Graph Neural Networks: Model III

Jiaxuan You

Assistant Professor at UIUC CDS

CS598: Deep Learning with Graphs, 2024 Fall

https://ulab-uiuc.github.io/CS598/

Recap: 5 Questions for Discussion

- **What is the problem?** (12:50 – 13:05)
  - Brainstorm at least 3 ideas
- **Why is it interesting and important?** (13:20 – 13:30)
- **Why is it hard?** (E.g., why do naive approaches fail?) (13:30 – 13:35)
- **Why hasn't it been solved before?** (Or, what's wrong with previous proposed solutions? How does mine differ?) (13:35 – 13:45)

- What are the key components of my approach and results? Also include any specific limitations. (Optional)

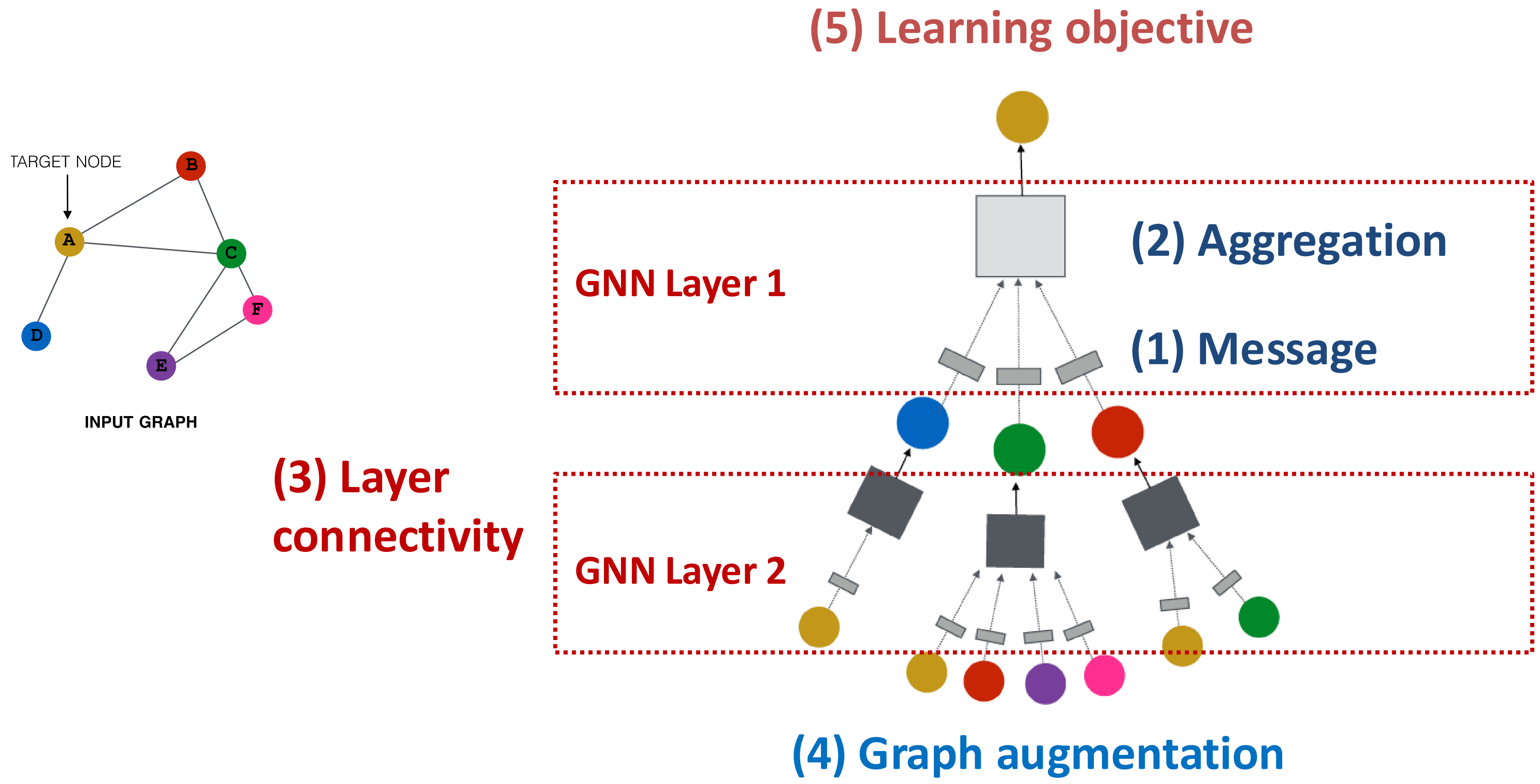
Reminder: HW and Proposal Task

- Coding Assignment 1
  - Submit your code and written answers downloaded from Colab to Canvas
  - **Submission Deadline: Sept 29 (Sun) 11:59 PM, CT**
- Proposal Task
  - Submit a PDF file to Canvas for each group. Groups have been created on Canvas. Contact the TA if you need to update.
  - Required to use the ICLR 2025 template. Contact the TA if you have problems with LaTeX.
  - Submissions are suggested to be between 1.5 to 2 pages in length, with a minimum of 1 page, excluding references.
  - **Submission Deadline: Oct 6 (Sun) 11:59 PM, CT**

Graph Neural Networks: Model II

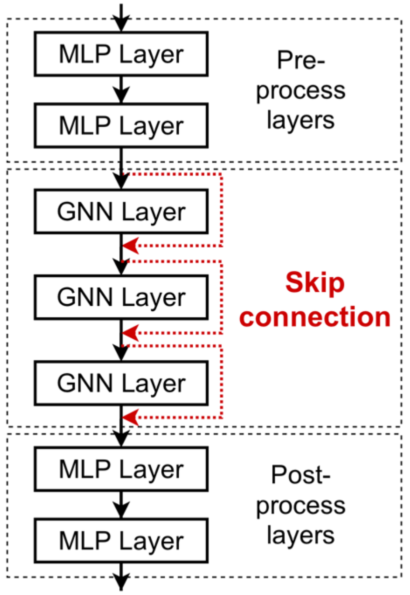
GNN Augmentation and Training

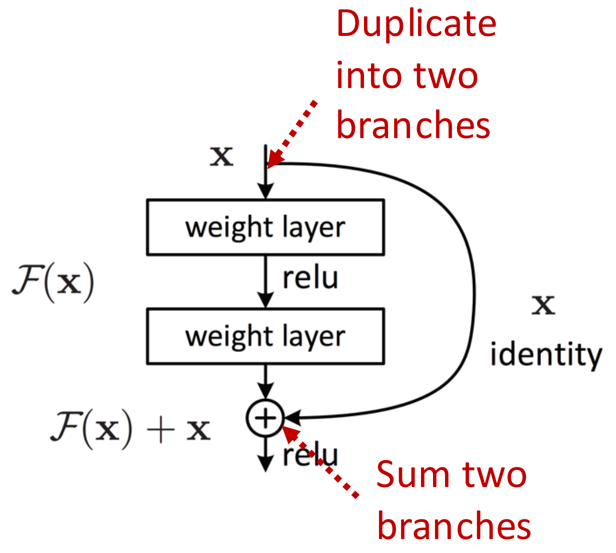
Recap: A General GNN Framework



Recap: GNN Layer Connectivity

- **What if my problem still requires many GNN layers?**
- **Lesson 2: Add skip connections in GNNs**
  - **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
  - **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**




Duplicate into two branches
$\mathbf{x}$
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
Sum two branches

Idea of skip connections:

Before adding shortcuts:

$$\boldsymbol{F}(\mathbf{x})$$

After adding shortcuts:

$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

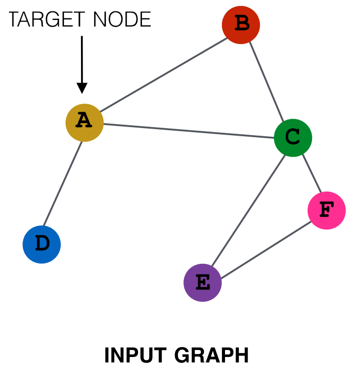
Recap: Graph Manipulation

- **Graph Feature manipulation**
  - The input graph **lacks features** → **feature augmentation**
- **Graph Structure manipulation**
  - The graph is **too sparse** → **Add virtual nodes / edges**
  - The graph is **too dense** → **Sample neighbors when doing message passing**
  - The graph is **too large** → **Sample subgraphs to compute embeddings**
    - Will cover later in lecture: Scaling up GNNs

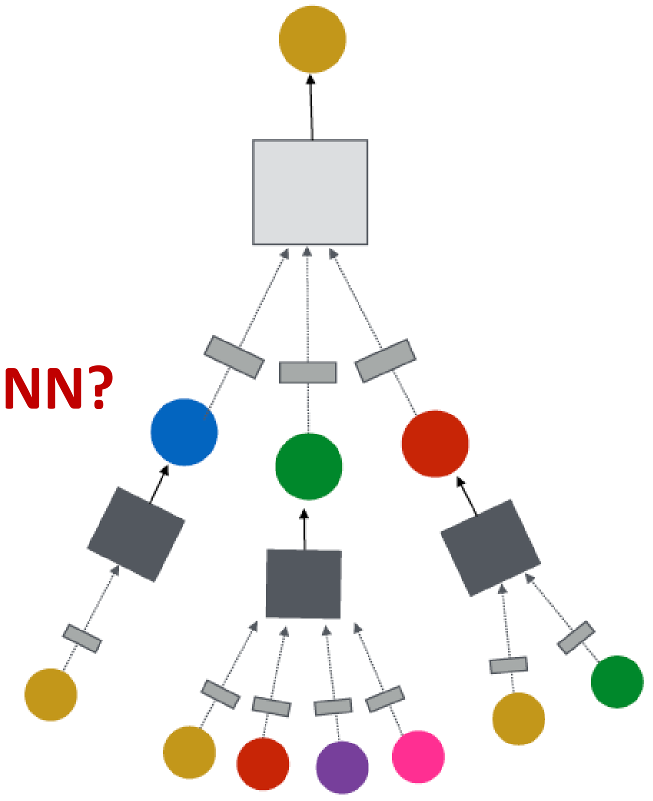
Graph Neural Networks: Model II

Prediction with GNNs

A General GNN Framework (4)

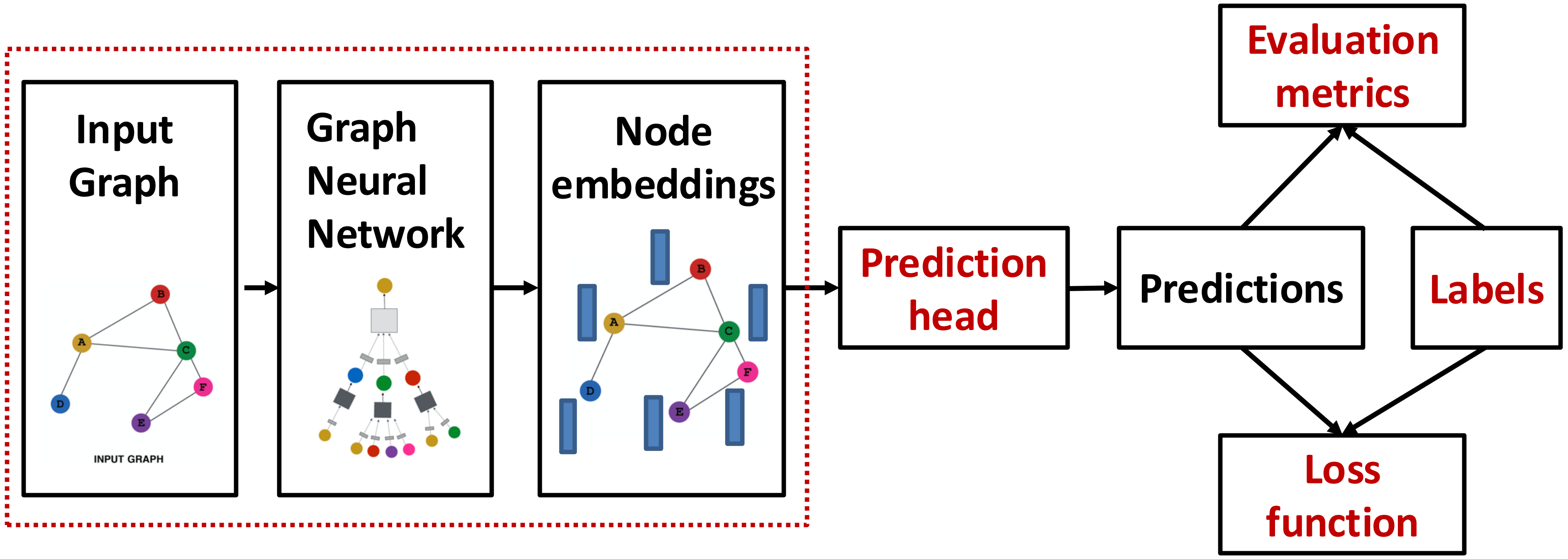


(5) Learning objective



Next: How do we train a GNN?

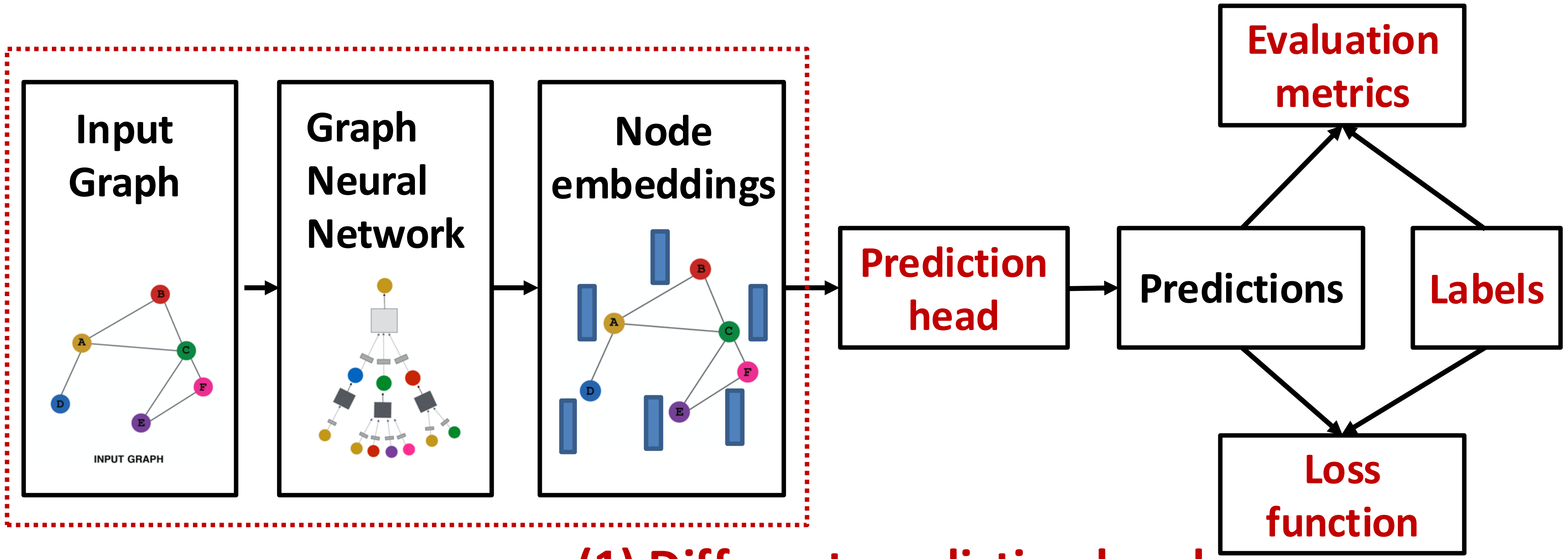
GNN Training Pipeline



Output of a GNN: set of node embeddings

$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

GNN Training Pipeline (1)

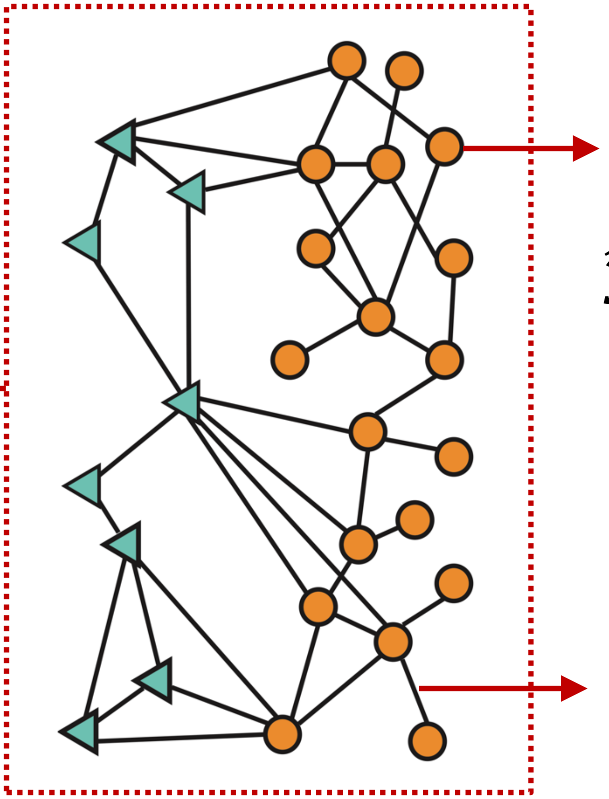


(1) Different prediction heads:

- **Node-level tasks**
- **Edge-level tasks**
- **Graph-level tasks**

Recap: Graph Learning Prediction Heads

- **Idea:** Different task levels require different prediction heads
- **Prediction head:** map **node embeddings** to the **predictions of interest**



Node-level prediction

$$\hat{\boldsymbol{y}}_v = \mathrm{Head}_{\mathrm{node}}(\mathbf{h}_v) = \mathbf{W} \quad \mathbf{h}_v$$

$$\hat{\boldsymbol{y}}_G = \mathrm{Mean}(\{\mathbf{h}_v \in \mathbb{R}^d, \forall v \in G\})$$

Graph-level prediction

$$\hat{\boldsymbol{y}}_v = \mathrm{Head}_{\mathrm{node}}(\mathbf{h}_v) = \mathbf{W} \quad \mathbf{h}_v$$

Edge-level prediction

Recap: Prediction Heads: Graph-level

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- **(1) Global mean pooling**

$$\hat{\boldsymbol{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

$$\hat{\boldsymbol{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\boldsymbol{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs
- **Can we do better for large graphs?**

Issue of Global Pooling

- **Issue:** Global pooling over a (large) graph will lose information
- **Toy example:** we use 1-dim node embeddings
  - Node embeddings for G_1: $\{-1, -2, 0, 1, 2\}$
  - Node embeddings for G_2: $\{-10, -20, 0, 10, 20\}$
  - Clearly G_1 and G_2 have very different node embeddings $\rightarrow$ Their structures should be different
- **If we do global sum pooling:**
  - **Prediction for** G_1: $\hat{y}_G = \mathrm{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
  - **Prediction for** G_2: $\hat{y}_G = \mathrm{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
  - We cannot differentiate G_1 and G_2!

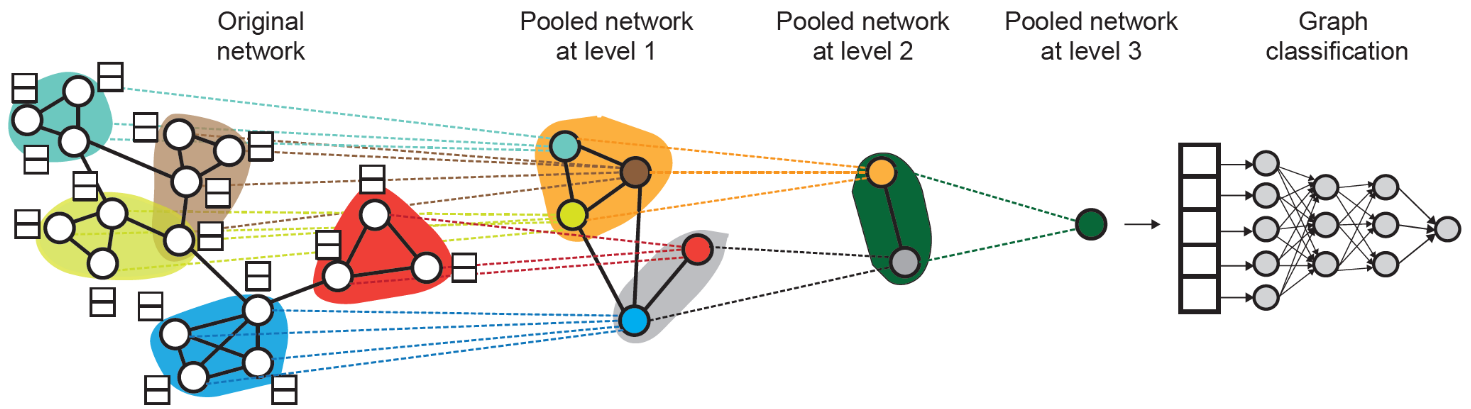
Hierarchical Global Pooling

- **A solution:** Let's aggregate all the node embeddings **hierarchically**
  - **Toy example:** We will aggregate via $\mathrm{ReLU}\left(\mathrm{Sum}(\cdot)\right)$
    - We first separately aggregate the first 2 nodes and last 3 nodes
    - Then we aggregate again to make the final prediction
  - G_1 node embeddings: $\{-1, -2, 0, 1, 2\}$
    - **Round 1**: $\hat{y}_a = \mathrm{ReLU}\left(\mathrm{Sum}(\{-1, -2\})\right) = 0$, $\hat{y}_b = \mathrm{ReLU}\left(\mathrm{Sum}(\{0,1, 2\})\right) = 3$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}\left(\mathrm{Sum}(\{y_a, y_b\})\right) = \mathbf{3}$
  - G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
    - **Round 1**: $\hat{y}_a = \mathrm{ReLU}\left(\mathrm{Sum}(\{-10, -20\})\right) = 0$, $\hat{y}_b = \mathrm{ReLU}\left(\mathrm{Sum}(\{0,10, 20\})\right) = 30$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}\left(\mathrm{Sum}(\{y_a, y_b\})\right) = \mathbf{30}$

Now we can differentiate G_1 and G_2 !

Hierarchical Pooling In Practice

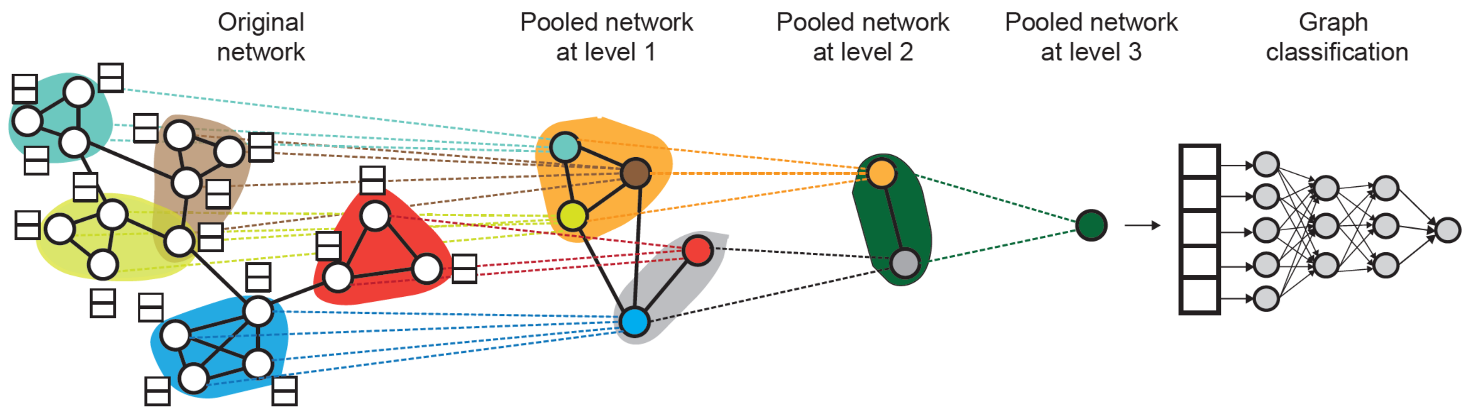
- **DiffPool idea:**
  - **Hierarchically pool node embeddings**


Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3
Graph classification

  - **Leverage 2 independent GNNs at each level**
    - **GNN A:** Compute node embeddings
    - **GNN B:** Compute the cluster that a node belongs to
  - **GNNs A and B at each level can be executed in parallel**

Hierarchical Pooling In Practice

- **DiffPool idea:**


Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3
Graph classification

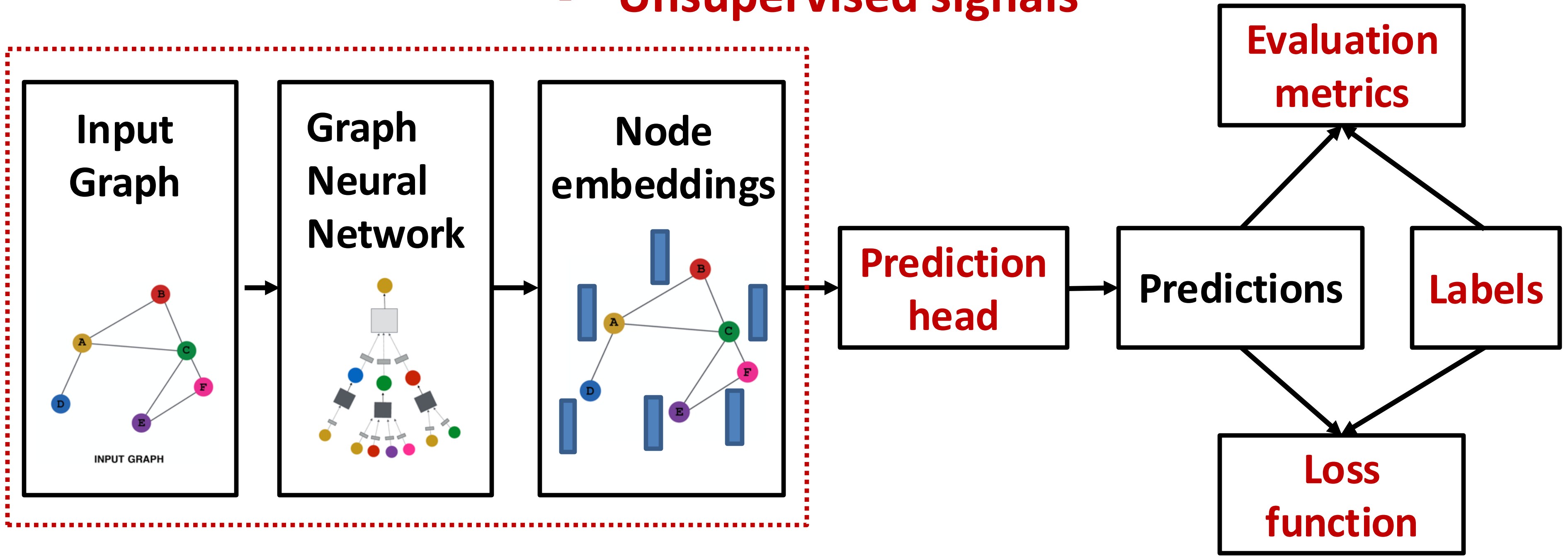
  - **For each Pooling layer**
    - Use clustering assignments from **GNN B** to aggregate node embeddings generated by **GNN A**
    - Create a **single new node** for each cluster, maintaining edges between clusters to generated a new **pooled** network
  - **Jointly train GNN A and GNN B**

Graph Neural Networks: Model II

Training Graph Neural Networks

GNN Training Pipeline (2)

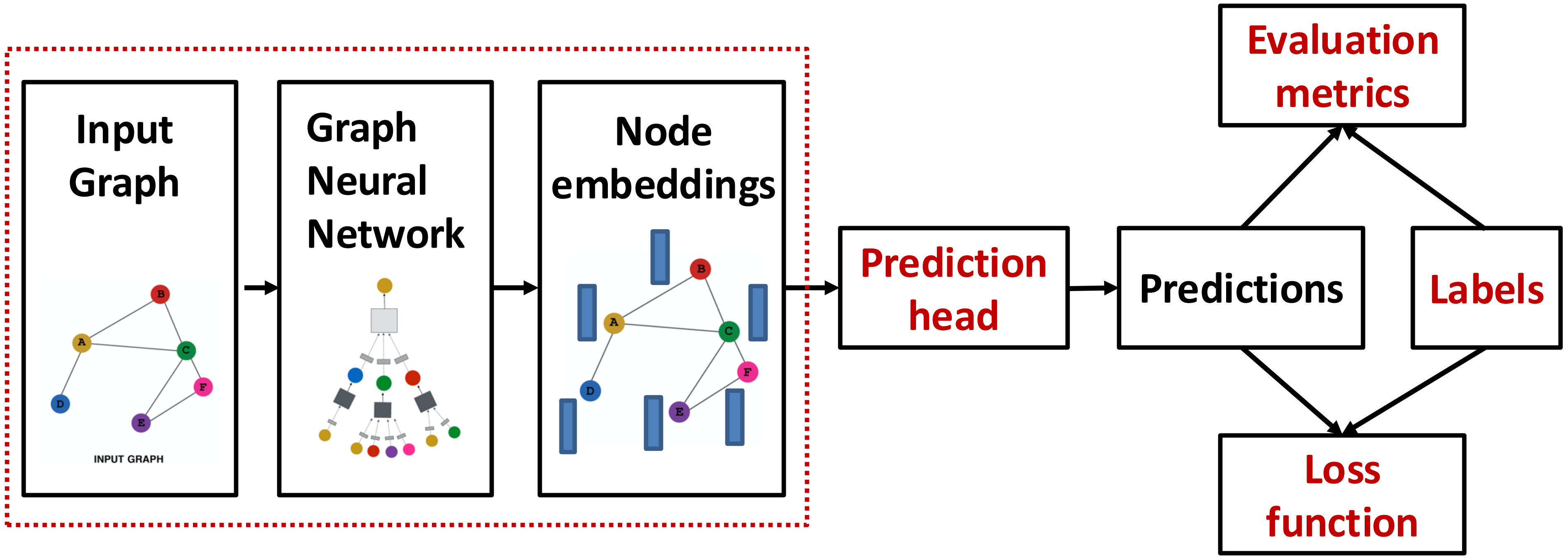
(2) Where does ground-truth come from?
- **Supervised labels**
- **Unsupervised signals**



Recap: Supervised vs Unsupervised

- **Supervised learning on graphs**
  - **Labels come from external sources**
    - E.g., predict drug likeness of a molecular graph
- **Unsupervised learning on graphs**
  - **Signals come from graphs themselves**
    - E.g., link prediction: predict if two nodes are connected
- **Sometimes the differences are blurry**
  - We still have “supervision” in unsupervised learning
    - E.g., train a GNN to predict node clustering coefficient
  - An alternative name for “**unsupervised**” is “**self-supervised**”

GNN Training Pipeline (3)



(3) How do we compute the final loss?

- **Classification loss**
- **Regression loss**

Settings for GNN Training

- **The setting:** We have N data points
  - Each data point can be a node/edge/graph
  - **Node-level**: prediction $\widehat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\widehat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\widehat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\widehat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

Classification or Regression

- **Classification**: labels $\boldsymbol{y}^{(i)}$ with discrete value
  - E.g., Node classification: which category does a node belong to
- **Regression**: labels $\boldsymbol{y}^{(i)}$ with continuous value
  - E.g., predict the drug likeness of a molecular graph
- GNNs can be applied to both settings
- **Differences: loss function & evaluation metrics**

Classification Loss

- As discussed in lecture 6, **cross entropy (CE)** is a very common loss function in classification
- K-way prediction for i-th data point:

$$\mathrm{CE}\Big(\underset{\textbf{Label}}{\boxed{\boldsymbol{y}^{(i)}}}, \underset{\textbf{Prediction}}{\boxed{\widehat{\boldsymbol{y}}^{(i)}}}\Big) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log\big(\widehat{\boldsymbol{y}}_{\boxed{j}}^{\boxed{(i)}}\big)$$

***i*-th data point**

***j*-th class**

where:

E.g.

0	0	1	0	0

$$\boldsymbol{y}^{(i)} \in \mathbb{R}^K = \text{one-hot label encoding}$$

$$\widehat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K = \text{prediction after } \mathrm{Softmax}(\cdot)$$

E.g.

0.1	0.3	0.4	0.1	0.1

- Total loss over all N training examples

$$\mathrm{Loss} = \sum_{i=1}^{N} \mathrm{CE}\big(\boldsymbol{y}^{(i)}, \widehat{\boldsymbol{y}}^{(i)}\big)$$

Regression Loss

- For regression tasks we often use **Mean Squared Error (MSE)** a.k.a. **L2 loss**
- K-way regression for data point (i):

$$\mathrm{MSE}\left(\boldsymbol{y}^{(i)}, \widehat{\boldsymbol{y}}^{(i)}\right) = \sum_{j=1}^{K} \left(\boldsymbol{y}_j^{(i)} - \widehat{\boldsymbol{y}}_{\boxed{j}}^{\boxed{(i)}}\right)^2$$

***i*-th data point**

***j*-th target**

where:

E.g.

1.4	2.3	1.0	0.5	0.6

$\boldsymbol{y}^{(\boldsymbol{i})} \in \mathbb{R}^k$ = Real valued vector of targets

$\widehat{\boldsymbol{y}}^{(\boldsymbol{i})} \in \mathbb{R}^k$ = Real valued vector of predictions

E.g.

0.9	2.8	2.0	0.3	0.8

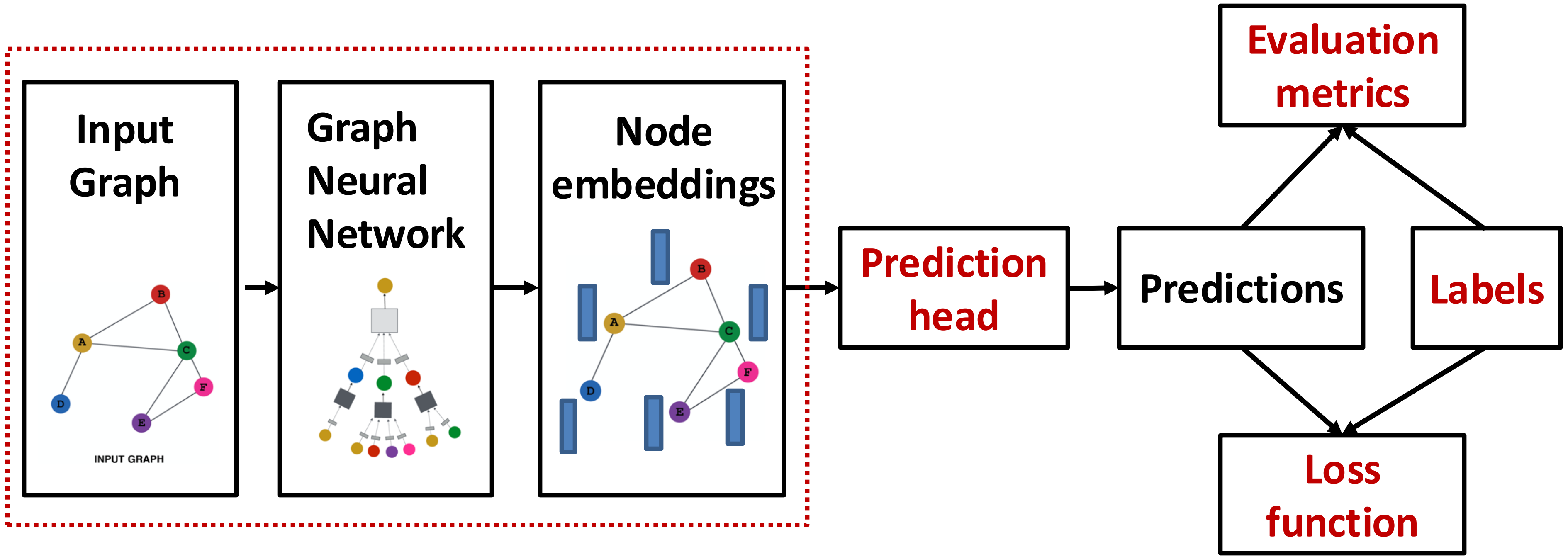
- Total loss over all N training examples

$$\mathrm{Loss} = \sum_{i=1}^{N} \mathrm{MSE}\left(\boldsymbol{y}^{(i)}, \widehat{\boldsymbol{y}}^{(i)}\right)$$

GNN Training Pipeline (4)

(4) How do we measure the success of a GNN?

- **Accuracy**
- **ROC AUC**



Evaluation Metrics: Regression

- **We use standard evaluation metrics for GNN**
  - (Content below can be found in any ML course)
  - In practice we will use sklearn for implementation
  - Suppose we make predictions for N data points
- **Evaluate regression tasks on graphs:**
  - **Root mean square error (RMSE)**

$$\sqrt{\sum_{i=1}^{N} \frac{(\boldsymbol{y}^{(i)} - \widehat{\boldsymbol{y}}^{(i)})^2}{N}}$$

  - **Mean absolute error (MAE)**

$$\frac{\sum_{i=1}^{N} \left|\boldsymbol{y}^{(i)} - \widehat{\boldsymbol{y}}^{(i)}\right|}{N}$$

Evaluation Metrics: Classification

- **Evaluate classification tasks on graphs:**
- **(1) Multi-class classification**
  - **We simply report the accuracy**

$$\frac{1\left[\operatorname{argmax}\left(\widehat{\boldsymbol{y}}^{(i)}\right) = \boldsymbol{y}^{(i)}\right]}{N}$$

- **(2) Binary classification**
  - Metrics sensitive to classification threshold
    - **Accuracy**
    - **Precision / Recall**
    - If the range of prediction is $[0,1]$, we will use 0.5 as threshold
  - Metric Agnostic to classification threshold
    - **ROC AUC**

Metrics for Binary Classification

- **Accuracy:**

$$\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}=\frac{\mathrm{TP}+\mathrm{TN}}{|\mathrm{Dataset}|}$$

- **Precision (P):**

$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$$

- **Recall (R):**

$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$$

- **F1-Score:**

$$\frac{2\mathrm{P}*\mathrm{R}}{\mathrm{P}+\mathrm{R}}$$

Confusion matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

(4) Evaluation Metrics

- **ROC Curve:** Captures the tradeoff in TPR and FPR **as the classification threshold is varied for a binary classifier.**

TPR

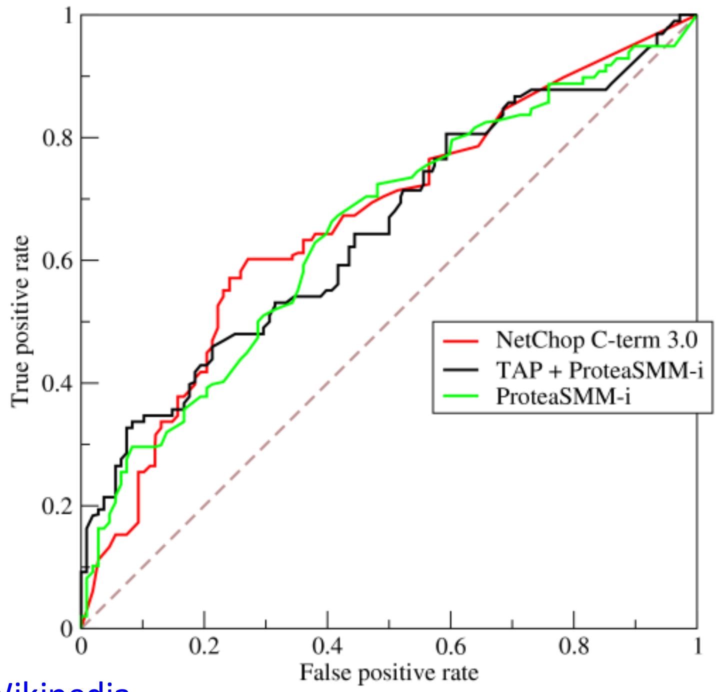


Image Credit: Wikipedia

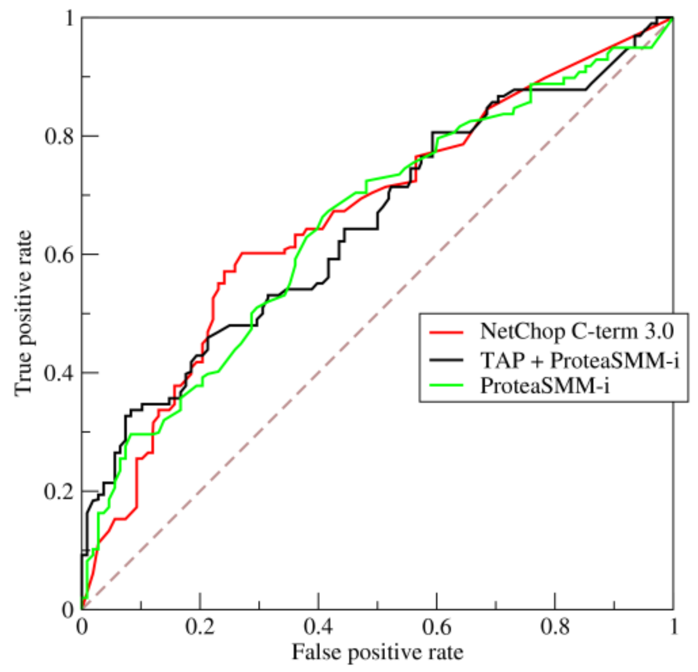
FPR

$$\text{TPR} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Note: the dashed line represents **performance of a random classifier**

(4) Evaluation Metrics



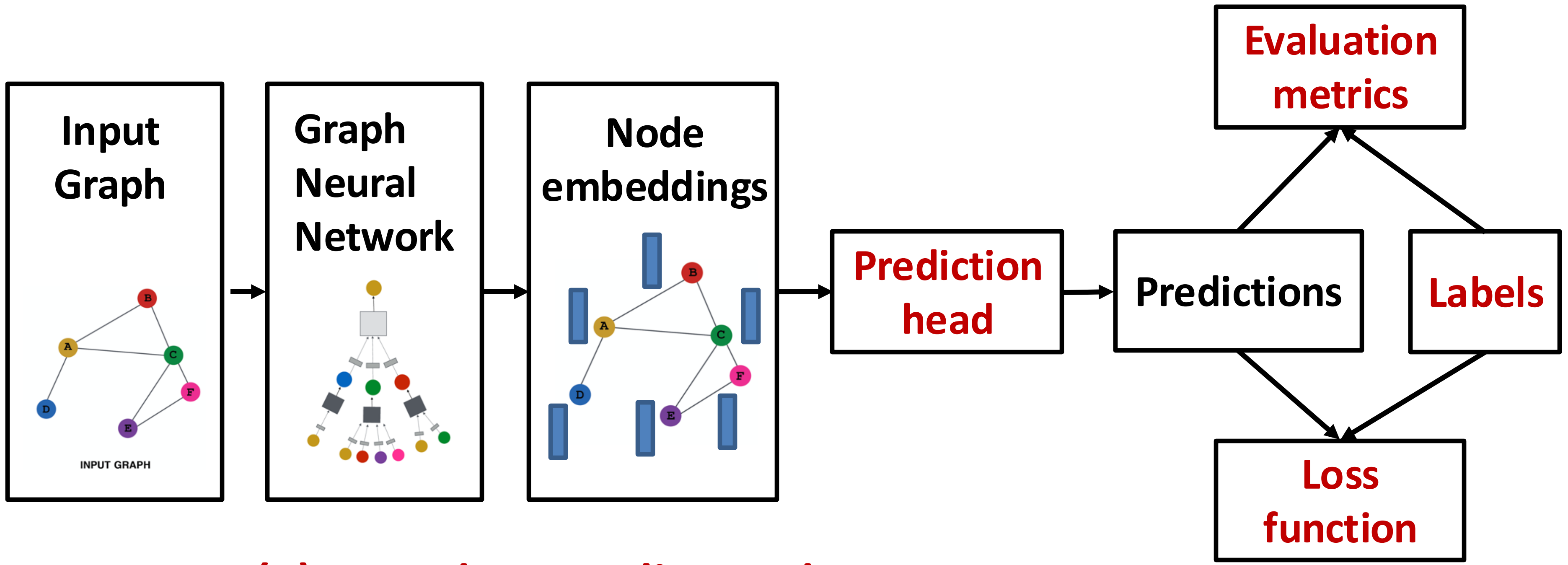
Content Credit: [Wikipedia]

- **ROC AUC: Area under the ROC Curve.**
- **Intuition:** The probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one

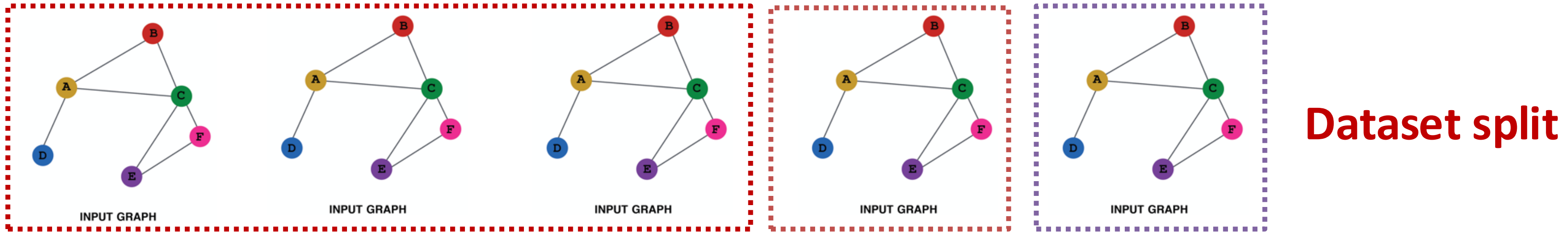
Graph Neural Networks: Model II

Setting-up GNN Prediction Tasks

GNN Training Pipeline (5)



(5) How do we split our dataset into train / validation / test set?

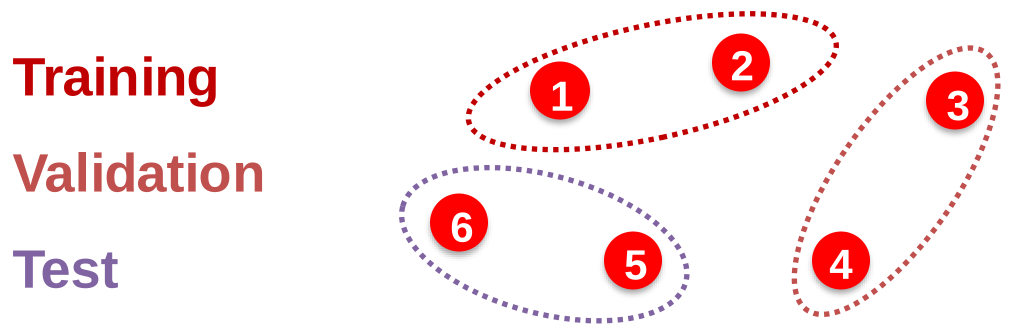


Dataset Split: Fixed / Random Split

- **Fixed split:** We will split our dataset **once**
  - **Training set**: used for optimizing GNN parameters
  - **Validation set**: develop model/hyperparameters
  - **Test set**: held out until we report final performance
- **A concern:** sometimes we cannot guarantee that the test set will really be held out
- **Random split:** we will **randomly split** our dataset into training / validation / test
  - We report **average performance over different random seeds**

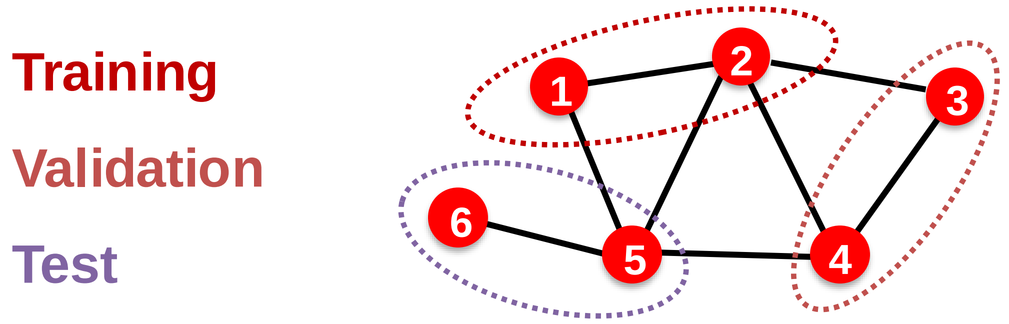
Why Splitting Graphs is Special

- **Suppose we want to split an image dataset**
  - **Image classification: Each data point is an image**
  - Here **data points are independent**
    - **Image 5 will not affect our prediction on image 1**



Why Splitting Graphs is Special

- **Splitting a graph dataset is different!**
  - **Node classification: Each data point is a node**
  - Here **data points are NOT independent**
    - **Node 5 will affect our prediction on node 1,** because it will participate in message passing → affect node 1's embedding



- **What are our options?**

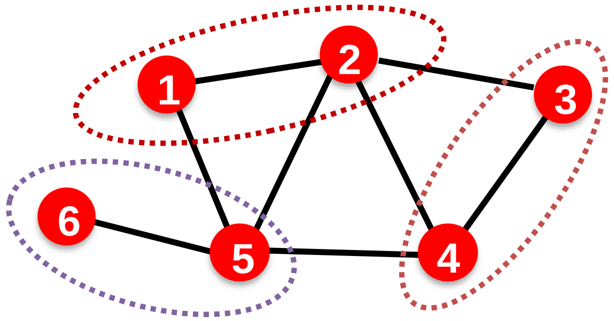
Why Splitting Graphs is Special

- **Solution 1 (Transductive setting): The input graph can be observed in all the dataset splits (training, validation and test set).**
- **We will only split the (node) labels**
  - **At training time**, we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**

Training

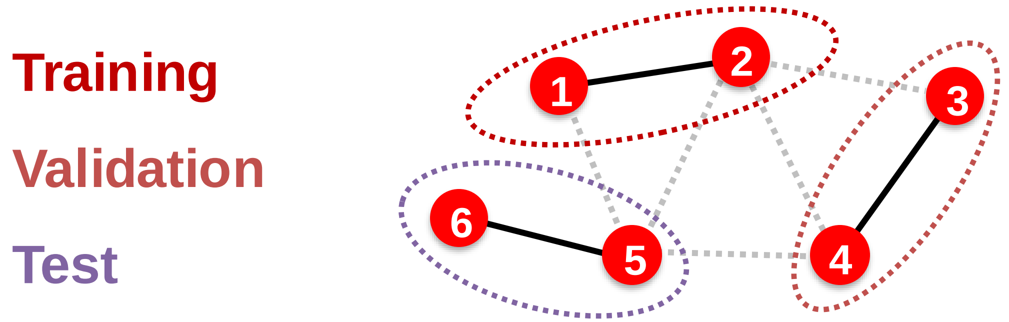
Validation

Test



Why Splitting Graphs is Special

- **Solution 2 (Inductive setting): We break the edges between splits to get multiple graphs**
  - **Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 any more**
  - **At training time**, we compute embeddings **using the graph over node 1&2**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the graph over node 1&2&3&4**, and **evaluate on node 3&4's labels**

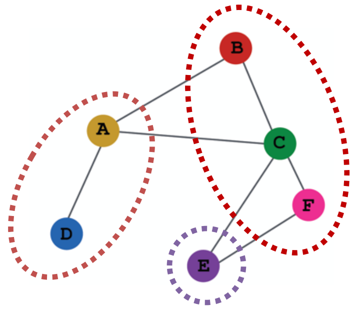


Transductive / Inductive Settings

- **Transductive setting:** training / validation / test sets are **on the same graph**
  - The **dataset consists of one graph**
  - **The entire graph can be observed in all dataset splits, we only split the labels**
  - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / validation / test sets are **on different graphs**
  - The **dataset consists of multiple graphs**
  - Each split can **only observe the graph(s) within the split**. A successful model should **generalize to unseen graphs**
  - Applicable to **node / edge / graph** tasks

Example: Node Classification

- **Transductive** node classification
  - **All the splits can observe the entire graph structure**, but can only observe the labels of their respective nodes

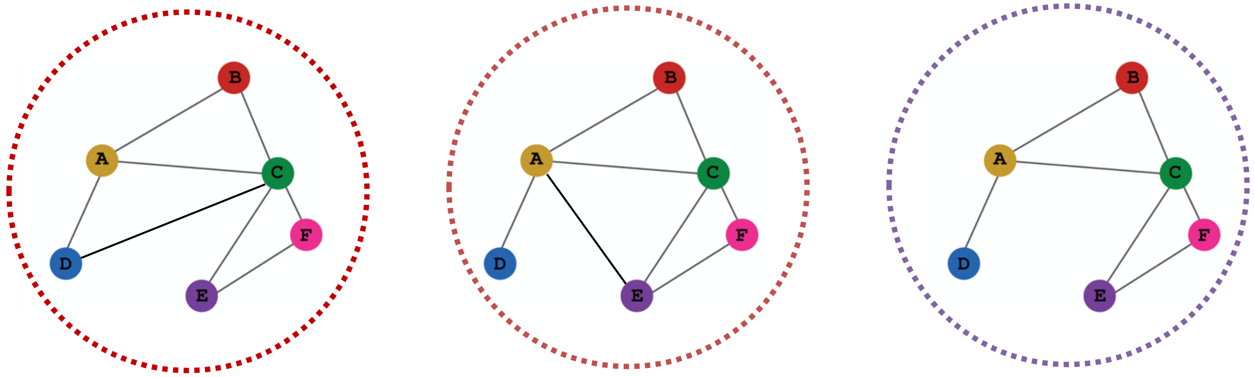


Training

Validation

Test

- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**



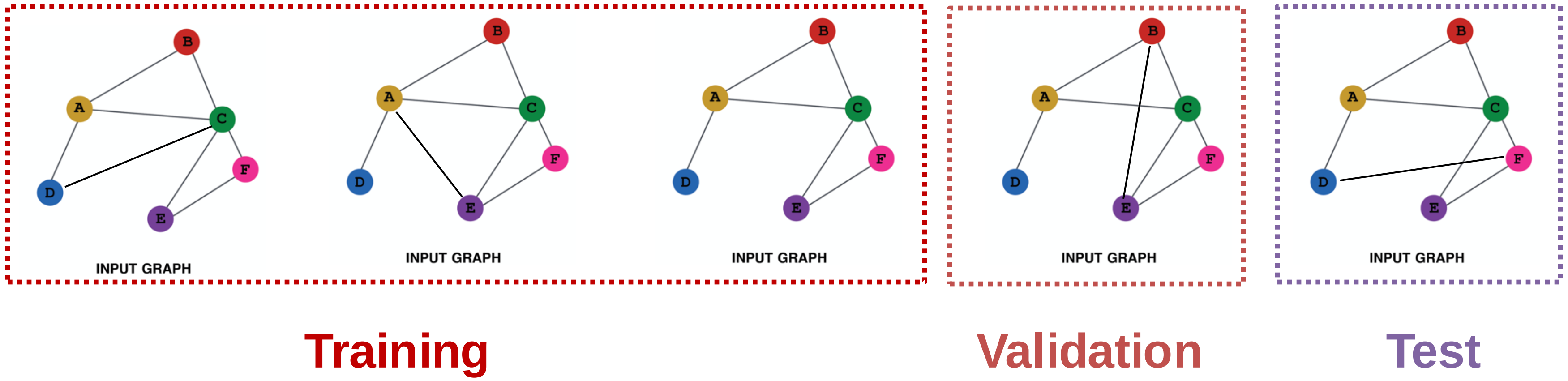
Training

Validation

Test

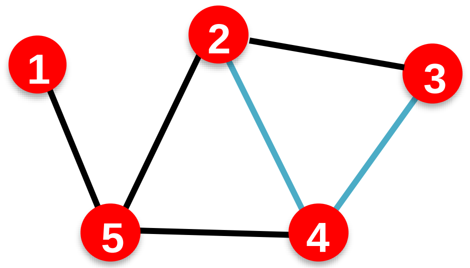
Example: Graph Classification

- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).

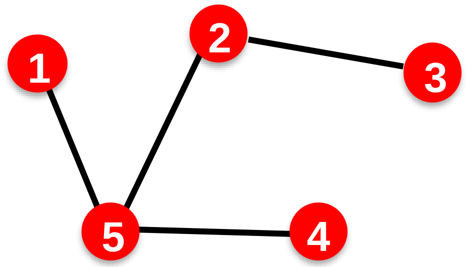


Example: Link Prediction

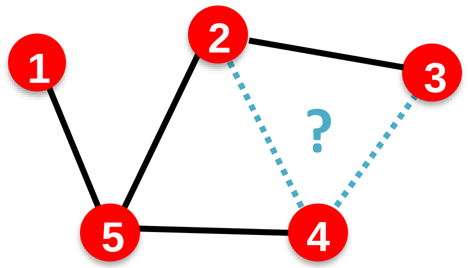
- **Goal of link prediction**: **predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let the GNN predict if the edges exist**
  - **Important for Knowledge graph completion & reasoning as well!**



Original graph

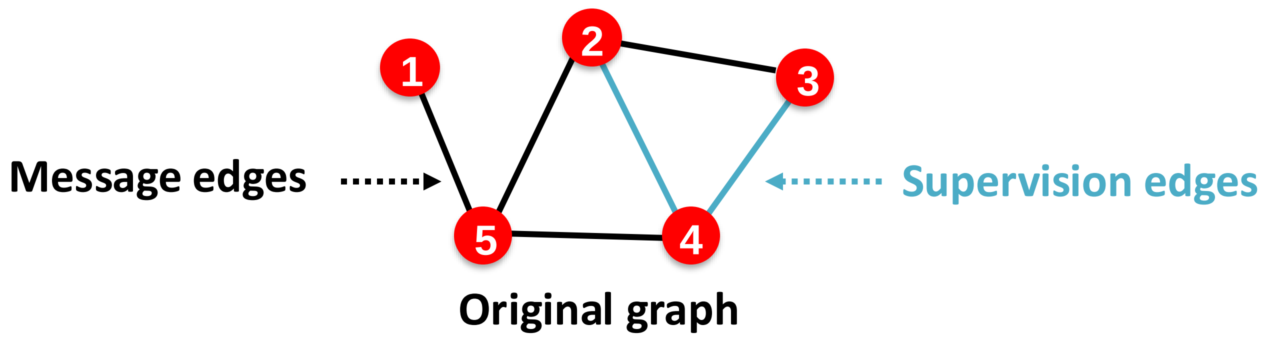


Input graph to GNN



Predictions made by GNN

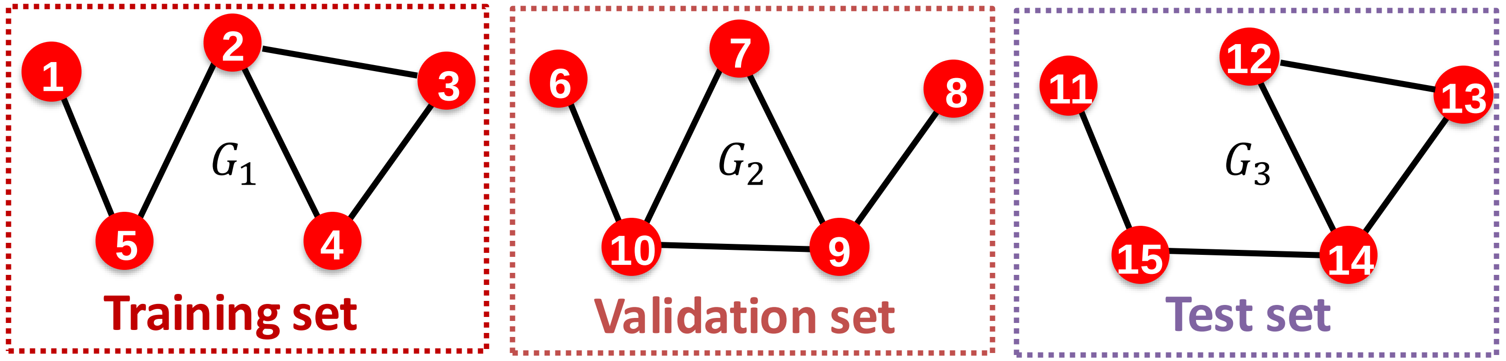
Setting up Link Prediction



- **For link prediction, we will split edges twice**
- **Step 1: Assign 2 types of edges in the original graph**
  - **Message edges: Used for GNN message passing**
  - **Supervision edges: Use for computing objectives**
  - **After step 1:**
    - **Only message edges will remain in the graph**
    - **Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN!**

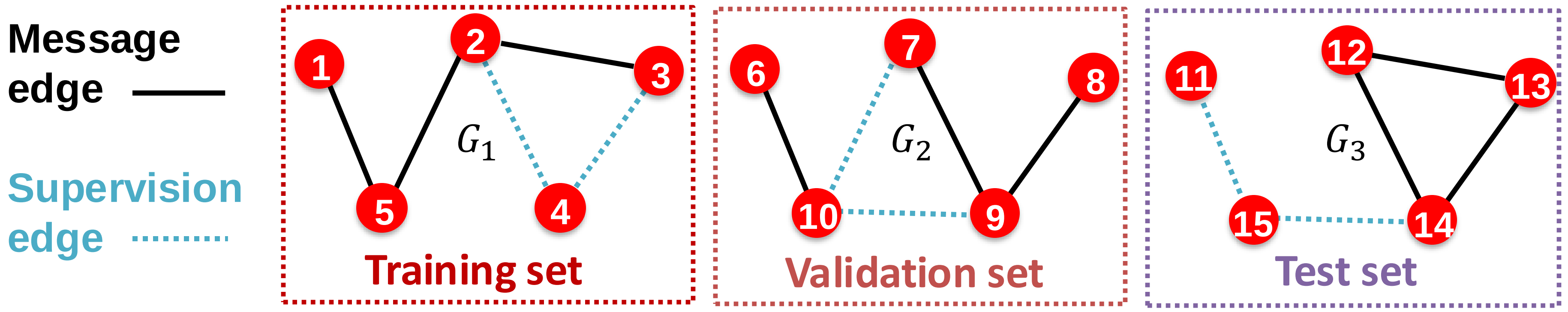
Setting up Link Prediction

- **Step 2: Split edges into train / validation / test**
- **Option 1: Inductive link prediction split**
  - **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**



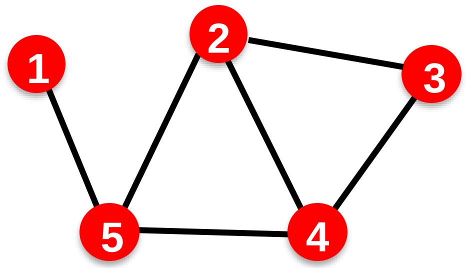
Setting up Link Prediction

- **Step 2: Split edges into train / validation / test**
- **Option 1: Inductive link prediction split**
  - **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**
  - **In train** or **val** or **test** set, **each graph will have 2 types of edges**: **message edges** + **supervision edges**
    - **Supervision edges** are not the input to GNN



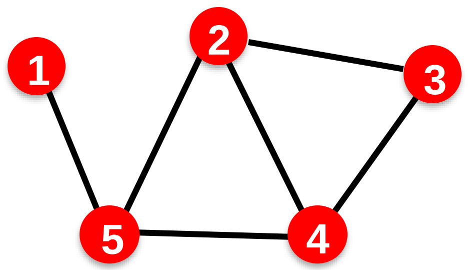
Setting up Link Prediction

- **Option 2: Transductive link prediction split:**
  - **This is the default setting when people talk about link prediction**
  - **Suppose we have a dataset of 1 graph**



Setting up Link Prediction

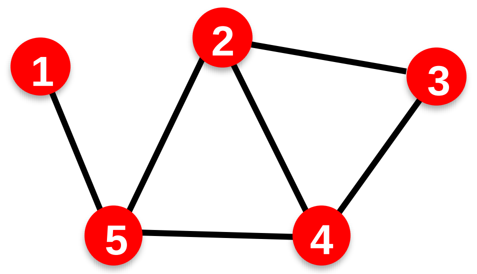
- **Option 2: Transductive link prediction split:**
  - **By definition of "transductive", the entire graph can be observed in all dataset splits**
    - **But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges**
    - **To train the training set, we further need to hold out supervision edges for the training set**



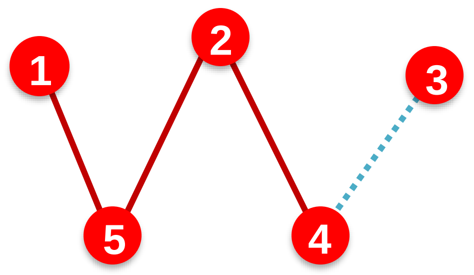
  - **Next:** we will show the exact settings

Setting up Link Prediction

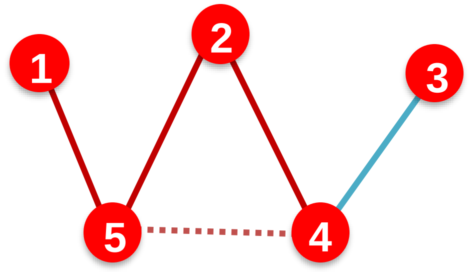
- **Option 2: Transductive link prediction split:**



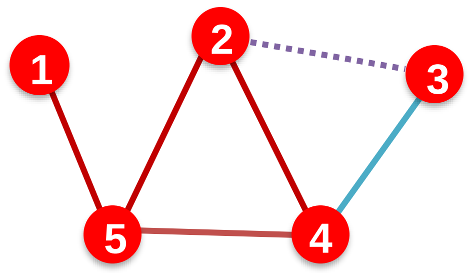
The original graph



(1) At training time:
Use **training message edges** to predict **training supervision edges**



(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**

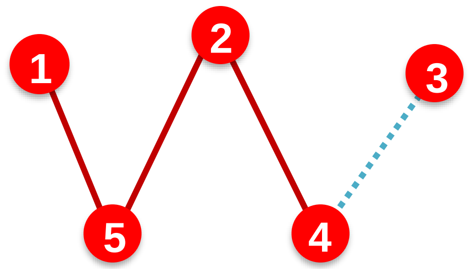


(3) At test time:
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

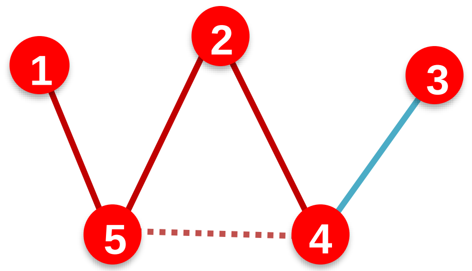
Setting up Link Prediction

- **Option 2: Transductive link prediction split:**

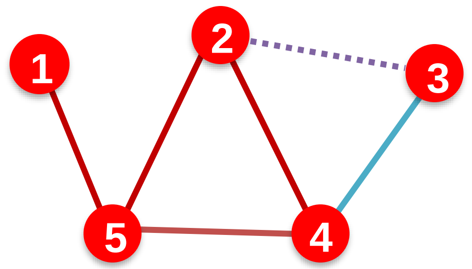
Why do we use growing number of edges?
After training, supervision edges are known to GNN.
Therefore, an ideal model should use supervision edges in message passing at validation time.
The same applies to the test time.



(1) At training time:
Use **training message edges** to predict **training supervision edges**



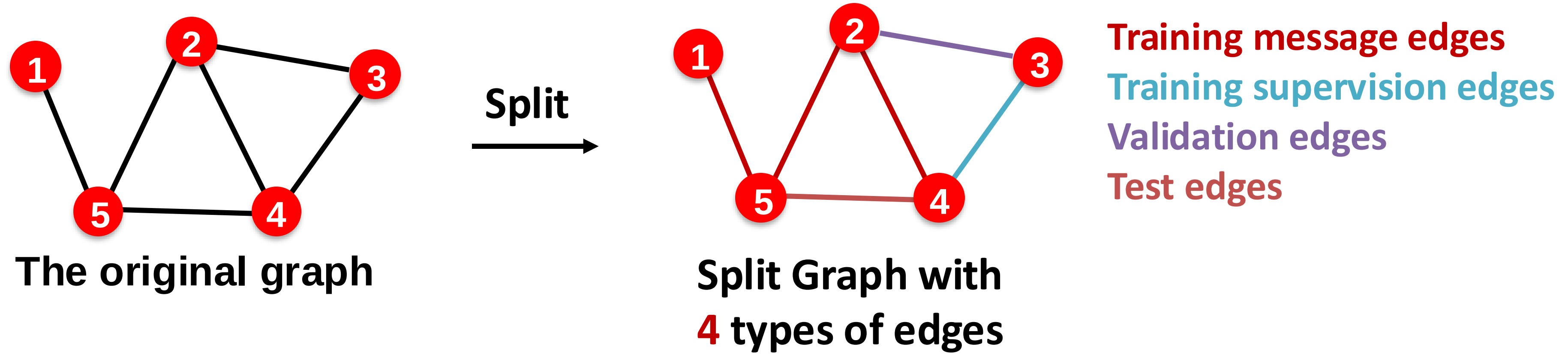
(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**



(3) At test time:
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

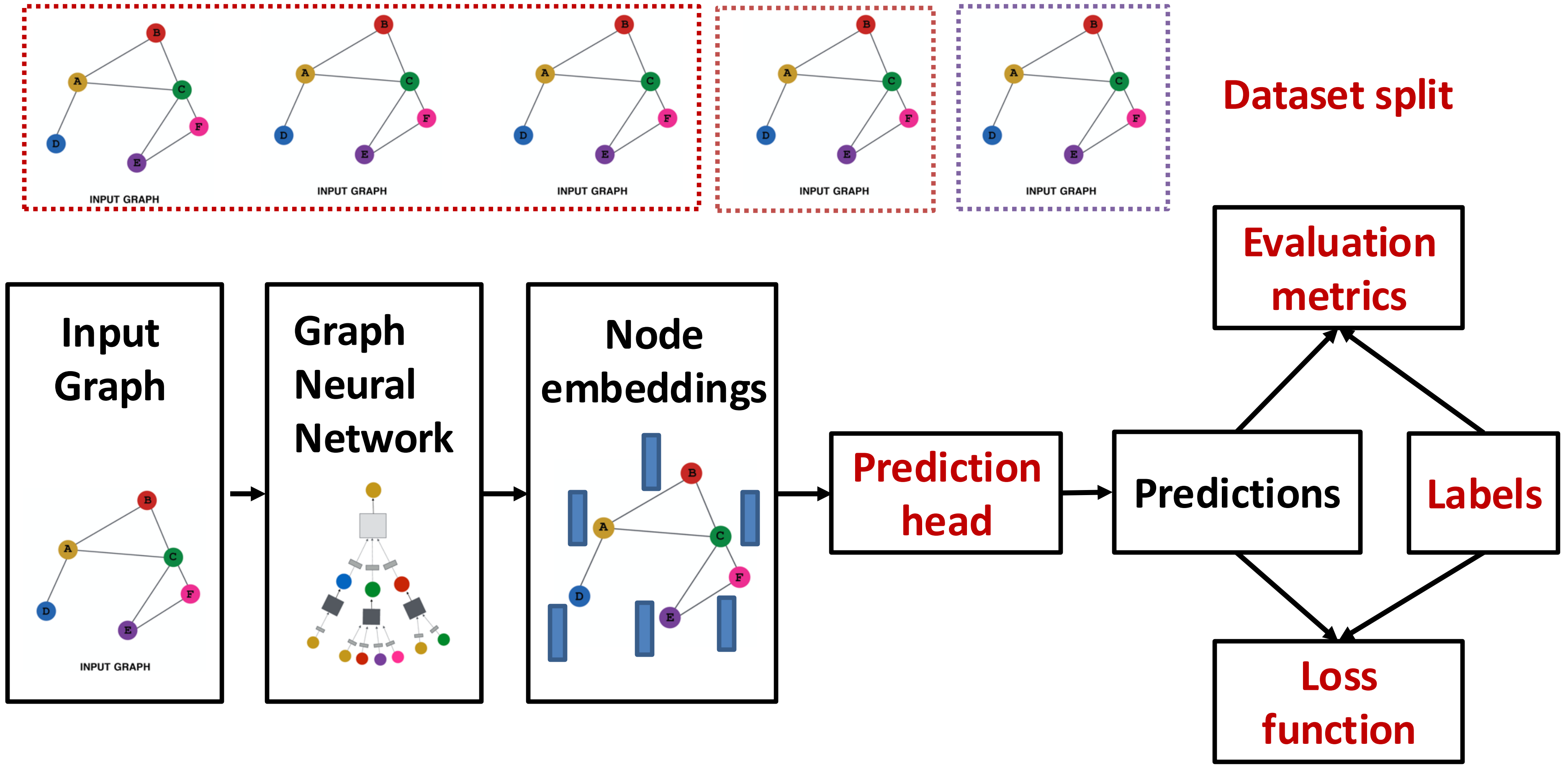
Setting up Link Prediction

- **Summary: Transductive link prediction split:**



- **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently.
- Luckily, we have full support in **PyG and GraphGym**

GNN Training Pipeline



Implementation resources:

[DeepSNAP]() provides core modules for this pipeline

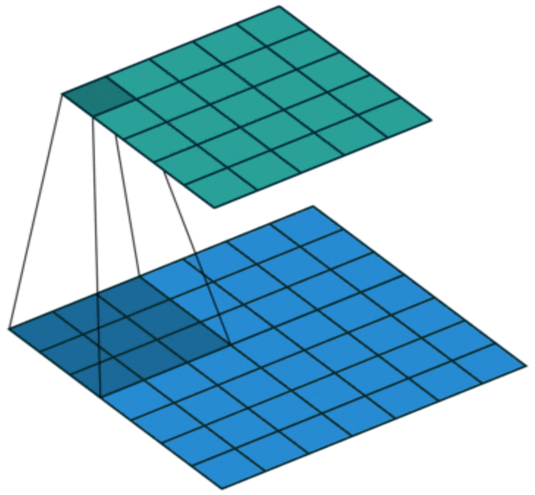
[GraphGym]() further implements the full pipeline to facilitate GNN design

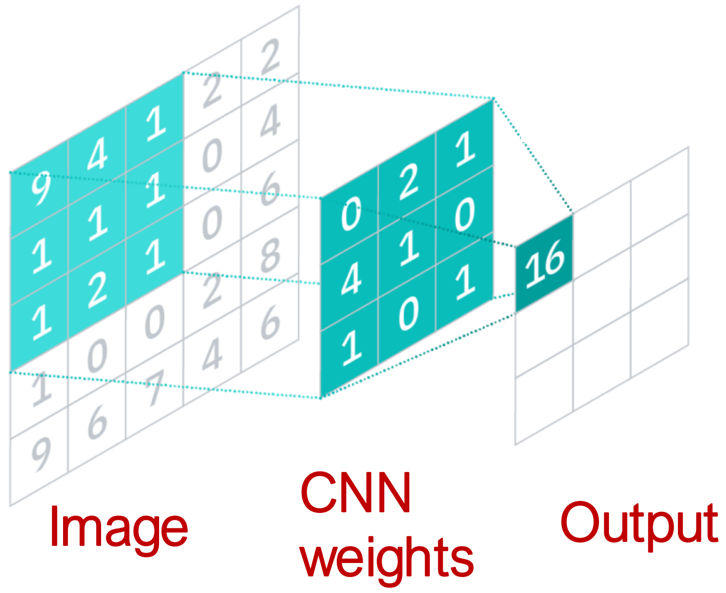
Graph Neural Networks: Perspective

GNNs vs CNNs and Transformers

Convolutional Neural Network

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



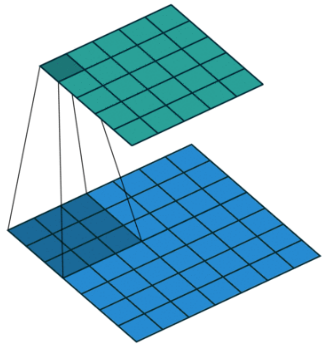


CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$

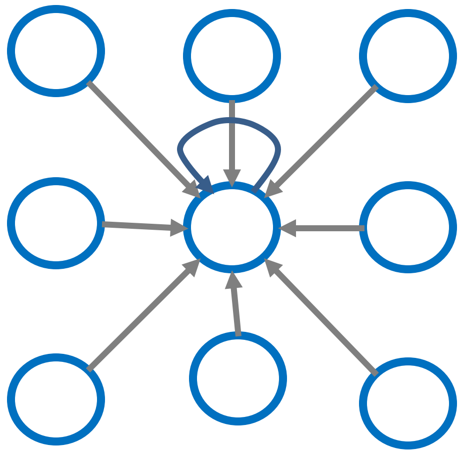
$N(v)$ represents the 8 neighbor pixels of v.

GNN vs. CNN

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



Image



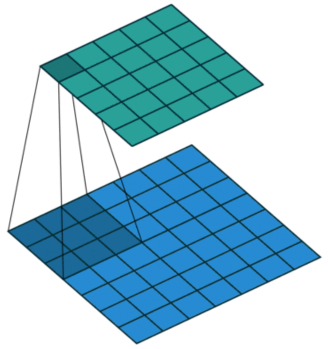
Graph

- GNN formulation: $h_v^{(l+1)} = \sigma(\mathbf{W_l} \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$
- CNN formulation: (previous slide) $h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \forall l \in \{0, \dots, L-1\}$

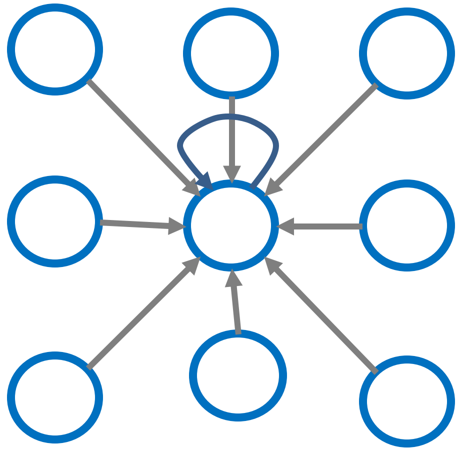
  if we rewrite: $h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W_l^u} h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1$

GNN vs. CNN

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



Image



Graph

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 define a canonical order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)} (in other words, image is an ordered data structure)

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3

CNN can be seen as a GNN with fixed neighbor size and non permutation invariant message function:

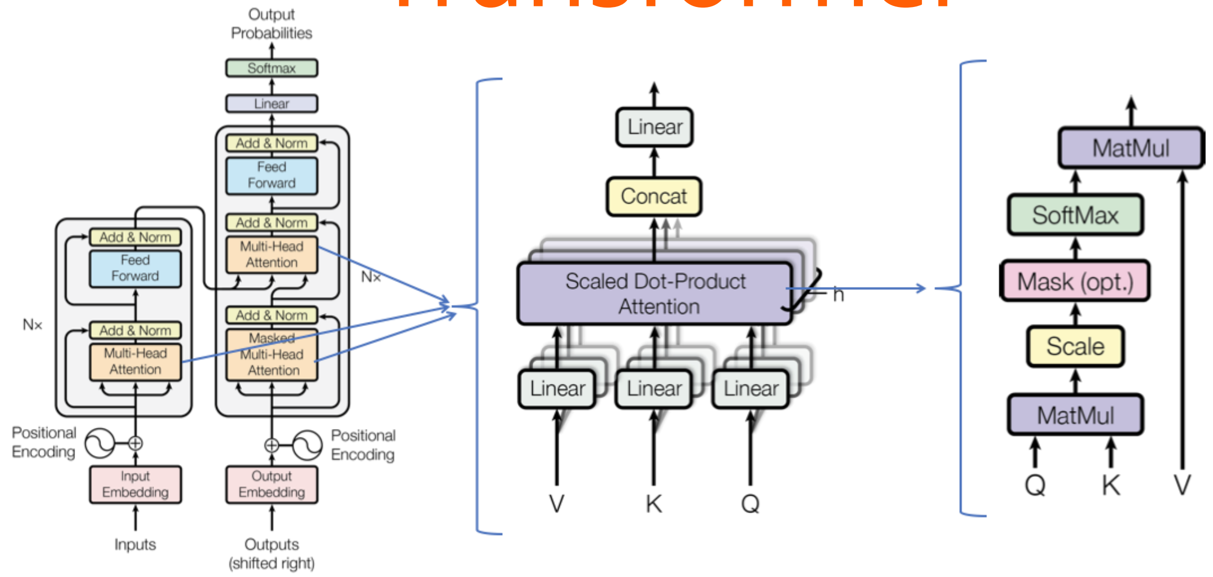
- The size of the filter is pre-defined for a CNN.
- The advantage of GNN is it processes **arbitrary graphs** with different degrees for each node.
- The advantage of CNN is that it's **more expressive** since it can leverage the canonical order in an image

CNN is not permutation invariant/equivariant.

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

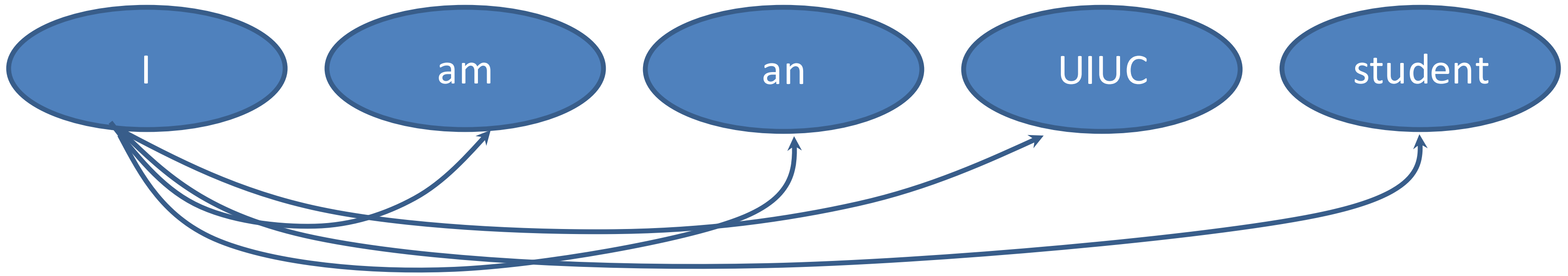
Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can define a canonical order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)} (in other words, image is an ordered data structure)

Transformer



Key component: self-attention

- Every token/word attends to all the other tokens/words via matrix calculation.



Recall: GAT is a Sparse Transformer

- **Normalize** e_{vu} into the **final attention weight** $\boldsymbol{\alpha_{vu}}$
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

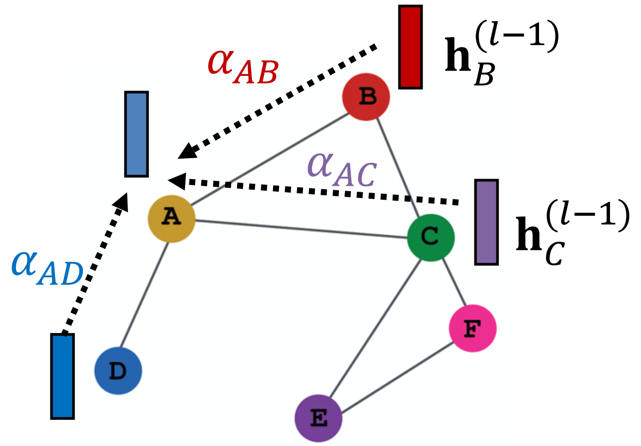
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu}

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Weighted sum using α_{AB}, α_{AC}, α_{AD}:

$$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)})$$

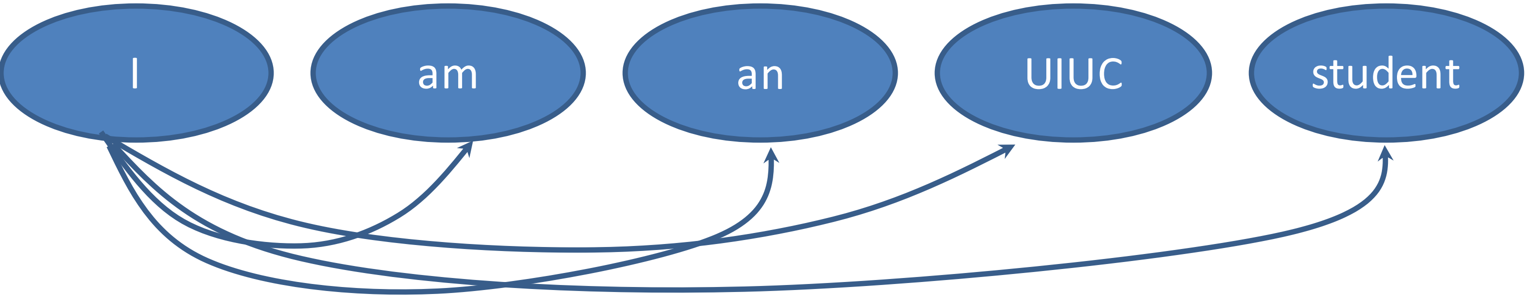


https://towardsdatascience.com/transformers-are-graph-neural-networks-bca9f75412aa

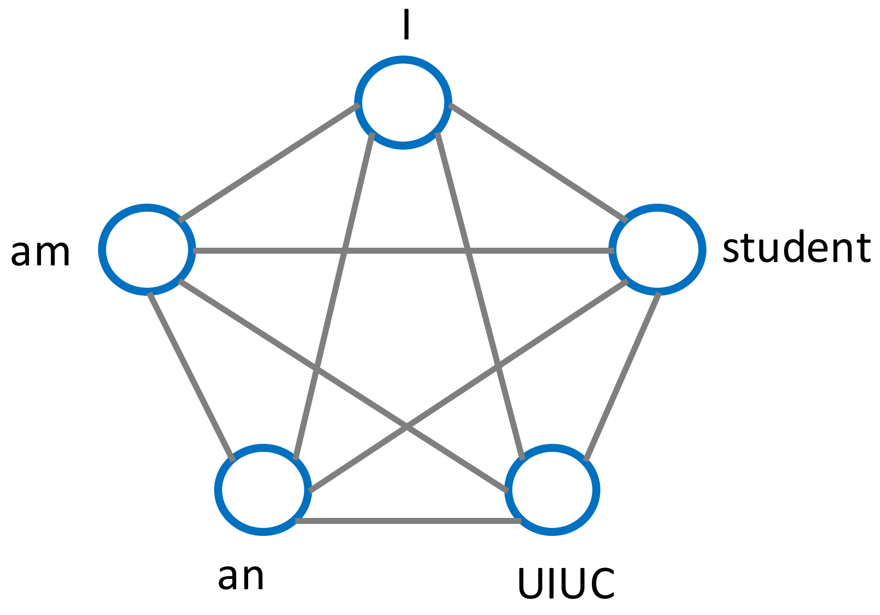
GNN vs. Transformer

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**.



Text



(Fully-connected) Graph

GNN vs. Transformer

	GNN	Transformer
Model architecture	A large & flexible model design space, including GAT that mimics Transformer	A specific NN architecture (minor changes have been made over the years)
Input data – order info	Permutation invariant, no positional encoding	Sequence order -> positional encoding (abs encoding added in the 1st layer, relative encoding added every layer)
Input data – relational info	Encoded as **sparsity pattern in attention**	No relational info -> fully connected graph
Output	Node embeddings + diverse node/edge/graph prediction heads	Node/Token embeddings. Could have "node/edge/graph" heads, but mostly node prediction heads

Summary of the Lecture

- **We introduce a general GNN framework:**
  - **GNN Layer**:
    - Transformation + Aggregation. Classic GNN layers: GCN, GraphSAGE, GAT
  - **Layer connectivity**:
    - The over-smoothing problem. Solution: skip connections
  - **Graph Augmentation:**
    - Feature augmentation. Structure augmentation
  - **Learning Objectives**
    - The full training pipeline of a GNN
- And **compared GNN to other common NNs**: CNNs & Transformer