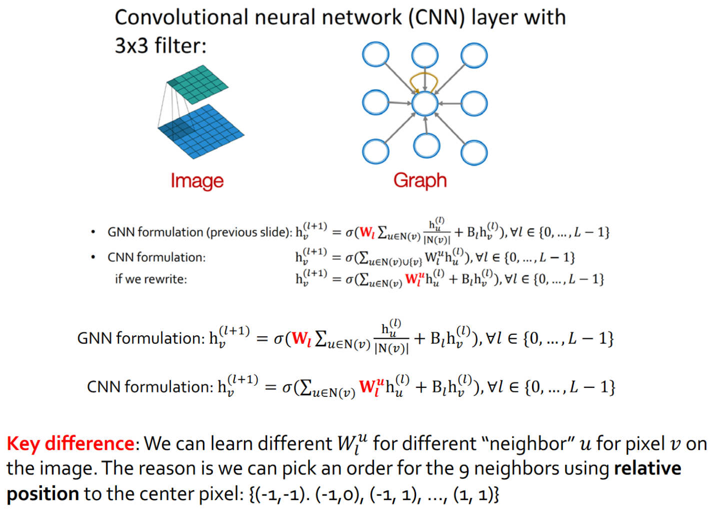
Convolutional neural network (CNN) layer with 3x3 filter:

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

$\boldsymbol{N(v)}$ **represents the 8 neighbor pixels of** $\boldsymbol{v}$**.**

GNN vs. CNN

- 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 equivariant
    - Switching the order of pixels will leads to different outputs.



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

- GNN formulation (previous slide): $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \dots, L-1\}$
- CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \forall l \in \{0, \dots, L-1\}$
  if we rewrite: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

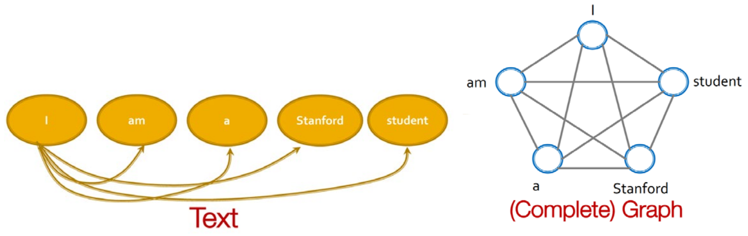
GNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

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)}

Transformer

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



Takeaway

- What if the ordering plan does matter?