Yale

Training Graph Neural Networks

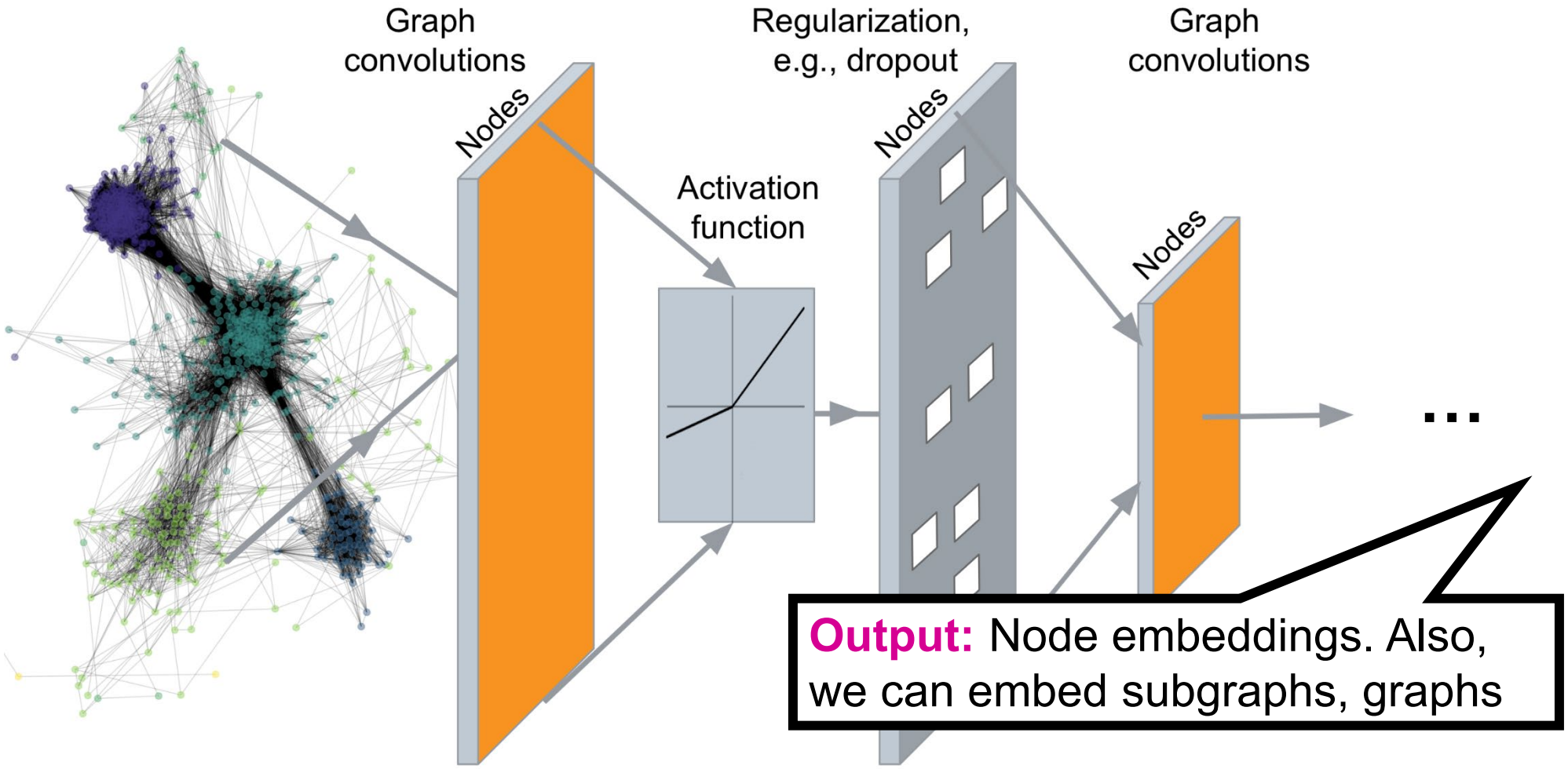
CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Readings

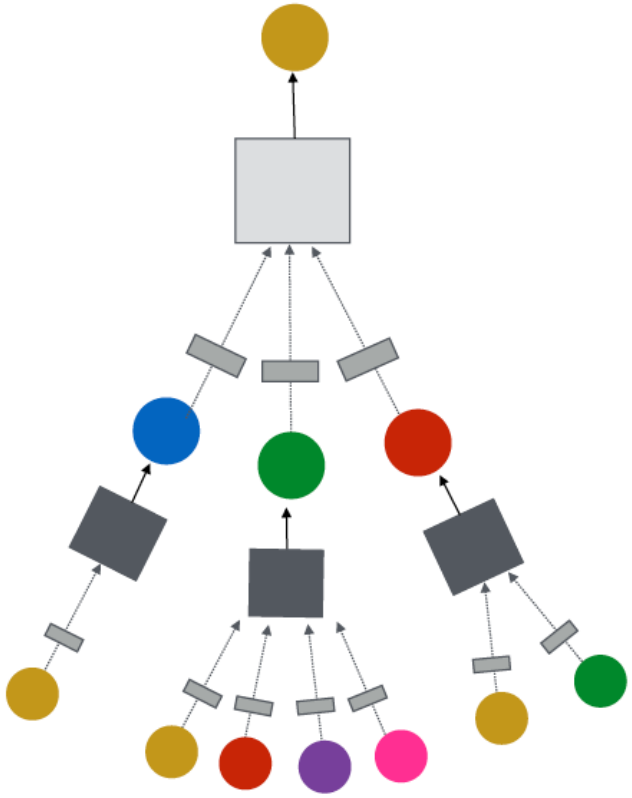
- Readings are updated on the website (syllabus page)
- **Lecture 4 readings**:
  - Semi-Supervised Classification with Graph Convolutional Networks
  - Principled Neighborhood Aggregation on Graph Nets
- **Lecture 5 readings:**
  - Design Space of Graph Neural Networks
  - OGB Datasets

Recap: Deep Graph Encoders



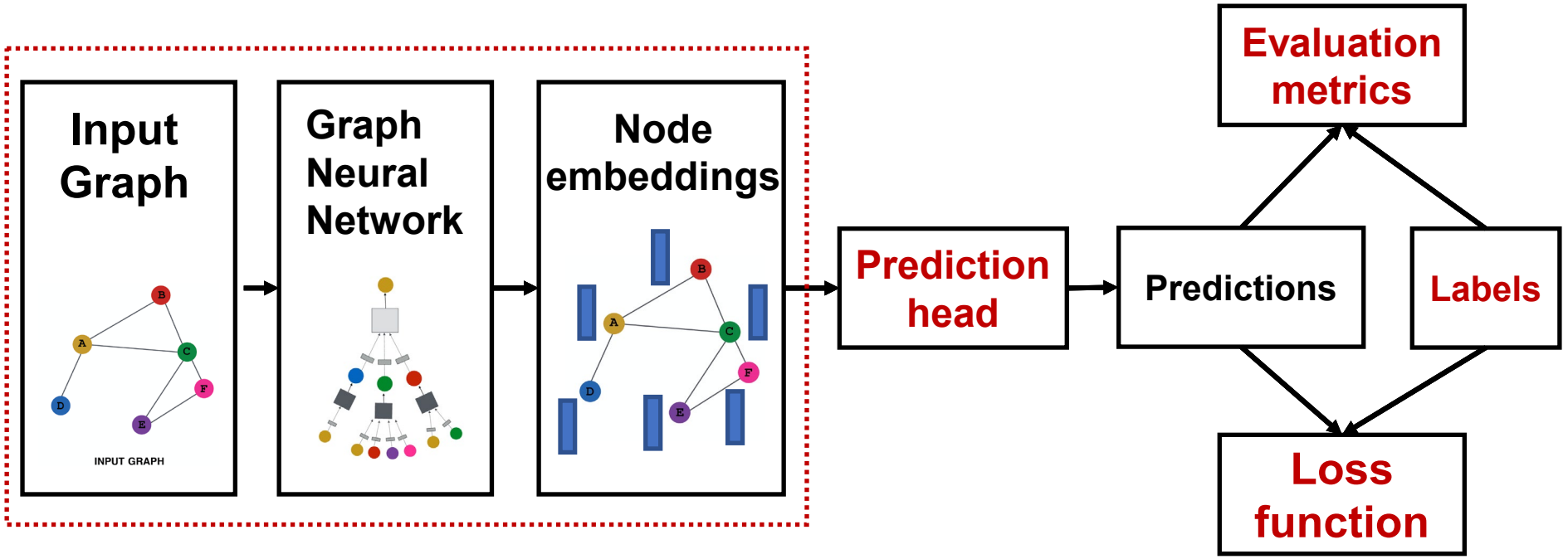
Recap: A General GNN Framework

Next: How do we train / test a GNN?



GNN Training Pipeline (1)

- **So far what we have covered**



- **Output of a GNN: set of node embeddings**

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

Outline of Today's Lecture

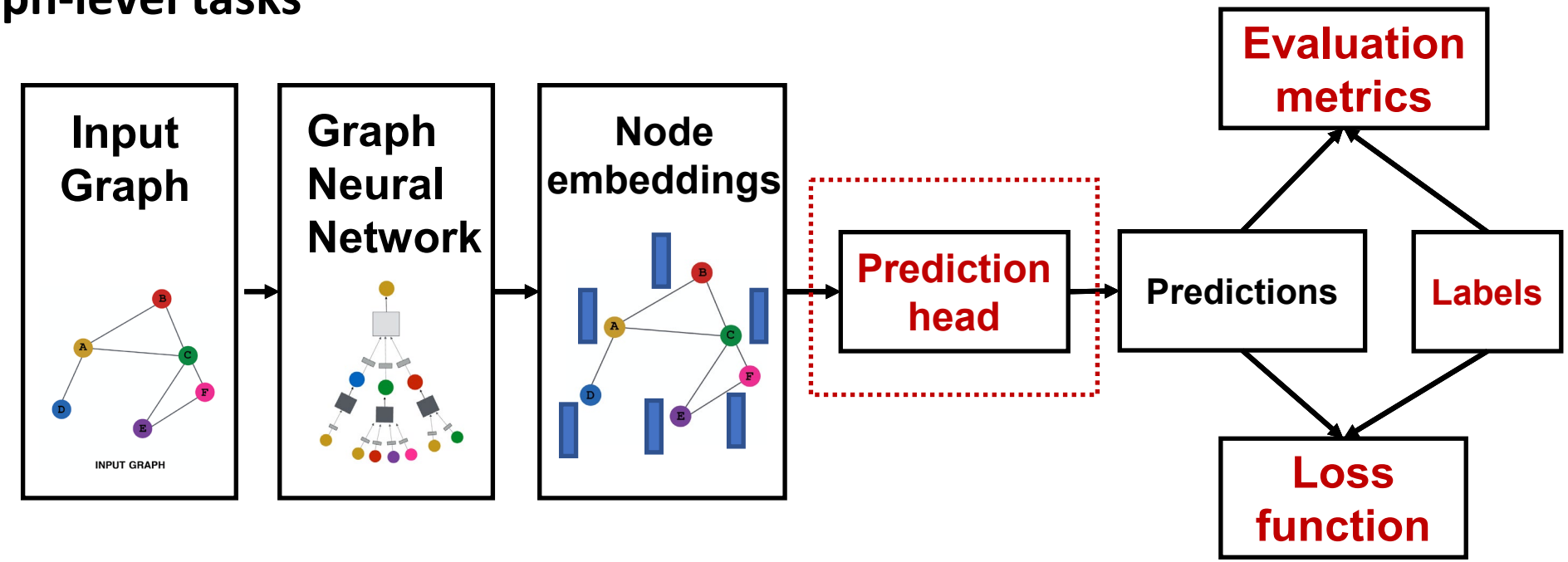
- **GNN Prediction Heads**
- **GNN Predictions & Labels**
- **GNN Loss Functions**
- **GNN Evaluation Metrics**
- **GNN Dataset Split**

Part 1

GNN Prediction Heads

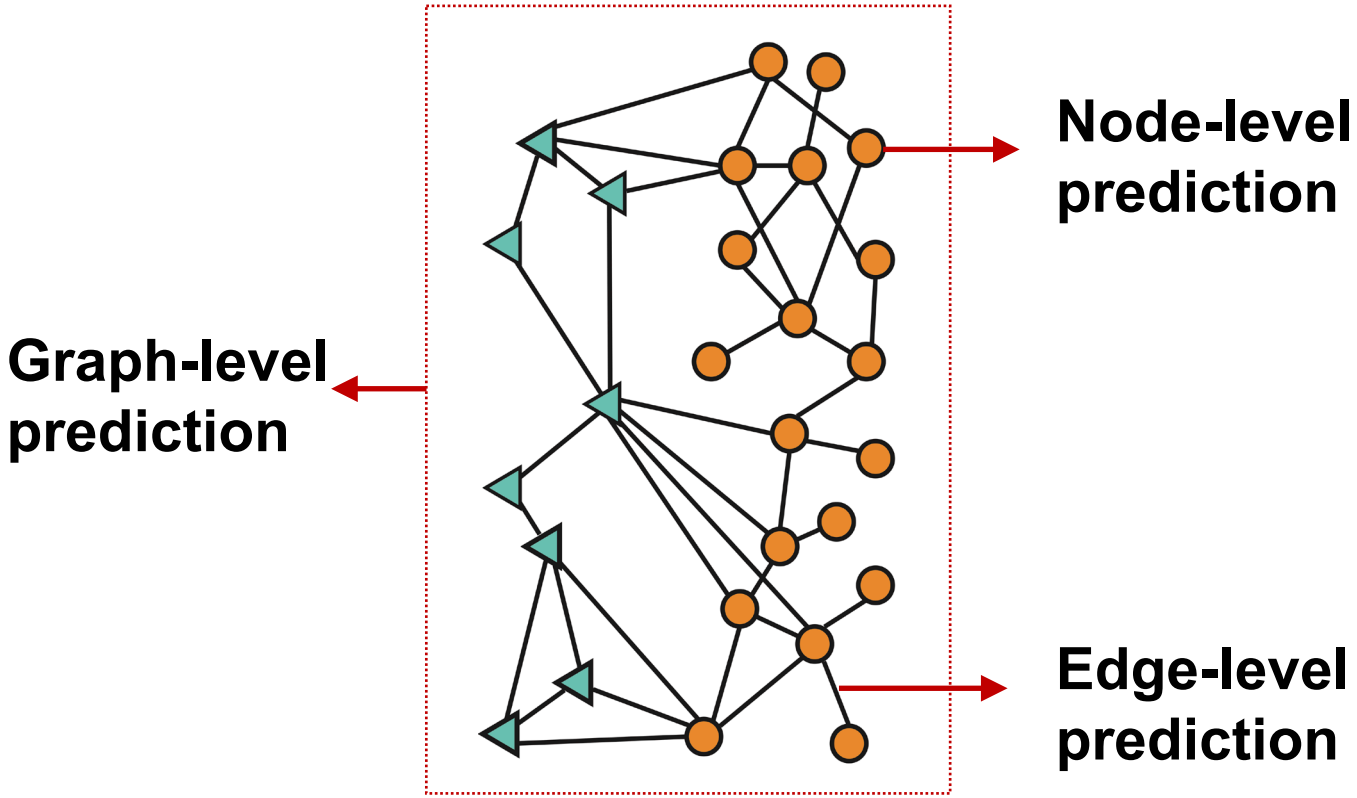
GNN Training Pipeline (2)

- **(1) Different prediction heads:**
  - **Node-level tasks**
  - **Edge-level tasks**
  - **Graph-level tasks**



GNN Predictions Heads

- Idea: Different task levels require different prediction heads

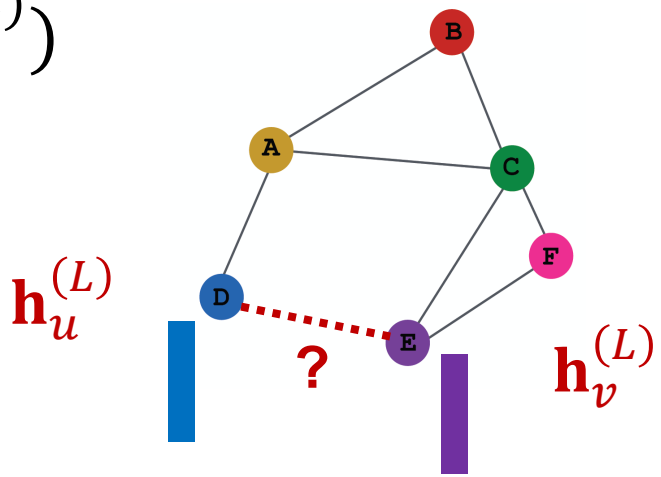


Prediction Heads: Node-Level

- **Node-level prediction**: We can directly make prediction using node embeddings!
- After GNN computation, we have d-dim node embeddings: $\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$
- Suppose we want to make k-way prediction
  - Classification: classify among k categories
  - Regression: regress on k targets
- $\hat{\boldsymbol{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)}\mathbf{h}_v^{(L)}$
  - $\mathbf{W}^{(H)} \in \mathbb{R}^{k \times d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{\boldsymbol{y}}_v \in \mathbb{R}^k$ so that we can compute the loss

Prediction Heads: Edge-Level (1)

- **Edge-level prediction**: Make prediction using pairs of node embeddings
- Suppose we want to make a k-way prediction
- $\hat{\boldsymbol{y}}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



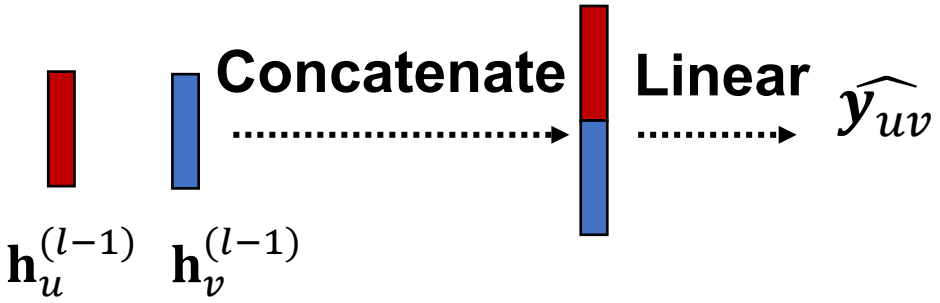
- What are the options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Prediction Heads: Edge-Level (2)

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:

- **(1) Concatenation + Linear**

  - $\hat{\boldsymbol{y}}_{uv} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$



We will see it in Graph Attention (lecture 7)

  - Here $\text{Linear}(\cdot)$ will map 2-dimensional embeddings (since we concatenated embeddings) to k-dimentional embeddings (**k-way** prediction)

Prediction Heads: Edge-Level (3)

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:

- **(2) Dot product**
  - $\hat{\boldsymbol{y}}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
  - **This approach only applies to 1-way prediction**
    (e.g., link prediction: predict the existence of an edge)
  - **Applying to k-way prediction:**
    - Let $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(k)}$ be trainable weights:

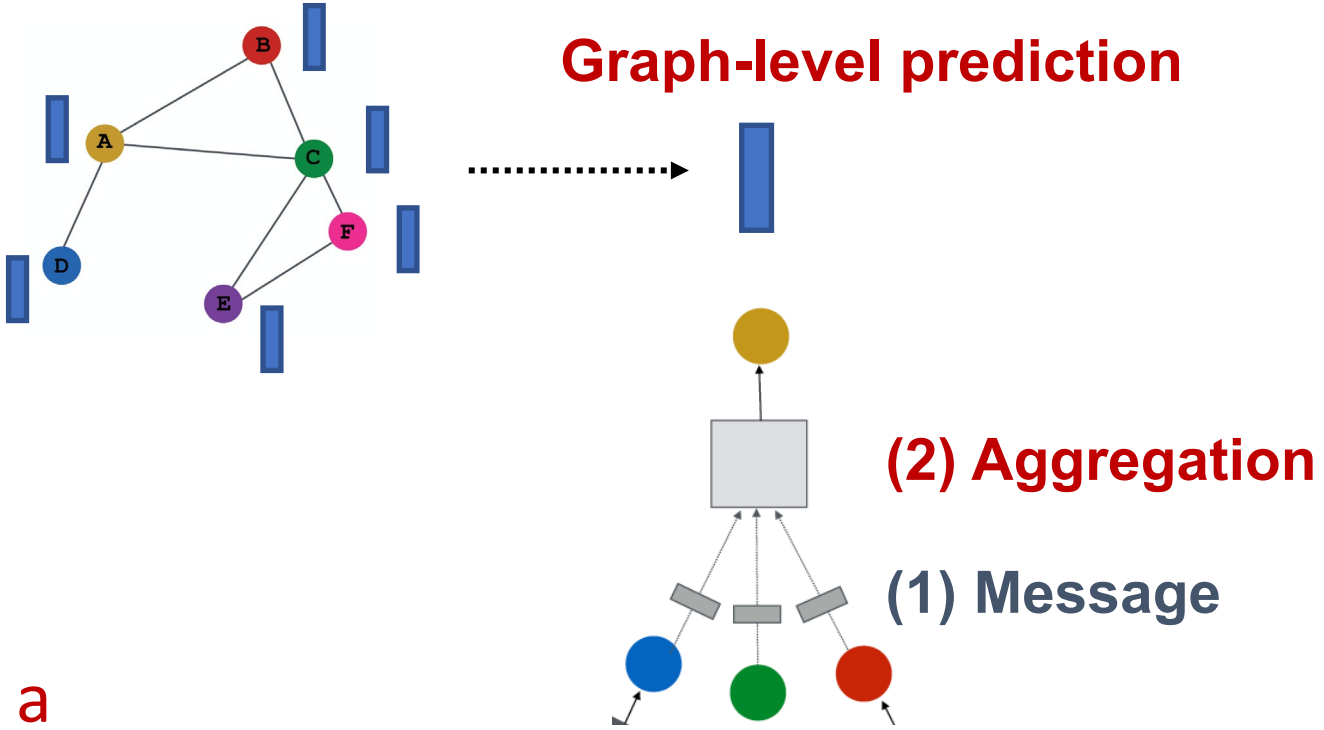
$$\hat{\boldsymbol{y}}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$
$$\dots$$
$$\hat{\boldsymbol{y}}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$
$$\hat{\boldsymbol{y}}_{uv} = \text{Concat}(\hat{\boldsymbol{y}}_{uv}^{(1)}, \dots, \hat{\boldsymbol{y}}_{uv}^{(k)}) \in \mathbb{R}^k$$

We will see it in Graph Attention (lecture 7)

Also called bilinear form

Prediction Heads: Graph-Level (1)

- **Graph-level prediction**: Make prediction using all the node embeddings in our graph
- Suppose we want to make k-way prediction
- $\hat{\boldsymbol{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- $\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!



Prediction Heads: Graph-Level (2)

- 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**

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

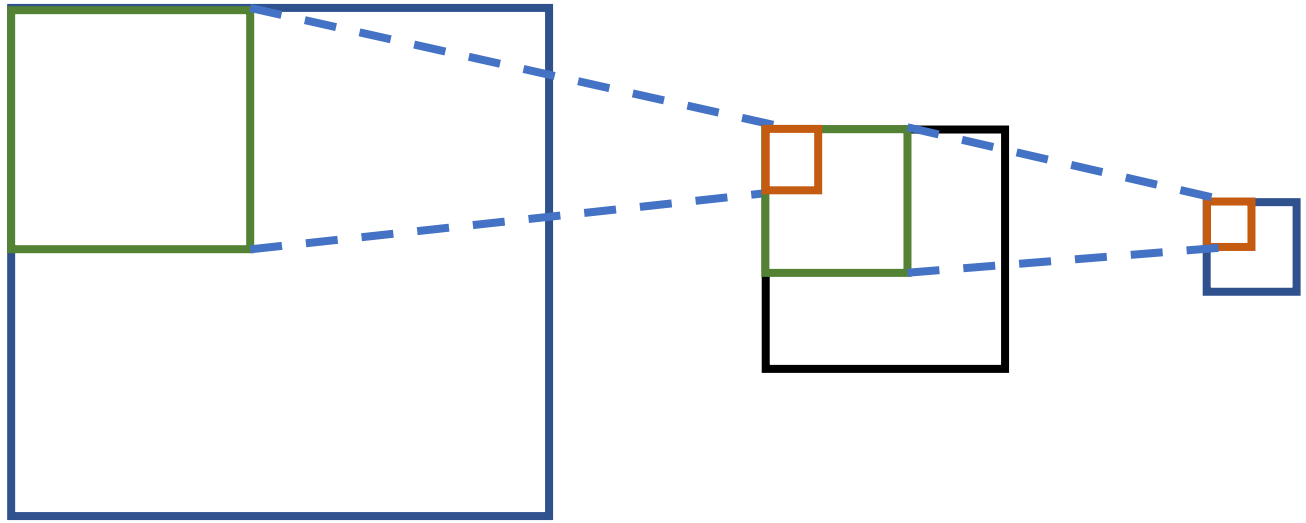
- These options work great for small graphs
- **Can we do better for large graphs?**

Issue of Global Pooling (1)

- **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 → Their graph structures should be different
- **If we do global sum pooling:**
  - **Prediction for** G_1: $\hat{y}_{G_1} = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
  - **Prediction for** G_2: $\hat{y}_{G_2} = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
  - We cannot differentiate G_1 and G_2!

Issue of Global Pooling (2)

- Aggregate node embeddings
  - Naive approach: global mean/max/sum
  - Better pooling strategies that respects structure?
  - Here we consider **hierarchical pooling analogous to CNNs**

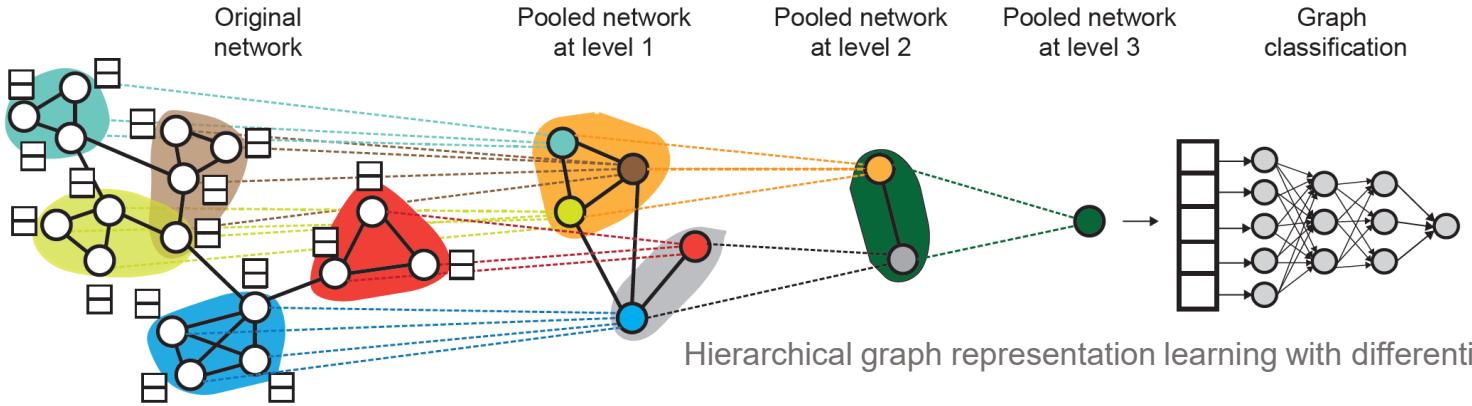


Hierarchical Global Pooling

- **A solution:** Let's aggregate all the node embeddings **hierarchically**
  - **Toy example:** We will aggregate via $\mathrm{ReLU}(\mathrm{Sum}(\cdot))$
    - 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**: $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,1, 2\})) = 3$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{3}$
  - G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
    - **Round 1**: $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,10, 20\})) = 30$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{30}$
- **Now we can differentiate G_1 and G_2 !**

Hierarchical Pooling in Practice

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



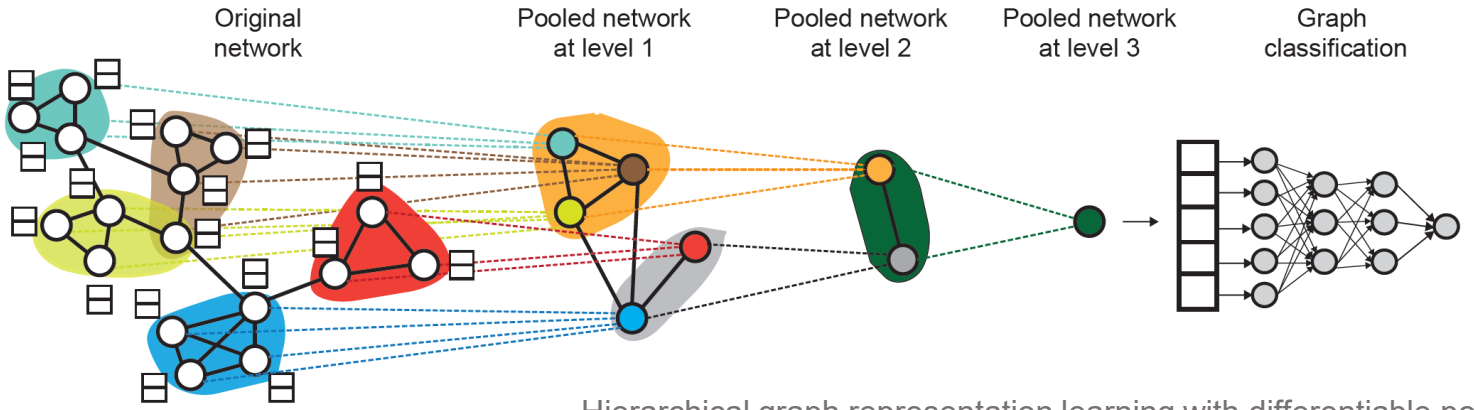
Hierarchical graph representation learning with differentiable pooling, NeurIPS 2018

  - **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

Every level has a different graph structure (different "resolution")

- **DiffPool idea:**



Hierarchical graph representation learning with differentiable pooling, NeurIPS 2018

- **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** (optimized together end-to-end)

DiffPool Architecture (1)

- Assuming general GNN model:

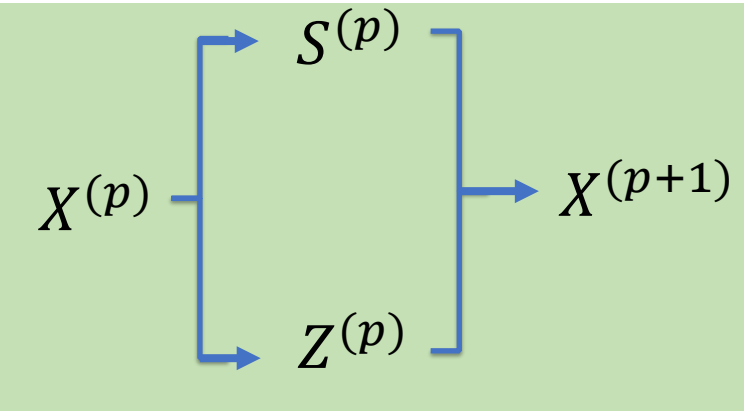
$$H^{(k)} = M\left(A, H^{(k-1)}; \theta^{(k)}\right)$$

- For example: $\mathrm{ReLU}\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k-1)}W^{(k-1)}\right)$
- Two-tower architecture

GNN A: $Z^{(p)} = \mathrm{GNN}_{l,\,\mathrm{embed}}\left(A^{(p)}, X^{(p)}\right)$ **Embedding**

GNN B: $S^{(p)} = \mathrm{Softmax}\left(\mathrm{GNN}_{l,\mathrm{pool}}\left(A^{(l)}, X^{(p)}\right)\right)$ **Assignment**

At every level p



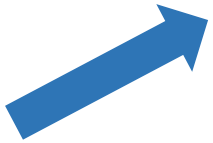
- Combine them to generate next-layer representations and adjacency

DiffPool Architecture (2)

- Let n_l be the number of nodes at level p
- The Assignment matrix S dimension is: $n_l \times n_{l+1}$
  - $S_{i,j} = 1$ if i-th node for layer l belongs to j-th node/cluster in layer $l+1$
- Having computed Z and S for level l, we can compute embeddings $X^{(l+1)}$ and adjacency matrix $A^{(l+1)}$ for level $l+1$

DiffPool Next-layer Embeddings and Adjacency

- **Having computed Z and S:**



Compute the embedding for nodes at layer $l+1$ which includes low-level nodes at layer l

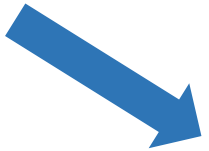
- Compute **embedding** for layer $l+1$:

$$X^{(l+1)} = S^{(l)^T} Z^{(l)} \in \mathbb{R}^{n_{l+1} \times d}$$

  - A sum aggregation of embeddings into nodes at the next level

- Compute **adjacency matrix** for layer $l+1$:

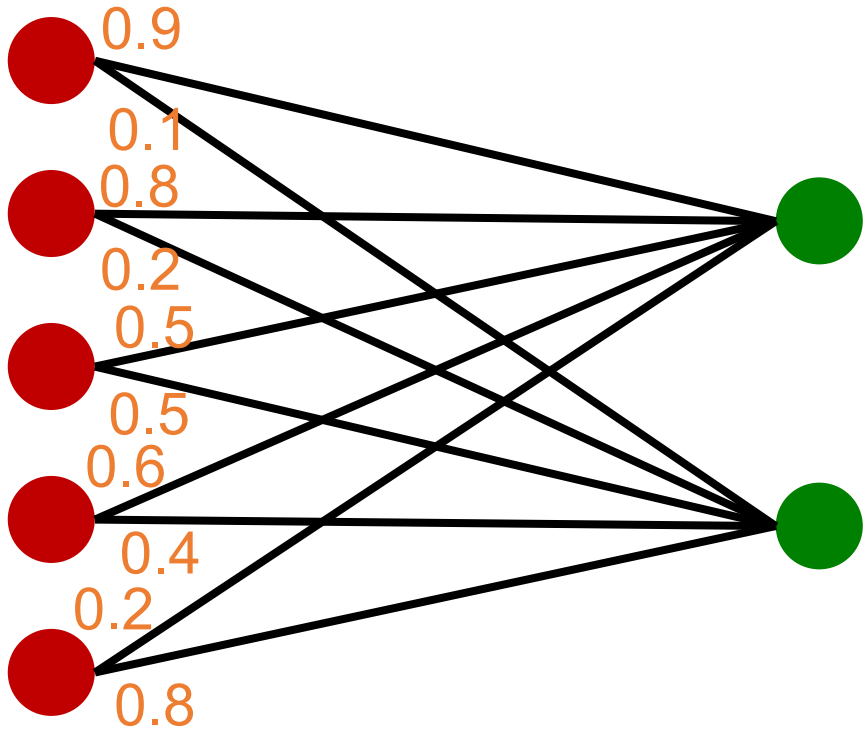
$$A^{(l+1)} = S^{(l)^T} A^{(l)} S^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}$$



Two level $l+1$ nodes are connected iff their level l nodes have connections

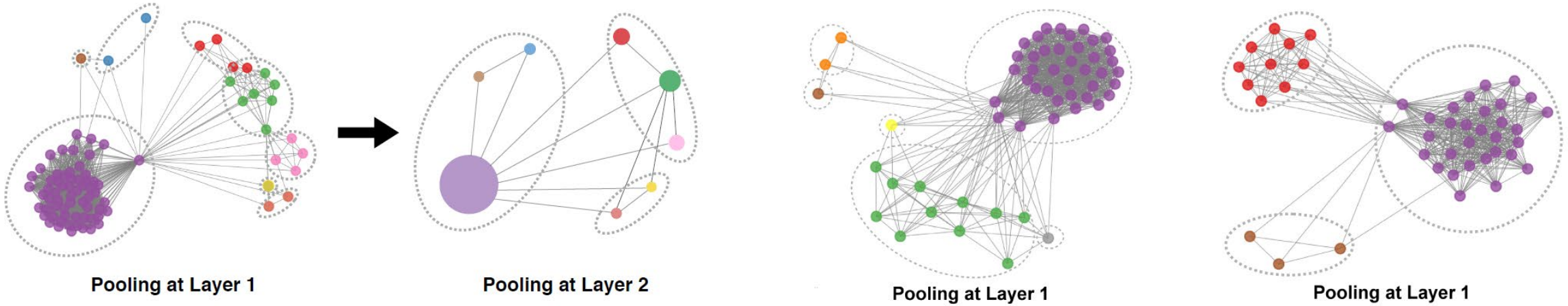
DiffPool Architecture: Example

- Computing node embeddings:
  - $S^{(l)} = \begin{pmatrix} 0.9 & 0.1 \\ 0.8 & 0.2 \\ 0.5 & 0.5 \\ 0.6 & 0.4 \\ 0.2 & 0.8 \end{pmatrix}$
- Computing adjacency matrix is analogous to this!



DiffPool Results

- Learns meaning clusters through downstream tasks
- An average of 6.27% improvement in accuracy for standard benchmarks



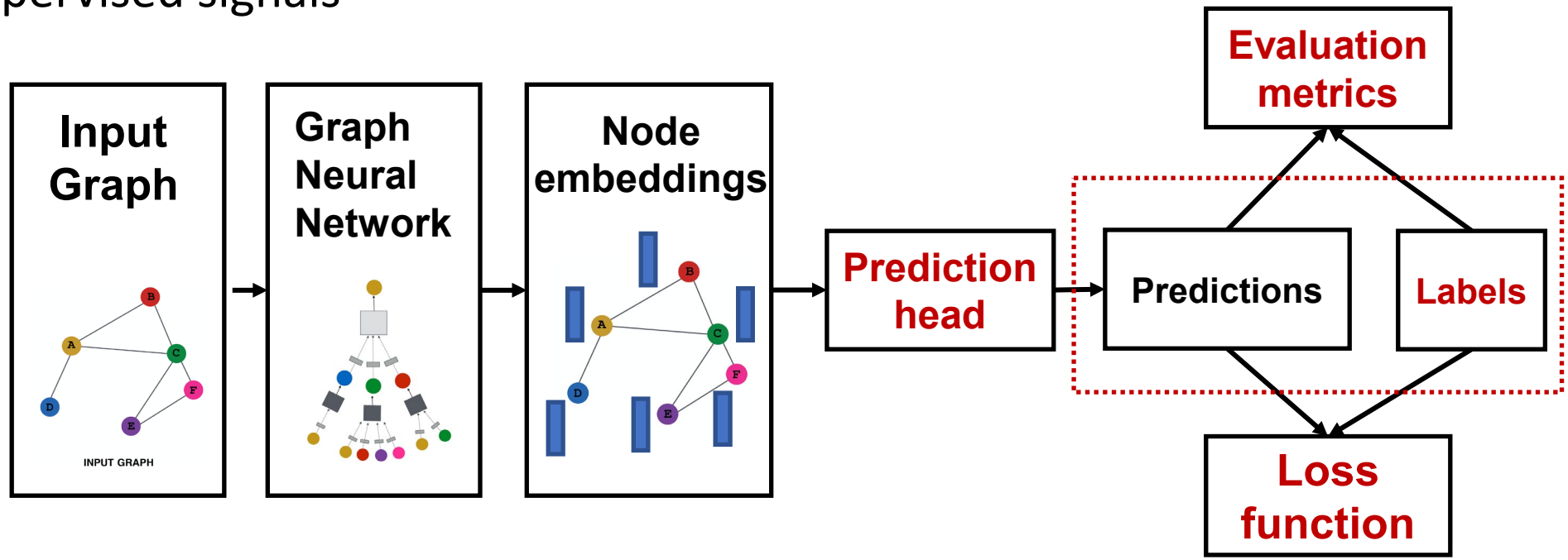
Hierarchical graph representation learning with differentiable pooling, NeurIPS 2018

Part 2

GNN Predictions & labels

GNN Training Pipeline (3)

- Where does ground-truth come from?
  - Supervised labels
  - Unsupervised signals



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**"

Supervised Labels on Graphs

- **Supervised labels come from the specific use cases**. For example:
  - **Node labels y_v:** in a citation network, which subject area does a node belong to
  - **Edge labels y_{uv}:** in a transaction network, whether an edge is fraudulent
  - **Graph labels y_G:** among molecular graphs, the drug likeness of graphs
- **Advice:** Reduce your task to node / edge / graph labels, since they are easy to work with
  - **E.g.,** we knew some nodes form a cluster. We can treat the cluster that a node belongs to as a **node label**

Unsupervised Signals on Graphs

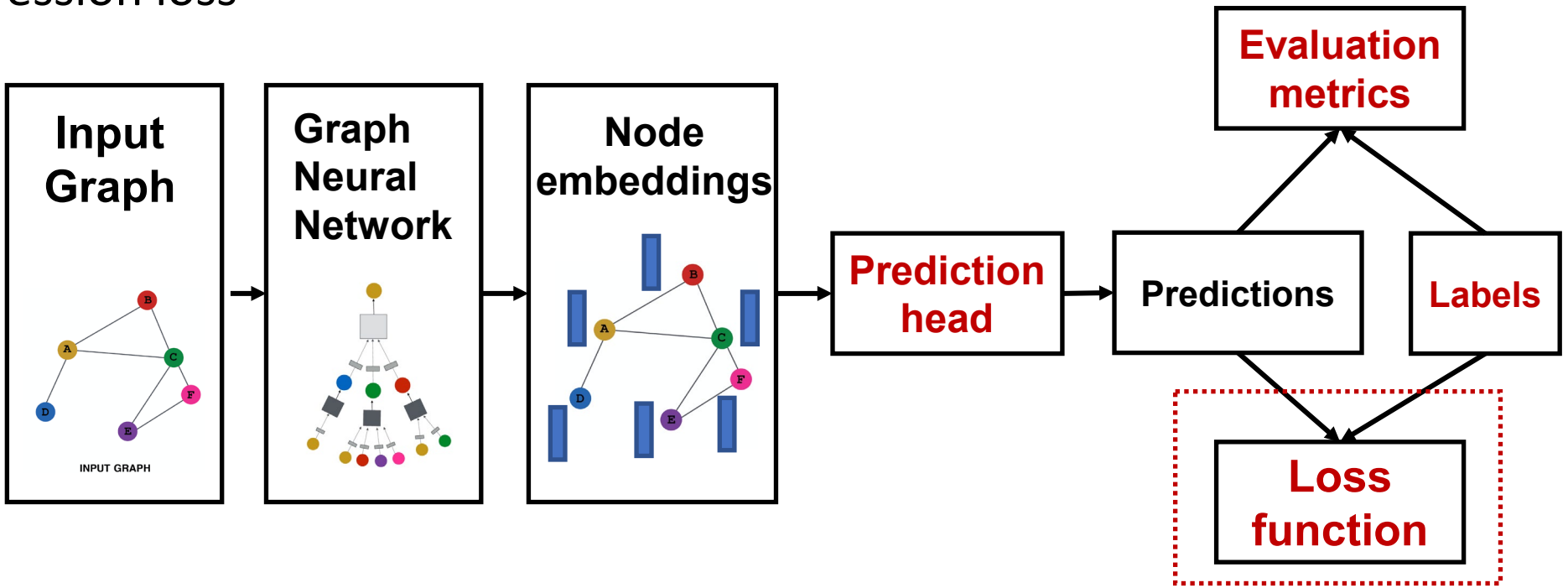
- **The problem:** sometimes **we only have a graph, without any external labels**
- **The solution:** "self-supervised learning", we can find supervision signals within the graph.
  - For example, we can **let GNN predict the following:**
  - **Node-level y_v.** Node statistics: such as clustering coefficient, PageRank, …
  - **Edge-level y_{uv}.** Link prediction: hide the edge between two nodes, predict if there should be a link
  - **Graph-level y_G.** Graph statistics: for example, predict if two graphs are isomorphic
  - **These tasks do not require any external labels!**

Part 3

GNN Loss Functions

GNN Training Pipeline (4)

- 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 tasks**: prediction $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level tasks**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level tasks**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$

  - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer to **predictions at all tasks**

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

$$\mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log\left(\hat{\boldsymbol{y}}_j^{(i)}\right)$$

Label Prediction

***i*-th data point**

***j*-th class**

where:

$$\boldsymbol{y}^{(i)} \in \mathbb{R}^K = \text{one-hot label encoding (e.g. } [0, 0, 1, 0, 0])$$
$$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K = \text{prediction after Softmax}(\cdot) \text{ (e.g. } [0.1, 0.3, 0.4, 0.1, 0.1])$$

- Total loss over all N training examples

$$\mathcal{L} = \sum_{i=1}^{N} \mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

Regression Loss

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

$$\text{MSE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) = \sum_{j=1}^{K} (\boldsymbol{y}_j^{(i)} - \hat{\boldsymbol{y}}_j^{(i)})^2$$

Label **Prediction**

i-th data point

j-th dimension

where:

$\boldsymbol{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets (e.g. $[1.4, 2.3, 1.0, 0.5, 0.6]$)

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions(e.g. [0.9, 2.8, 2.0, 0.3, 0.8])

- Total loss over all N training examples

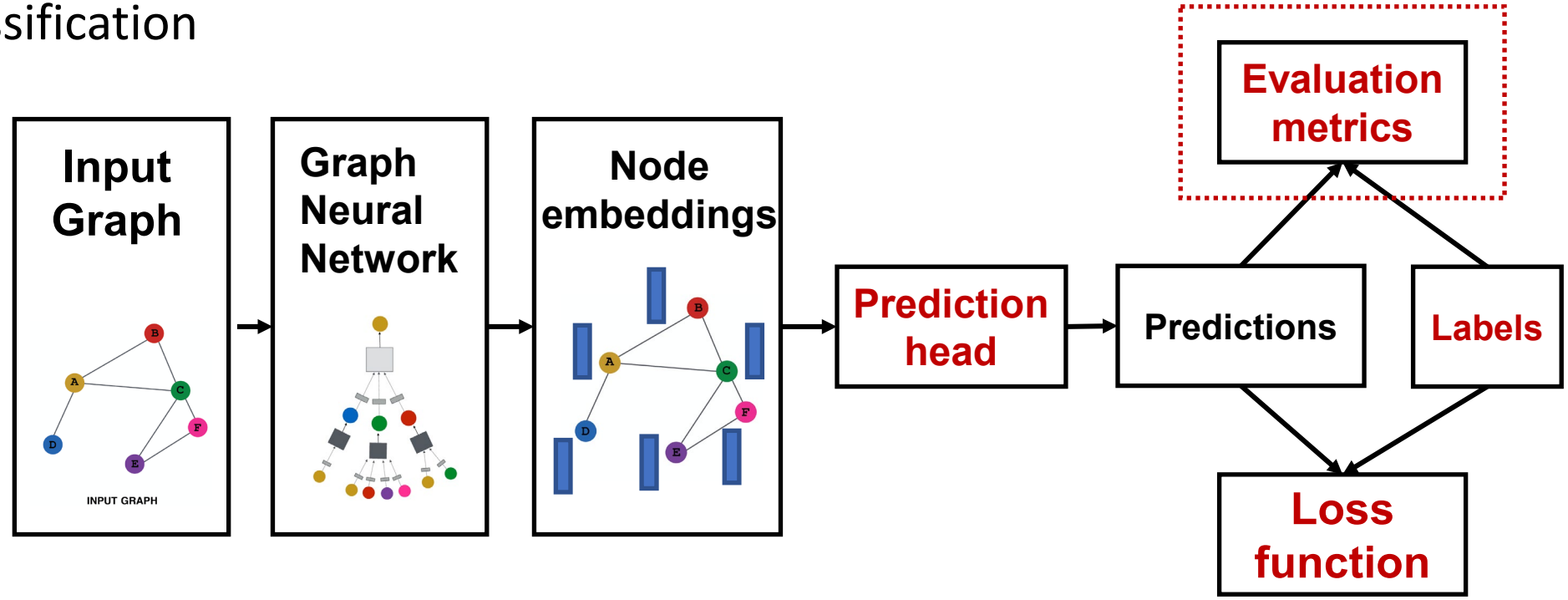
$$\mathcal{L} = \sum_{i=1}^{N} \text{MSE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)})$$

Part 4

GNN Evaluation Metrics

GNN Training Pipeline (5)

- How do we measure the success of a GNN?
  - Regression
  - Classification



Evaluation Metrics: Regression

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

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

  - **Mean absolute error (MAE)**

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

Evaluation Metrics: Classification

- **Evaluate classification tasks on graphs:**
- **(1) Multi-class classification**
  - **We simply report the accuracy (confusion matrix can also be used)**

$$\frac{\mathbf{1}\left[\text{argmax}\left(\hat{\boldsymbol{y}}^{(i)}\right) = \boldsymbol{y}^{(i)}\right]}{N}$$

- **(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**

Classification Metrics: Precision / Recall

- **Accuracy**:

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

- **Precision (P)**:

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

- **Recall (R)**:

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

- **Micro-F1 Score**:

$$\frac{2\text{PR}}{\text{P} + \text{R}}$$

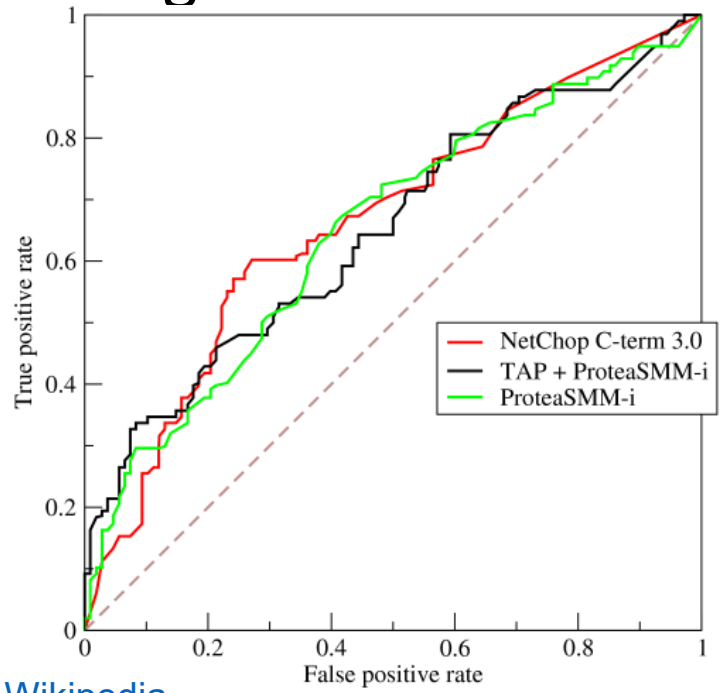
Confusion matrix

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

Classification Metrics: ROC (2)

- **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

TPR



FPR

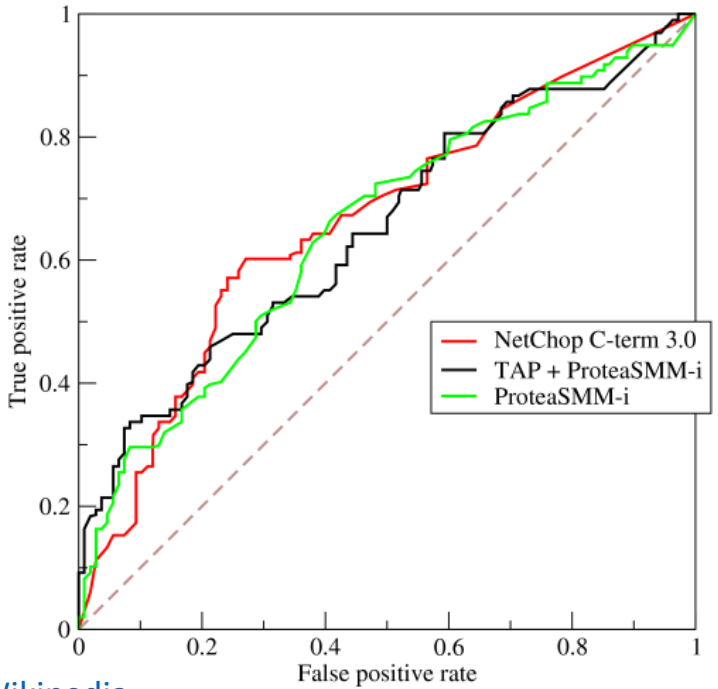
Image Credit: Wikipedia

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

Classification Metrics: ROC (1)

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

TPR



FPR

Image Credit: Wikipedia

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

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

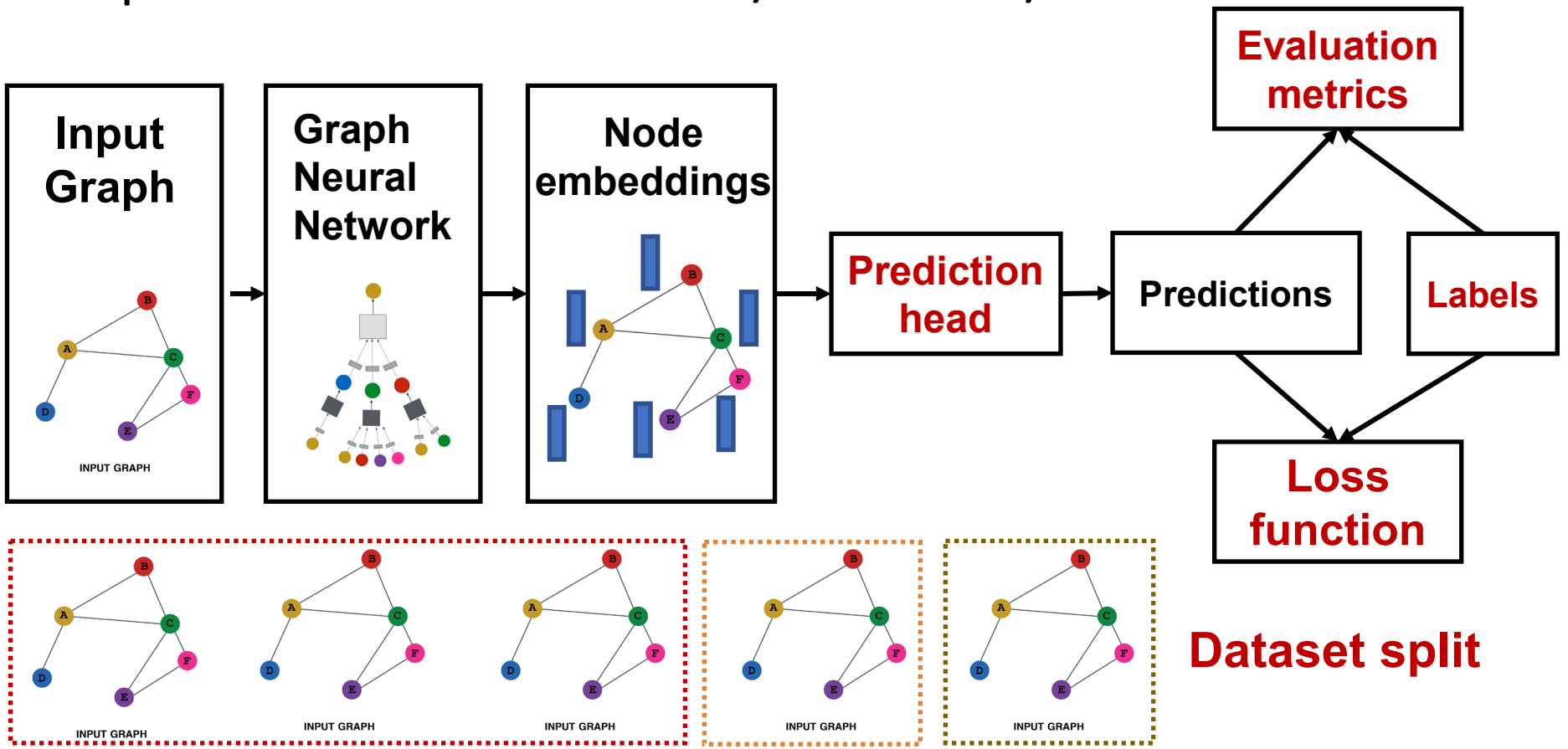
Note: the dashed line represents **performance of a random classifier**

Part 5

GNN Dataset Split

GNN Training Pipeline (6)

- How do we split our dataset into train/validation/test set?

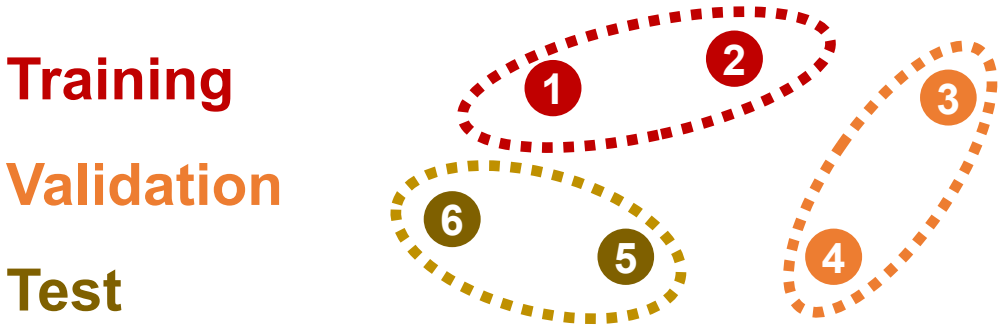


Dataset Split: Fixed / Random Split

- **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 sets
  - We report **average performance over different random seeds**
  - Split ratio: the ratio of the training / validation / test set sizes (e.g. 0.8/0.1/0.1)

Why Splitting Graphs is Special (1)

- **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**



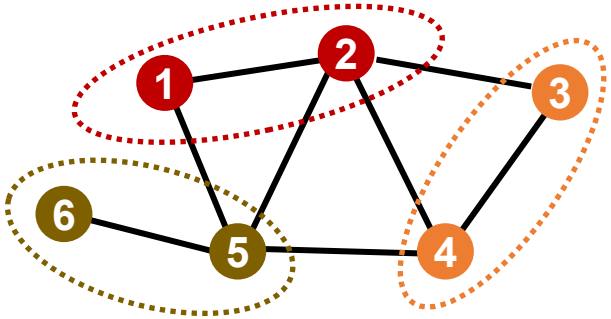
Why Splitting Graphs is Special (2)

- **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

Training

Validation

Test



- **What are our options?**

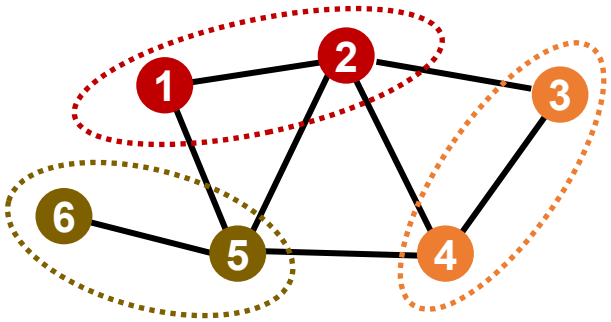
Why Splitting Graphs is Special (3)

- **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



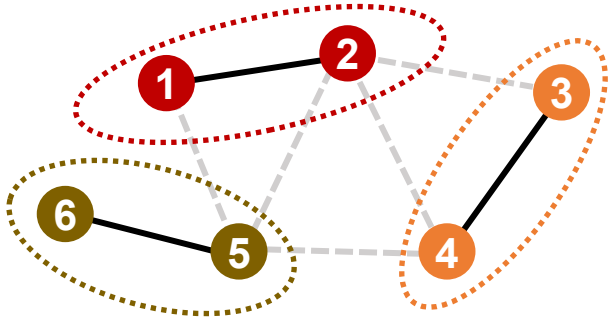
Why Splitting Graphs is Special (4)

- **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 3&4**, and **evaluate on node 3&4's labels**

Training

Validation

Test

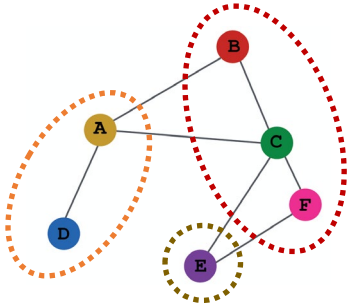


Transductive / Inductive Settings

- **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

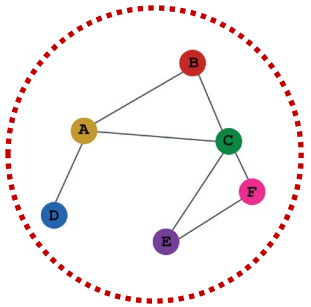


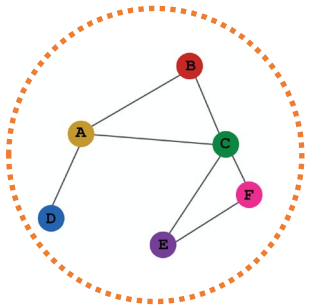
Training

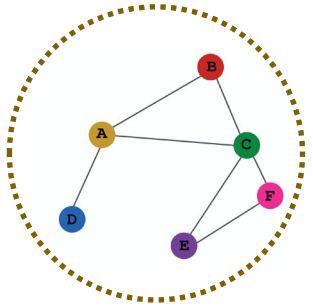
Validation

Test

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







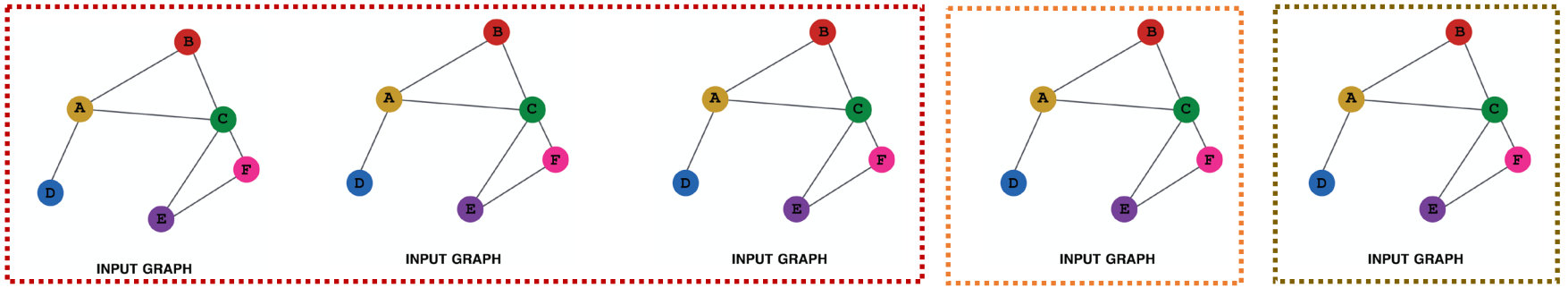
Training

Validation

Test

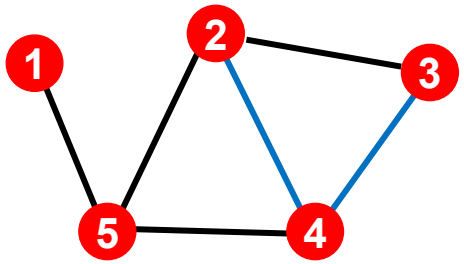
Example: Graph Classification

- 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).

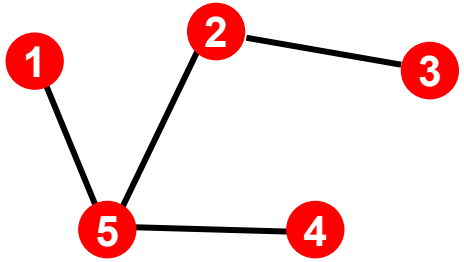


Example: Link Prediction

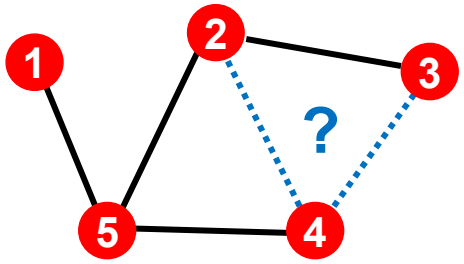
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let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



Original graph

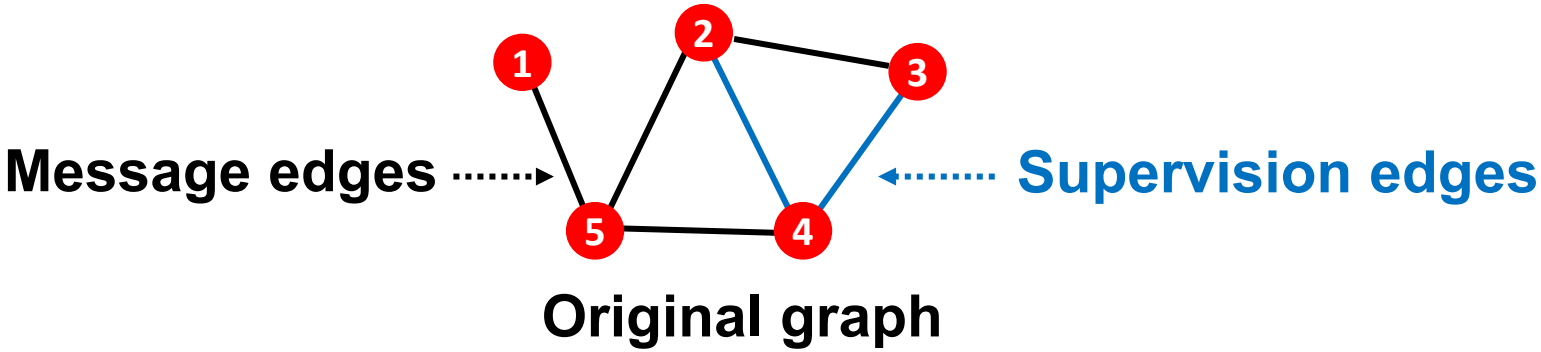


Input graph to GNN



Predictions made by GNN

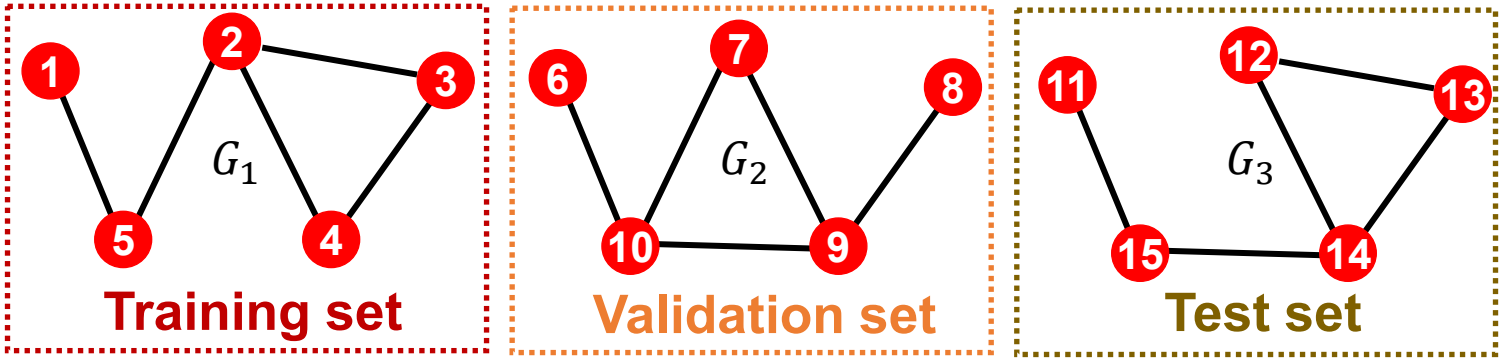
Setting up Link Prediction



- **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!**

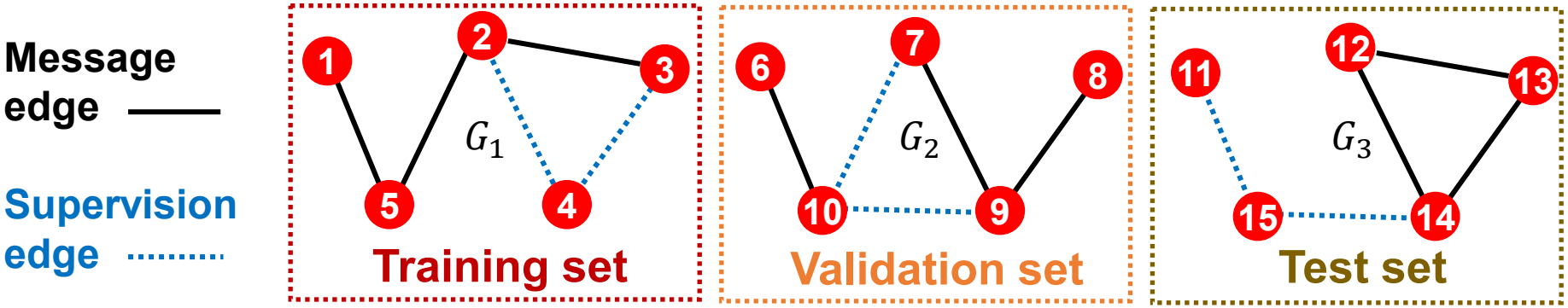
Setting up Link Prediction

- **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**



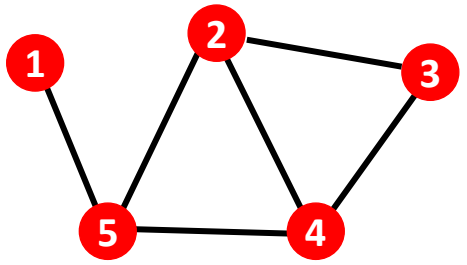
Setting up Link Prediction

- **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



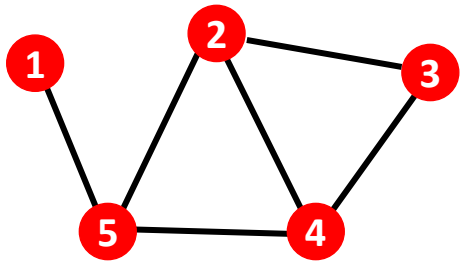
Setting up Link Prediction

- **Option 2: Transductive link prediction split:**
  - **This is often the default setting when people talk about link prediction**
  - **Suppose we have a dataset of 1 (potentially very large) 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**



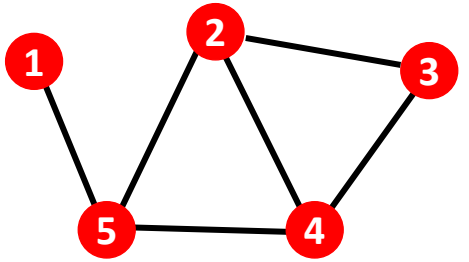
  - **Next:** we will show the exact settings

Setting up Link Prediction (1)

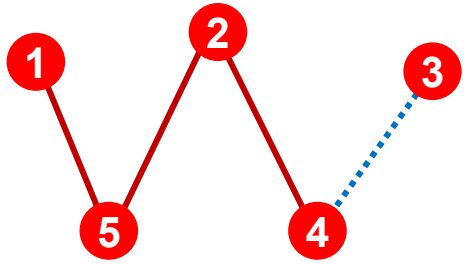
- **Option 2: Transductive link prediction split:**

Why do we use growing number of edges?

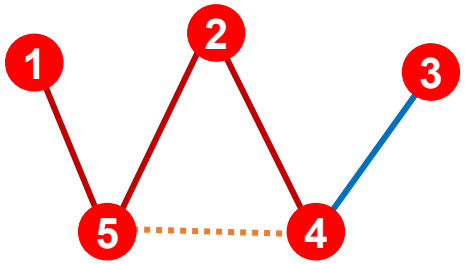
- **After training, supervision edges are known to GNN. Therefore, an ideal model should use supervision edges in message passing at validation time.**
- **The same applies to the test time.**



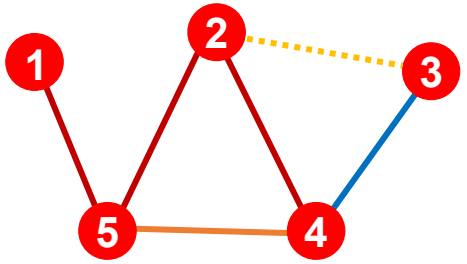
The original graph



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



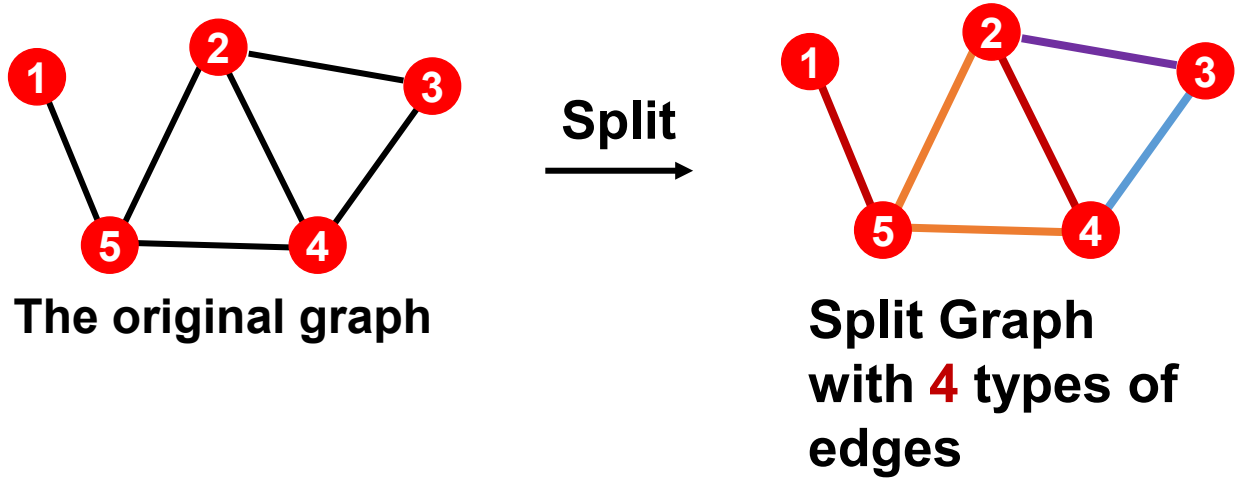
(2) At validation time: Use **training message edges & training supervision edges** to predict **validation edges**



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

Setting up Link Prediction (2)

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



The original graph

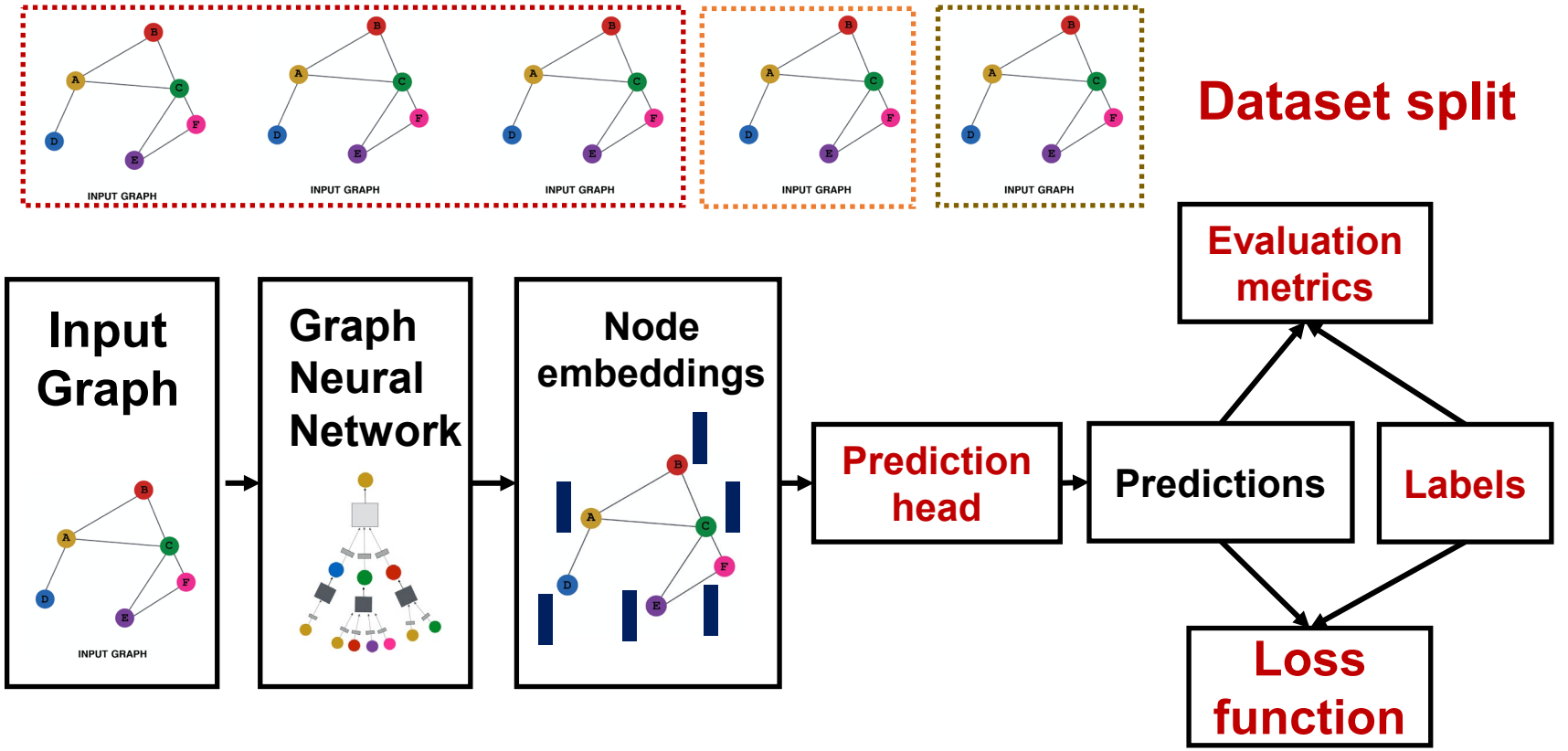
Split Graph with 4 types of edges

Training message edges
Training supervision edges
Validation edges
Test edges

- **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently. But if you follow our reasoning steps, **this should be the right way to implement link prediction**
- Luckily, we have full support in **DeepSNAP** and **GraphGym**

GNN Training Pipeline (7)

- **Implementation resources:**
- DeepSNAP provides core modules for this pipeline
- GraphGym further implements the full pipeline to facilitate GNN design



Summary of the Lecture

- **We introduce the pipeline of GNN training**
  - **Prediction Heads**:
    - Node-level / Edge-level / Graph-level
  - **Predictions & Labels**
    - Supervised / unsupervised
  - **Loss Functions:**
    - Regression / Classification
  - **Evaluation Metrics:**
    - Regression / Classification
  - **Dataset Split**
    - Transductive / inductive
    - Node / edge / graph