## Contents



# Overview

**Idea: Raw input graph ≠ computational graph**

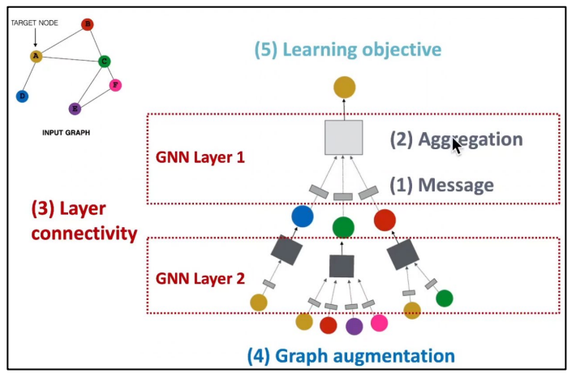
- **Graph feature augmentation**
- **Graph structure augmentation**

## GNN definition / component Recap

What is a GNN

- For every node in the network, we define a computation graph based on the network neighbourhood given a target node.

- The input graph network structure around the target node defines the GNN structure
- To make this network architecture works, since every node has its own structure, and need to define its own computational graph (depends on the position in the network)
- Needs differention operators in the architecture
  - Message passing function
  - Message transformation function
    - Transform the child and pass to the parent
  - Message aggregation
    - Take the transformed message from the children and aggregate them in a order invariant way
  - When the message arrives with the parent
    - Define how to combine with the parent own message from the previous layers to create the embedding of the node
    - Which will be passed on – single layer NN
  - How to stack multiple layers together
- Today
  - Feature augmentation



# Graph augmentation

## Why augment graph – features & sturcture

Our assumption so far has been
- Raw input graph = computational graph

Reasons for breaking this assumption
- **Features:**
  - The input graph **lacks features**
- **Graph structure:**
  - The graph is **too sparse** → inefficient message passing
  - The graph is **too dense** → message passing is too costly
  - The graph is **too large** → cannot fit the computational graph into a GPU
- It's **unlikely that the input graph happens to be the optimal computation graph** for embeddings

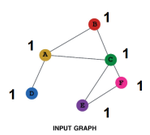
## Approaches – feature, structure augmentation

- **Graph Feature augmentation**
  - The input graph **lacks features** → **feature augmentation**
- **Graph Structure augmentation**
  - 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

## No node features

### a) assign constant

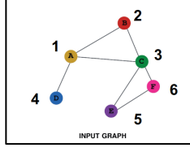
- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**




### b) Assign unique IDs to nodes

- order dependent – not generalise well
- costly when #nodes increases

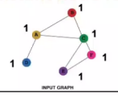
- These IDs are converted into **one-hot vectors**




One-hot vector for node with ID=5
ID = 5
[0, 0, 0, 0, 1, 0]
Total number of IDs = 6

## Table - Feature augmentation: constant vs. one-hot

- Feature augmentation: **constant** vs. **one-hot**

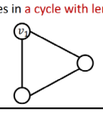
| | Constant node feature | One-hot node feature |
|---|---|---|
| |  |  |
| **Expressive power** | **Medium**. All the nodes are identical, but GNN can still learn from the graph structure | **High**. Each node has a unique ID, so node-specific information can be stored |
| **Inductive learning (Generalize to unseen nodes)** | **High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN | **Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs |
| **Computational cost** | **Low**. Only 1 dimensional feature | **High**. $O(\|V\|)$ dimensional feature, cannot apply to large graphs |
| **Use cases** | **Any graph, inductive settings** (generalize to new nodes) | **Small graph, transductive settings** (no new nodes) |

## Why feature augmentation

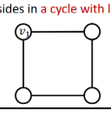
### Certain structures are hard to learn by GNN: Cycle count feature

- **Example:** Cycle count feature:
  - Can GNN learn the length of a cycle that $v_1$ resides in?
  - **Unfortunately, no**

$v_1$ resides in a cycle with length 3




$v_1$ resides in a cycle with length 4

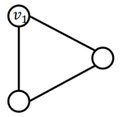



### Cannot differentiate

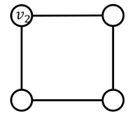
- since the nodes all have 2 neighbours
- computation graph the same unless some discriminator e.g. nodes have some colors

- $v_1$ **cannot differentiate which graph it resides in**
  - Because all the nodes in the graph have degree of 2
  - The computational graphs will be the same binary tree

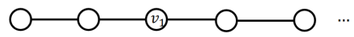
$v_1$ resides in a cycle with length 3



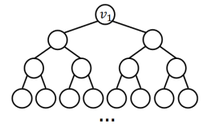

$v_1$ resides in a cycle with length 4




$v_1$ resides in a cycle with infinite length




**The computational graphs for node $v_1$ are always the same**



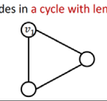

More about this topic later!

### Solution – one hot

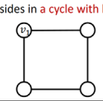
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for $v_1$
[0, 0, 0, 1, 0, 0]
$v_1$ resides in a cycle with length 3




Augmented node feature for $v_1$
[0, 0, 0, 0, 1, 0]
$v_1$ resides in a cycle with length 4




## Common augmented features

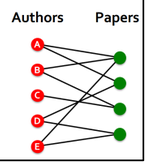
- Other commonly used augmented features:
  - **Node degree**
  - **Clustering coefficient**
  - **PageRank**
  - **Centrality**
  - **…**
- Any **feature we have introduced in Lecture 2 can be used!**

### Graph too sparse

Add virtual nodes / edges

- $A + A^2$ – 2 hops neighbour (see lec2)
- $A * A^T$ or $A^T * A$
  - Message passing
    - Authors to same paper or papers to same author
    - Rather than author to paper, paper to author
- Pros
  - The depth can be smaller –train faster
  - More neighbours – more complexity

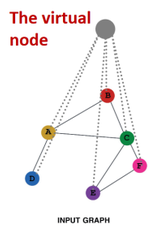
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix $A$ for GNN computation, use $A + A^2$
- **Use cases:** Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph




Add virtual nodes

- Reduce distance

- The virtual node will connect to all the nodes in the graph
  - Suppose in a sparse graph, two nodes have shortest path distance of 10
  - After adding the virtual node, **all the nodes will have a distance of two**
    - Node A – Virtual node – Node B
- **Benefits:** Greatly **improves message passing in sparse graphs**



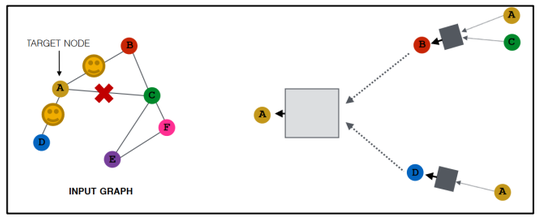

### Neighbour node sampling – large graph

Previously: A

- All the nodes are used for message passing

Now

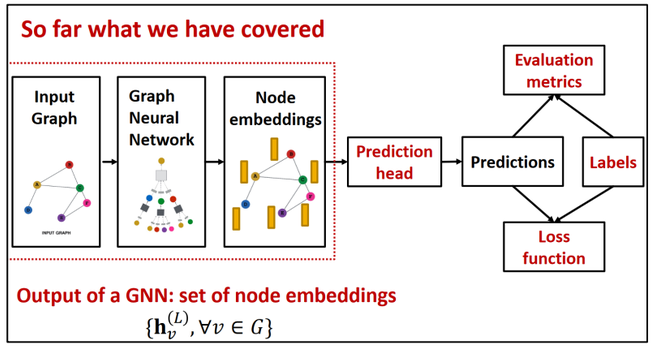
- New idea: (Randomly) sample a node's neighborhood for message passing
- Only nodes $B$ and $D$ will pass messages to $A$




- In the next layer when we compute the embeddings, we can sample different neighbors
  - **Benefits:** Greatly reduces computational cost
    - **Allows for scaling to large graphs (more about this later)**
  - And in practice it works great!

# 8.2 GNN training

### pipeline




(1) Different prediction heads:

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

# prediction

### Node level – 1 node embedding mapping

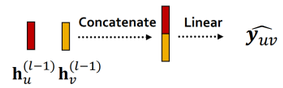
- Map embedding to label

- **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
- $\hat{\boldsymbol{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)}\mathbf{h}_v^{(L)}$
  - $\mathbf{W}^{(H)} \in \mathbb{R}^{k*d}$ : 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

### Edge level – 2 nodes

1) Concatenation + Linear

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(1) Concatenation + Linear**
  - We have seen this in graph attention




  - $\hat{\boldsymbol{y}}_{\boldsymbol{uv}} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
  - Here $\text{Linear}(\cdot)$ will map $2d$-dimensional embeddings (since we concatenated embeddings) to $k$-dim embeddings ($k$-way prediction)

2) dot product

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

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

## Graph level – all nodes

- Graph-level prediction: Make prediction **using all the node embeddings** in our graph
- Mean pooling
  - Compare graph with different sizes
- Sum pooling
  - How many nodes & structure

### Global mean / max / sum pooling

- 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?**

### Global pooling issues – same represenatation

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

### Solution – hierachy global pooling: agg first few and last separately then agg all

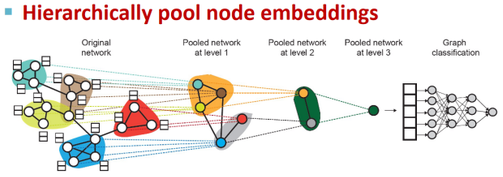
- **Toy example:** We will aggregate via $\text{ReLU}(\text{Sum}(\cdot))$
  - We first separately aggregate the first 2 nodes and last 3 nodes
  - Then we aggregate again to make the final prediction
- $G_1$ node embeddings: $\{-1, -2, 0, 1, 2\}$
  - **Round 1**: $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-1,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

**Now we can differentiate $G_1$ and $G_2$ !**

### DiffPool idea

- Graph clustering – community detection algorithm
- Can use GNN to learn such community

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




- **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**

- **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**

## Pipline 2 – loss

### Supervised task – node / edge / graph labels

- **Node labels $\boldsymbol{y_v}$:** in a citation network, which subject area does a node belong to
- **Edge labels $\boldsymbol{y_{uv}}$:** in a transaction network, whether an edge is fraudulent
- **Graph labels $\boldsymbol{y_G}$:** 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

- **The solution:** "self-supervised learning", we can find supervision signals within the graph.
  - For example, we can **let GNN predict the following:**
  - **Node-level $\boldsymbol{y_v}$.** Node statistics: such as clustering coefficient, PageRank, …
  - **Edge-level $\boldsymbol{y_{uv}}$.** Link prediction: hide the edge between two nodes, predict if there should be a link
  - **Graph-level $\boldsymbol{y_G}$.** Graph statistics: for example, predict if two graphs are isomorphic
  - **These tasks do not require any external labels!**

### Loss – Classfication & regression

### Notation

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

- **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**

## CE loss

- Force those class not belong to have 0 value
- Force the class belongs to have high values

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

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

Label: $\boldsymbol{y}^{(i)}$; Prediction: $\hat{\boldsymbol{y}}^{(i)}$; $i$-th data point; $j$-th class

where:

E.g. | 0 | 0 | 1 | 0 | 0 |

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

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K$ = prediction after Softmax(·)

E.g. | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

- Total loss over all $N$ training examples

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

## Regression loss

- Why square
  - Smooth, continuous, easy to differentiate, positive

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

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

$i$-th data point; $j$-th target

where:

E.g. | 1.4 | 2.3 | 1.0 | 0.5 | 0.6 |

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

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

E.g. | 0.9 | 2.8 | 2.0 | 0.3 | 0.8 |

- Total loss over all $N$ training examples

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

# Pipeline 4 – evaluation metric

### regression

- **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)

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

## Classification

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

$$\frac{1[\mathrm{argmax}(\hat{\boldsymbol{y}}^{(i)}) = \boldsymbol{y}^{(i)}]}{N}$$

- **(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

## Metric for binary classification

- F1 score: Harmonic mean

- **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) |

## ROC Curve

- Captures the tradeoff in TPR and FPR as the classification threshold is varied for a binary classifier
- Intuition: The probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one

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

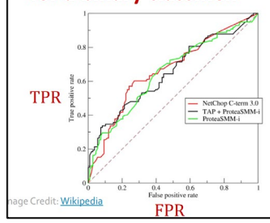



Image Credit: Wikipedia

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

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

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

# Pipeline 5

## Dataset split

- No leakage from the testing to the validation set

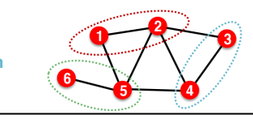
- **Fixed split:** We will split our dataset **once**
  - **Training set**: used for optimizing GNN parameters
  - **Validation set**: develop model/hyperparameters
  - **Test set**: held out until we report final performance
- **A concern:** sometimes we cannot guarantee that the test set will really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

## Different to image task

- **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**

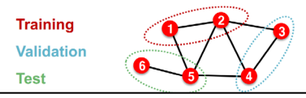

### Transductive setting – all nodes for embedding & split labels

- **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**



## Inductive setting – break edges & split graph to subgraph

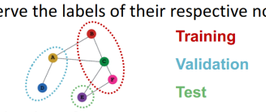
- **Solution 1 (Transductive setting): The input graph can be observed in all the dataset splits (training, validation and test set).**
- **We will only split the (node) labels**
  - **At training time,** we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At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



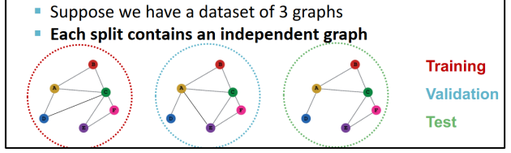

Example

node classification

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

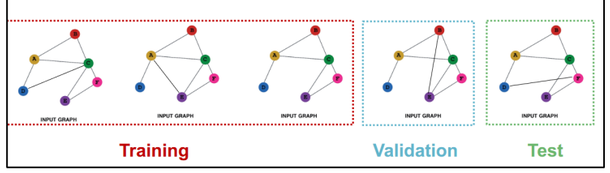



- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**




Graph classification – only inductive (need unseen test set)

- 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).

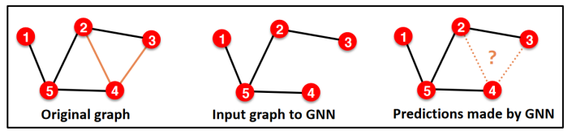



# Link prediction - unsupervised / self-supervised task

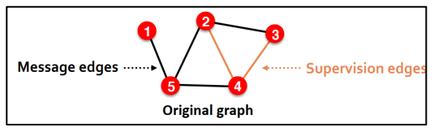
- Goal of link prediction: predict missing edges

need to

- create the labels and dataset splits on our own
- hide some edges from the GNN and the let the GNN predict if the edges exist




### Step 1: Message (passing) / Supervision (objectives) edges

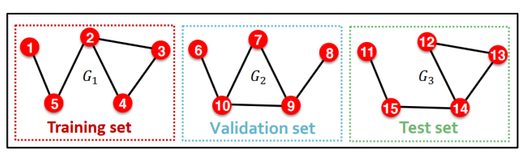



- **For link prediction, we will split edges twice**
- **Step 1: Assign 2 types of edges in the original graph**
  - **Message edges: Used for GNN message passing**
  - **Supervision edges: Use for computing objectives**
  - **After step 1:**
    - **Only message edges will remain in the graph**
   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

### 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

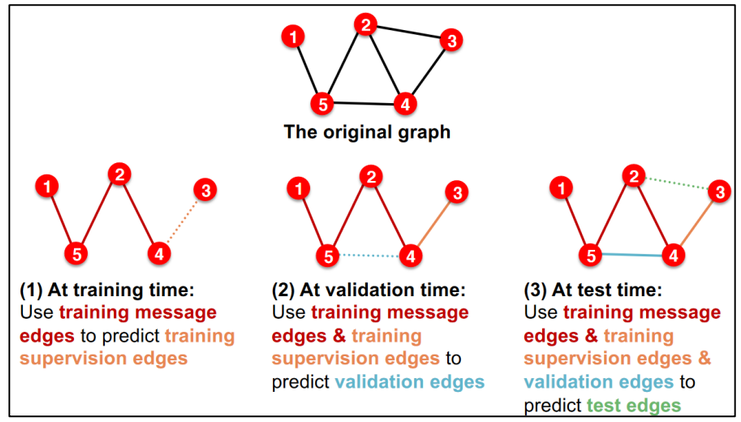



Option 2: Transductive link prediction split

- **default setting** when people talk about link prediction
- Suppose we have a dataset of 1 graph
- the entire graph can be observed in all dataset splits

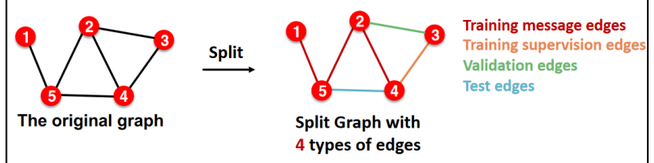
- **But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges**
- **To train the training set, we further need to hold out supervision edges for the training set**

Training, validation, testing time




4 types of links

- **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**

# Summary

**Transductive setting.** Generalise to new nodes
**Inductive setting.** No new nodes

Similar to NLP, GNN also has some hard feature engineering (thoes that are hard to captured by the network) :
Add node features:

1. Constant
2. One hot
3. Structure info
   - node degree
   - centrality

add sturture features:

neighbour samlping

- Feels like dropouts in aggregating neighbour info

Diffpool hierachy pooling

**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 – reuse existing methods
- Loss
  - Rank loss
  - Classifcation
  - Regression

**Transductive vs Inductive setting**

| | Transductive | Inductive |
|---|---|---|
| training / validation / test sets | same graph | different graphs |
| Dataset consists of | one graph | multiple graphs |
| observe | - entire graph can be observed in all dataset splits<br>- only split the labels | Each split can only observe the graph(s) within the split |
| Generalisation | | generalize to unseen graphs |
| Task | node / edge | node / edge / graph |
| Cons | Leakage in the structure information from the test set | Remove some edge info |

**Implementation resources:**

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

- **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

**General tips**

- **Data preprocessing is important:**
  - Node attributes can vary a lot!
    - E.g. probability ranges $(0,1)$, but some inputs could have much larger range, say $(-1000, 1000)$
  - Use normalization
- **Optimizer:**
  - ADAM is relatively robust to learning rate
- **Activation function**
  - ReLU activation function often works well
  - Other alternatives: LeakyReLU, SWISH, rational activation
  - No activation function at your output layer:
- **Include bias term in every layer**
- **Embedding dimensions:**
  - 32, 64 and 128 are often good starting points

**More reading**

**Tutorials and overviews:**
- Relational inductive biases and graph networks (Battaglia et al., 2018)
- Representation learning on graphs: Methods and applications (Hamilton et al., 2017)

**Attention-based neighborhood aggregation:**
- Graph attention networks (Hoshen, 2017; Velickovic et al., 2018; Liu et al., 2018)

**Embedding entire graphs:**
- Graph neural nets with edge embeddings (Battaglia et al., 2016; Gilmer et. al., 2017)
- Embedding entire graphs (Duvenaud et al., 2015; Dai et al., 2016; Li et al., 2018) and graph pooling (Ying et al., 2018, Zhang et al., 2018)
- Graph generation and relational inference (You et al., 2018; Kipf et al., 2018)
- How powerful are graph neural networks(Xu et al., 2017)

**Embedding nodes:**
- Varying neighborhood: Jumping knowledge networks (Xu et al., 2018), GeniePath (Liu et al., 2018)
- Position-aware GNN (You et al. 2019)

**Spectral approaches to graph neural networks:**
- Spectral graph CNN & ChebNet (Bruna et al., 2015; Defferrard et al., 2016)
- Geometric deep learning (Bronstein et al., 2017; Monti et al., 2017)

**Other GNN techniques:**
- Pre-training Graph Neural Networks (Hu et al., 2019)
- GNNExplainer: Generating Explanations for Graph Neural Networks (Ying et al., 2019)