CS 224w: Machine Learning with Graphs

Leni Aniva

Autumn 2024

- Instructor: Jure Leskovec
- Website: `http://cs244w.stanford.edu`
- Note: A page with a dark background indicates it is from Winter 2023.

Super! – Dr. Jure Leskovec

Contents

Introduction

0.1 Why Graphs?

Graphs are a general language for describing and analyzing entities with relations and interactions.
Applications:

- Molecules: Vertices are atoms and edges are bonds
- Event Graphs
- Computer Networks
- Disease Pathways
- Code Graphs

Complex domains have a rich relational structure which can be represented as a relational graph. By explicitly modeling relationships we achieve a better performance with lower modeling capacity.
The Modern ML Toolbox processes tensors, e.g. images (2D), text/speech (1D). Modern deep learning toolbox is designed for simple sequences and grids. Not everything can be represented as a sequence of a grid. How can we develop neural networks that are much more broadly applicable? We can use graphs. Graphs *connect* things.

- Graph neural network is the 3rd most popular keyword in ICLR '22.
- Graph learning is also very difficult due to the complex and less structured nature of graphs.
- Graph learning is also associated with representation learning. In some cases it may be possible to learn a d-dimensional embedding for each node in the graph such that similar nodes have closer embeddings.

A number of different tasks can be executed on graph data.

- **Node level prediction**: to characterize the structure and position of a node in the network.

  Example: In Protein folding, each atom is a node and the task is to predict the coordinate of the node. predicted.

- **Edge/Link level prediction**: Predicting property for a pair of nodes. This can be either trying to find missing links or finding new links as time progresses

  Example: Graph-based recommender systems and drug side effects.

- **Graph level prediction**: Predict for an entire subgraph or graph

  Examples: traffic prediction, drug discovery, physics simulation, and weather forecasting

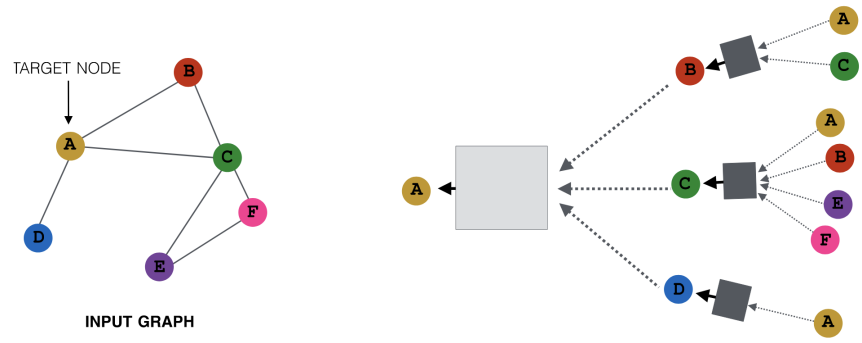


Figure 0.1: When a machine learning model is applied to a graph, each node defines its own computational graph in its neighbourhood.

0.2 Choice of Graph Embeddings

A graph has several components:

- **Objects** N: Nodes, vertices
- **Interactions** E: Edges, links
- **Systems** $G(N, E)$: Networks, graphs

Sometimes there is a ubiquitous representation in a particular case. Some times there is not. The choice of representation determines what information can be mined from the graph. A graph may also have some other properties:

- Undirected/Directed edges
- Allow/Disallow self-loop
- Allow/Disallow multi-graphs (mutliple edges between nodes)
- **Heterogeneous Graphs**: A graph $G = (V, E, R, T)$ where edges have relation types $(v_i, r, v_j) \in E$ and nodes have types $T(v_i)$ with relation type $r \in R$.

  Many graphs are heterogeneous. For example, drug-protein interaction graph is heterogeneous.
- **Bipartite Graphs**: e.g. Author-Paper graph, Actor-Films graph

Most real-world networks are sparse. The adjacency matrix is a sparse matrix with mostly 0's. The density of the matrix (E/N^2) is $1.51 \cdot 10^{-5}$ for the WWW and $2.27 \cdot 10^{-8}$ for MSN IM.

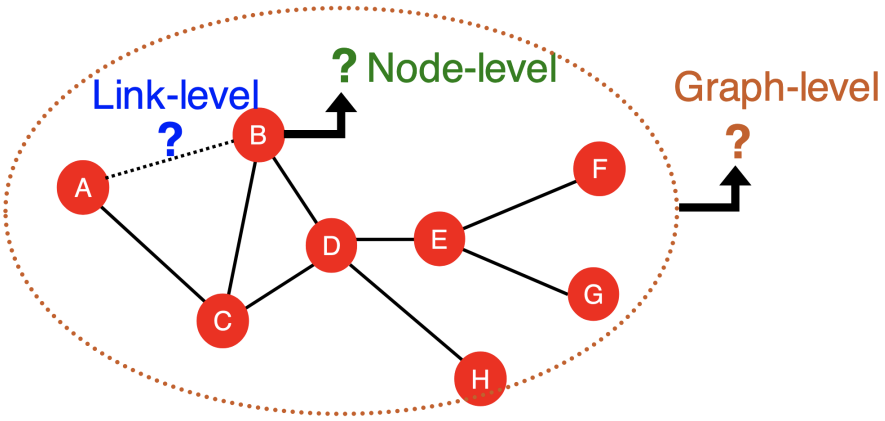


Figure 1.1: Different levels of tasks on a graph

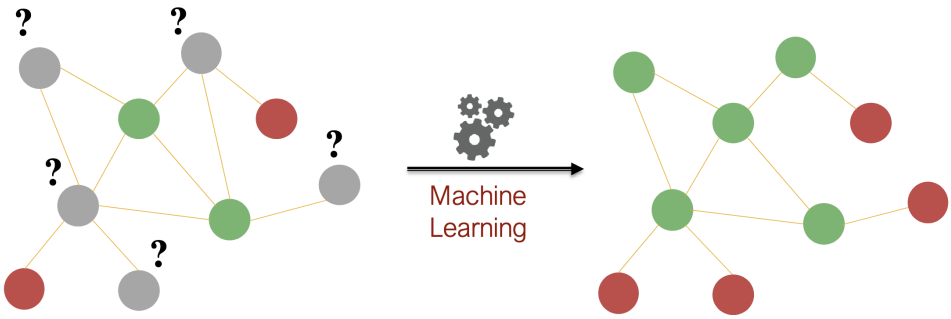


Figure 1.2: Classifying nodes on a graph when a few labels are provided

1 Traditional Machine Learning on Graphs

In a traditional ML pipeline such as Logistic regression, Random Forest, and Neural networks, the model is first trained on features of a graph and then the model can be applied on a new graph. *Using effective features over graphs is the key to achieving good model performance.* For simplicity, in this section we focus on un-directed graphs.

1.1 Node-Level Features

A simple example of node-level task is node classification based on a few samples. It is illustrated in Figure 1.2
A couple different measures characterise the structure and position of a node on a graph:

- **Node degree**: Number of neighbours
- **Node centralities**: A measure of the "importance" of a node in a graph. There several different types of centrality.
  - **Eigenvector centrality**: A node is important if it is surrounded by important neighbouring nodes. We define the centrality of node v as the centrality of

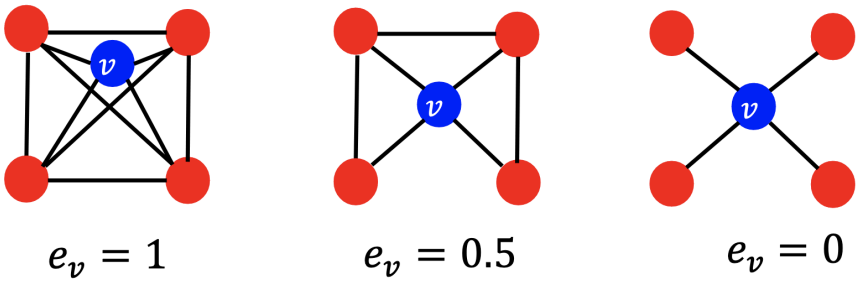


Figure 1.3: Examples of clustering coefficients

neighbouring nodes. This leads to a set of $|N|$ simultaneous linear equations:

$$c_v := \frac{1}{\lambda} \sum_{u \in N(v)} c_u \iff \lambda \boldsymbol{c} = \boldsymbol{A}\boldsymbol{c}$$

where λ is a normalisation constant and $\boldsymbol{A}$ is the adjacency matrix of the graph.
By Perron-Forbenius Theorem, the largest eigenvalue $\lambda_{\max}$ is always positive and unique, and its corresponding eigenvectors can be used for centrality.
When λ is the second-largest eigenvalue, c_v has a different meaning.

- **Betweenness centrality**: A node is important if it is a gatekeeper. i.e. when it lies on many shortest paths between other nodes.

$$c_v := \sum_{s \neq v \neq t} \frac{|\{\text{shortest paths between } s, t \text{ containing } v\}|}{|\{\text{shortest paths between } s, t\}|}$$

This is important for social networks.

- **Closeness centrality**:

$$c_v := \sum_{u \neq jv} \frac{1}{|\{\text{shortest paths between } u, v\}|}$$

- **Clustering Coëfficients**: How connected v's neighbouring nodes are

$$e_v := \frac{1}{\binom{k_v}{2}} |\{\text{edges among } N(v)\}| \in [0, 1]$$

Social networks have a huge amount of clustering.

Clustering coëfficient counts the number of triangles in the **ego-network** (the network formed by $\{v\} \cup N(v)$, where v is the *ego*). We can generalise the above by counting graphlets.
An **induced graph** is a graph formed by taking a subset of vertices in a larger graph such that only edges connecting the remaining vertices are preserved.
Two graphs with identical topologies are **isomorphic**.

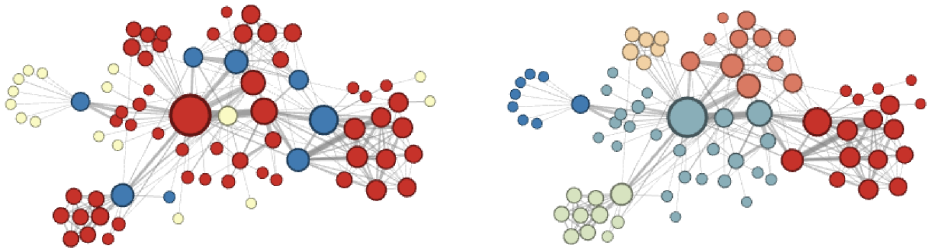


Figure 1.4: Different levels of features distinguish nodes in different ways

Graphlets are small subgraphs that describe the structure of u's neighbourhood network. Specifically, they are *rooted*, *connected*, *induced*, *non-isomorphic* subgraphs. Considering graphs of size 2 to 5 nodes we get a vector of 73 (number of graphlets with 2 to 5 vertices) elements that describes the topology of a node's neighbourhood. This vector is the **graphlet degree vector (GDV)** of a node.
The features we have discussed so far capture local topological properties of the graph but cannot distinguish points in a global scale.

1.2 Link-Level Features

Two formulations of link-level prediction

- Links missing at random: Remove a random set of links and then aim to predict them.

  e.g. drug interactions

- Links over time: Given $G[t_0, t_0']$ a graph defined by edges up to time t_0' output a rankd list L of edges that are predicted to appear in time $G[t_1, t_1']$.

  Evaluation: $n = |E_{\text{new}}|$: number of new edges that appear during the test period.

  Methodology: For each pair of nodes x, y compute a distance $c(x, y)$ and predict top n elements as links.

A couple measures exist for *local neighbourhood overlap*:

- **Common neighbours**:
$$c(u, v) := |N(v_1) \cap N(v_2)|$$
- **Jaccard's Coëfficient**:
$$c(u, v) := \frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$$
- **Adamic-Adar Index**:
$$c(u, v) := \sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{k_u}$$
  The problem with the three indices above is that they are always 0 if u, v do not share a neighbour.

- **Katz Index**:

$$c(u, v) := |\{\text{All walks of all lengths between } u, v\}|$$

This can be computed by powers of the adjacency matrix. The matrix counting all walks of length n between vertices is $\boldsymbol{A}^n$, so the Katz index can be computed by

$$\boldsymbol{C} := \sum_{i=1}^{\infty} \beta^i \boldsymbol{A}^i = (\boldsymbol{I} - \beta \boldsymbol{A})^{-1} - \boldsymbol{I}$$

where the $\beta < 1$ decay factor is necessary to prevent $\boldsymbol{C}$ from blowing up to $+\inf$.

An analogous definition exists for directed graphs.

1.3 Graph Kernels

The goal of graph kernels is to create a feature vector for the entire graph.
Kernel methods are widely used for traditional ML for graph-level prediction. Instead of designing feature vectors, we design kernels:

- $k(G, G') \in \mathbb{R}$
- Kernel matrix $\boldsymbol{K} = [K(G, G')]_{G,G'}$ must always be positive semi-definite.
- There exists a feature representation ϕ such that $K(G, G') = \phi(G)\phi(G')$ which can even be infinite-dimensional.

We could use a *bag-of-words (BOW)* for a graph. Recall that in NLP, BoW simply uses the word count as features for documents with no ordering being considered. We regard nodes as a word.
Graph-Level Graphlet features counts the number of different graphlets in a graph. The graphlets here are not rooted and do not have to be connected. This definition of graphlet is slightly different from the definition of node level features. A limitation of this definition is that counting graphlets is expensive. Counting size-k gfor a raph with size n by enumeration takes time $\mathrm{O}(n^k)$ due to costly subgraph isomorphism tests. If a graph's node has bounded degree the time could be compressed down to $\mathrm{O}(nd^{k-1})$.
so far we have only considered features related to the graph structure and not considered the attributes of nodes and their neighbours.

2 Node Embeddings

Representation learning avoids the need of doing feature engineering every time. The goal is to map individual nodes or an entire graph into vectors, or embeddings.
In node embeddings, we would like the embedding to have the following properties:

- Similarity of embeddings between nodes indicates their similarity in the network.

  e.g. Nodes closer together could be considered similar.

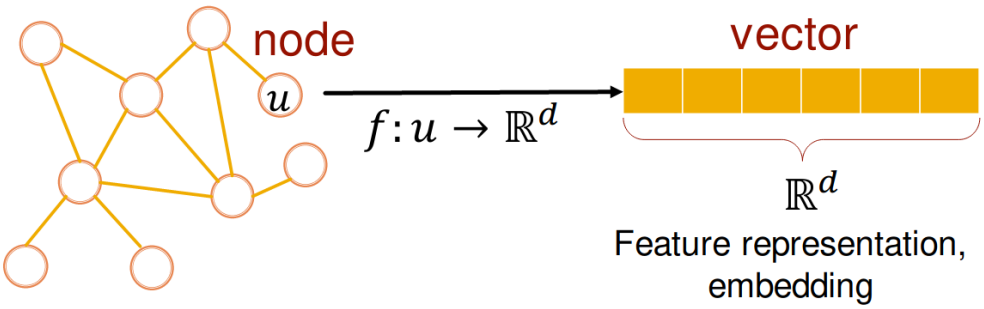


Figure 2.1: Node Level Embeddings of graphs map each node to a vector

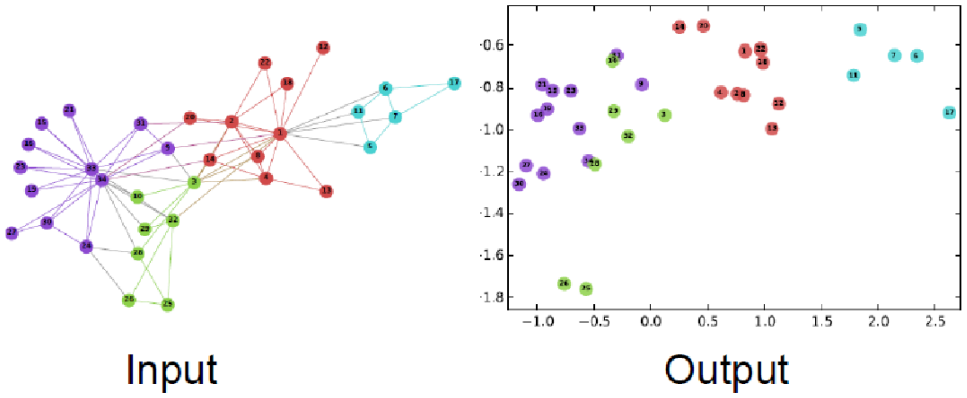


Figure 2.2: Embeddings of Karate Club Graph

- Encode network information
- Potentially useful for downstream tasks

Suppose we have a graph G and we wish to devise an embedding for the nodes V of G. The goal is to *encode* nodes so that similarity in the embedding space approximates similarity in the graph. We have 3 design choices, the encoder, decoder, and a target similarity function

- **Encoder**: $\mathsf{Enc} : V \to \mathbb{R}^{d \times |V|}, \mathsf{Enc}(u) := \boldsymbol{z}_u$

  This could be defined to be a simple lookup table from nodes to vectors. In this case it is a *shallow encoder*. Deep encoders using GNN will be covered in Subsection 5.1. This is the only learnable function.

- **Decoder**: Given two embeddings, measure the similarity. Usually chosen to be the dot product $\mathsf{Dec}(\boldsymbol{z}, \boldsymbol{w}) = \boldsymbol{z} \cdot \boldsymbol{w}$

- **Similarity**: A measure of similarity of the nodes. This could be distance in the graph, neighbourhood topology, etc. It is approximated by a combination of encoder and decoder. $\text{similarity}(u, v) \simeq \mathsf{Dec}(\mathsf{Enc}(u), \mathsf{Enc}(v))$

  In this example, it is $\text{similarity}(u, v) \simeq \boldsymbol{z}_u \cdot \boldsymbol{z}_v$

Many methods stem from this simple setting, including DeepWalk and node2vec.

2.1 Random Walk Embedding

An unsupervised/self-supervised node similarity metric is based on *random walks*. In which case we are not using node labels or features. The embeddings are task independent and do not use any node labels or features. The rationale of choosing random walks:

- *Expressivity*: Flexible stochastic definition of node similarity that incorporate lower and higher order neighbourhood information.
- *Efficiency*: The graph does not need to be throughoutly traversed when training.

We define:

- Vector $\mathsf{Enc}(u) := \boldsymbol{z}_u$: Embedding of node u
- Probability $P(v|\boldsymbol{z}_u)$: Predicted probability of visiting v on random walks starting from u.
- **Softmax Function**: Turn vector of k values into k probabilities that sum to 1:

$$\sigma(\boldsymbol{z}) = \frac{\exp \boldsymbol{z}}{\sum_{i=1}^{k} \exp \boldsymbol{z}}$$

- **Sigmoid Function**: Compress $\mathbb{R}$ into $(0, 1)$:

$$S(x) = \frac{1}{1 + \exp(-x)}$$

- A **random walk** is a sequence $v_0, v_1, \ldots$ of random vertices such that v_{i+1} is chosen randomly from $N(v_i)$.
- The target similarity function is

$$\text{similarity}(u, v) := P(u, v \text{ occur on a random walk over } G)$$

- We select a random walk strategy R and use such strategy to determine $P_R(v|u)$, the probability that a random walk starting from u visits v. The strategy defines $N_R(u)$, a random multiset (can have repeats for nodes visited multiple times, essentially a probability distribution) collected from combining all *short fixed-length* random walks starting at u

Now we are ready to mathematically state the objective function. The negative log-likelihood

$$L(\mathsf{Enc}) := -\sum_{u \in V} \log P(N_R(u)|\boldsymbol{z}_u) = -\sum_{u \in V} \sum_{v \in N_R(u)} \log P(v|\boldsymbol{z}_u)$$

Predicted probability of u, v co-occurring

Summation over all vertices

Summation over nodes seen on random walk from u

We parameterise $P(v|\boldsymbol{z}_u)$ using softmax. Essentially, we view the exponentiated similarity scores $\exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_v)$ as unnormalised probabilities:

$$P(v|\boldsymbol{z}_u) := \frac{\exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_v)}{\sum_{n \in V} \exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_n)}$$

However, this function is expensive to evaluate. The two $\sum_{n \in V}$ loops already give $\mathrm{O}(|V|^2)$ time complexity. The solution to this problem is **negative sampling**[1], which provides the estimate

$$\log \frac{\exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_v)}{\sum_{n \in V} \exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_n)} \simeq \log \sigma(\boldsymbol{z}_u \cdot \boldsymbol{z}_v) - \sum_{i=1}^{k} \log \sigma(\boldsymbol{z}_u \cdot \boldsymbol{z}_n) \qquad (n_i \sim P_v)$$

where P_i is a probability distribution over V. Instead of normalising w.r.t. all nodes, just normalise against k random *negative samples* n_i. We could select P_V such that $P_V(n) \propto \deg n$. The value of k is usually chosen to be 5 to 20 since

- Higher k gives more robust estimates
- Higher k corresponds to higher bias on negative events.

With this in mind, we can robustly evaluate $\frac{\partial L}{\partial \boldsymbol{z}_u}$ and execute stochastic gradient descent (SGD) to optimise Enc.

[1] This is a form of Noise Contrastive Estimation (NCE). See `https://arxiv.org/pdf/1402.3722.pdf`

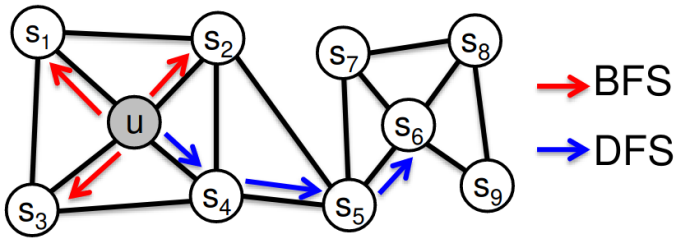


Figure 2.3: Comparison of neighbourhood N_R generated by BFS and DFS strategies. BFS provides a micro-view of the neighbourhood while DFS provides a macro-view.

Excursion: Stochastic Gradient Descent

A recap of the stochastic gradient descent algorithm in the setting of random walk embedding:

1. Initialise $\boldsymbol{z}_u$ at random for all u
2. For all u:
   - Compute $\partial L / \partial \boldsymbol{z}_u$
   - Make a step $\boldsymbol{z}_u \leftarrow \boldsymbol{z}_u - \eta \cdot \partial L / \partial \boldsymbol{z}_u$. η is the *learning rate*.
3. Return to step (2) until convergence.

A couple different options are in order for the random walk strategy R. In **DeepWalk**, this is just an unbiased random walk starting from each node but this could be too constrained. In **node2vec**, the strategy is chosen to embed nodes with similar network neighbourhood close in the feature space. We frame this as a maximum-likelihood optimisation problem which is independent to the downstream prediction task. The key observation is that flexible notion of N_R lead to rich node embeddings.
Two classic strategies define a neighbourhood N_R: Breadth First Search and Depth First Search. We could interpolate BFS and DFS using two parameters:

- Return parameter p: Return back to previous node.
- In-out parameter q: Moving outwards (DFS) vs. inwards (BFS). Intuitively q is the interpolation parameter.

We define a biased *2nd-order* random walk to explore network neighbourhoods. Suppose we just traveled the edge (s_1, w) and is now at w. Three choices are ahead in $N(w)$.

- Return to s_1 (distance 0) with weight $1/p$
- Stay in $N(s_1)$ (distance 1) with weight 1 *for each nodes in* N.
- Leave $N(s_1)$ and explore further (distance 2) with weight $1/q$ *for each node further out.*

Figure 2.4: Biased 2nd order neighbourhoods along with unnormalised probabilities.

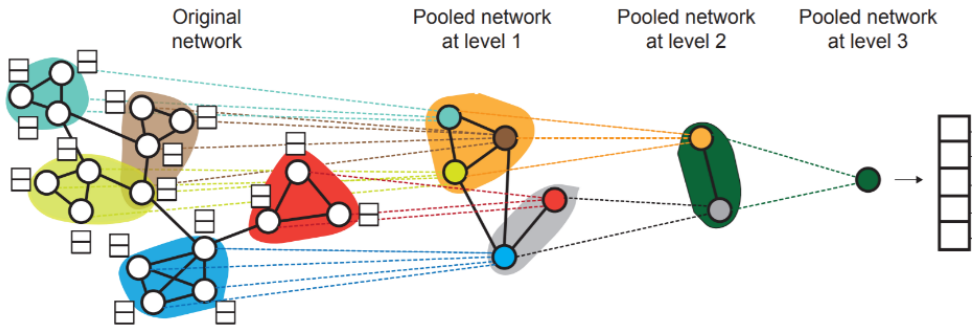


Figure 2.5: DiffPool: Hierarchical Embedding

A BFS-like walk has low value of p (easy to backtrace) and a DFS-like walk has a low value of q.
In a survery in 2017, node2vec performs better on node classification tasks while alternative methods perform better on link prediction. No one method wins all cases. Random walk approaches are generally more efficient.

2.2 Embedding Entire Graphs

We sometimes wish to embed an entire graph in some embedding space. This could be useful to e.g. classifying toxic/non-toxic molecules or identifying anomalous graphs. A simple approach is to sum over all node embeddings on an existing node embedding:

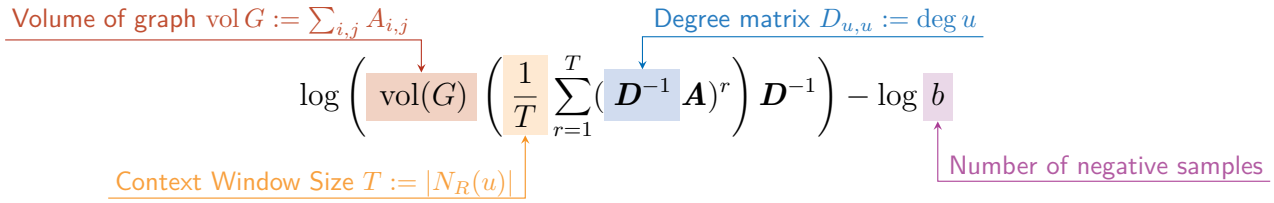
$$\boldsymbol{z}_G := \sum_{v \in G} \boldsymbol{z}_v$$

Another approach is to introduce a "virtual node" to represent the subgraph and run a node embedding algorithm to use its embedding.
We will discuss *hierarchical embeddings*, which successively summarises the graph in smaller clusters to generate an embedding.

2.3 Relations to Matrix Factorization

Recall that the encoder can be viewed as an embedding lookup into a matrix $\boldsymbol{Z}$ of size $d \times |V|$.
The most simple node similarity is the adjacency matrix. Two nodes can be considered similar when they are connected by an edge and not otherwise. This is the adjacency

matrix $\boldsymbol{A}$ and an attempt to learn the embedding is a matrix factorisation $\boldsymbol{A} = \boldsymbol{Z}^\intercal \boldsymbol{Z}$.
Since $d \ll |V|$, this factorisation cannot be done exactly due to rank, so we may instead optimise to minimise the Frobenius norm $\min_{\boldsymbol{Z}} \|\boldsymbol{A} - \boldsymbol{Z}^\intercal \boldsymbol{Z}\|_{\mathsf{F}}$.
In the example of DeepWalk, the embedding factorises the matrix



Essentially $\boldsymbol{D}^{-1}\boldsymbol{A}$ is a Markov matrix of the random walk.

2.4 Applications and Limitations

Embeddings can be used for

- Clustering/Community detection
- Node classification
- Link prediction: Predict edge (i, j) based on $(\boldsymbol{z}_i, \boldsymbol{z}_j)$
- Graph classification: Classification of $\boldsymbol{z}_G$

Node embeddings via matrix factorisation and random walks have some limitations:

- $O(|V|\, d)$ parameters are needed.

  No sharing of parameters is possible between nodes and each node has its own unique embedding.
- **Transductivity**: The embedding can only be generated after all nodes in the graph are seen. Cannot obtain embeddings for nodes not in the training set.
- Cannot capture structural similarity of local topologies.
- Cannot utilise node, edge, and graph features.

Deep Representation Learning and Graph Neural Networks mitigate these limitations, which will be covered in depth in Section 4 and Section 5.

3 Graph Neural Networks

Recall from Lecture 2 that node embeddings map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together. To learn such an embedding, we need to define encoder Enc, decoder Dec, and a target similarity function similarity(u, v).
Today we discuss a class of deep learning methods based on Graph Neural Networks. We use a *Deep Graph Encoder* as Enc. The modern machine learning toolbox is based off regular, repeating lattice or grids, which cannot be easily adapted to graphs since the structure of a graph is far more complex than a rectangular grid. There is no fixed node ordering or reference point and graphs are often dynamic.

3.1 Basics of Deep Learning

This is a brief introduction to deep learning.

Excursion: Supervised Learning

In **supervised learning**, we are given inputs x and our goal is to predict output y. The input x can be vectors, sequences, matrices, graphs. We formula the task as an optimisation problem.

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

where

- θ is a set of **parameters** we optimise. e.g. in shallow encoder, $\theta = \{Z\}$.
- L is the **loss function**.

  Common choices of L:

  - L^2**-loss**: $L(\boldsymbol{y}, \hat{\boldsymbol{y}}) := \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|$
  - **Cross Entropy**: $L(\boldsymbol{y}, \hat{\boldsymbol{y}}) := -\sum_{i=1}^{C} y_i \log \hat{y}_i$
    In this case $\boldsymbol{y} \in \{0, 1\}^C$ is a *one-hot* encoding of the ground truth category, while $\hat{\boldsymbol{y}} \in [0, 1]^C$ is a probability vector $(\sum \hat{\boldsymbol{y}} = 1)$
- The optimisation problem is solved using gradients.
  - The gradient $\nabla_\theta L$ is a directional derivative in the direction of largest increase.
  - An iterative algorithm which updates $\theta \leftarrow \theta - \eta \nabla_\theta L$ moves θ in the opposite direction of $\nabla_\theta L$ until convergence.
  - η is the **learning rate**.
  - Evaluating the gradient w.r.t. the entire training set could be expensive, so often $\nabla_\theta L = \sum_{i=1}^{n} \nabla_\theta L(y_i, f(x_i; \theta))$ is approximated by a random sample $\sum_{i \in I} \nabla_\theta L(y_i, f(x_i; \theta))$ where $I \subseteq \{1, \ldots, n\}$. This is

stochastic gradient descent and I is the **batch**. $|I|$ is the **batch size** and the number of full passes through the dataset is the **epoch**.

- Other higher order optimisers exist such as RMSprop, Adam, Adagrad, etc.

- A **multi-layer perceptron** (**MLP**) is a neural network constructed from stacking layers of the form $f_i(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{W}, \boldsymbol{b}$ are learnable. σ is a non-linear function referred to as **activation function**.

3.2 Deep Learning for Graphs

Settings

Suppose we have a graph G with vertex set V, adjacency matrix $\boldsymbol{A} \in \{0,1\}^{|V|\times|V|}$, node features $\boldsymbol{X} \in \mathbb{R}^{|V|\times d}$.

A naïve approach would be to join the adjacency matrix and features and feed them into a deep neural net. The problems with this idea are

- O($|V|$) parameters
- Graph size is inherently baked into the size of the neuural network
- Sensitive to node ordering

One solution is to use **convolutional networks**, which use a sliding **kernel** which is invariant across all points on the graph. There is no fixed notion of locality or sliding window on a graph, and graphs do not give an inherent order to their vertices.
Consider we learn a function f that maps a graph $G := (\boldsymbol{A}, \boldsymbol{X})$ to a vector in $\mathbb{R}^d$. Then we would like $f(\boldsymbol{A}_1, \boldsymbol{X}_1) = f(\boldsymbol{A}_2, \boldsymbol{X}_2)$ for two different orderings of the vertices of G. For any *graph* function $f : \mathbb{R}^{|V|\times m} \times \mathbb{R}^{|V|\times|V|} \to \mathbb{R}^d$,

- f is **permutation invariant** if $f(\boldsymbol{A}, \boldsymbol{X}) = f(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\intercal, \boldsymbol{P}\boldsymbol{X})$ for any permutation matrix $\boldsymbol{P}$.
- f is **permutation equivariant** if $\boldsymbol{P}f(\boldsymbol{A}, \boldsymbol{X}) = f(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\intercal, \boldsymbol{P}\boldsymbol{X})$ for any permutation matrix $\boldsymbol{P}$.

e.g.

- $f(\boldsymbol{A}, \boldsymbol{X}) = \boldsymbol{1}^\intercal\boldsymbol{X}$ is permutation invariant (summing of all node features).
- $f(\boldsymbol{A}, \boldsymbol{X}) = \boldsymbol{X}$ is permutation equivariant.
- $f(\boldsymbol{A}, \boldsymbol{X}) = \boldsymbol{A}\boldsymbol{X}$ is permutation equivariant.

Graph neural networks consist of multiple permutation equivariant/invariant layers. Other neural network architectures e.g. MLP, do not exhibit this property.

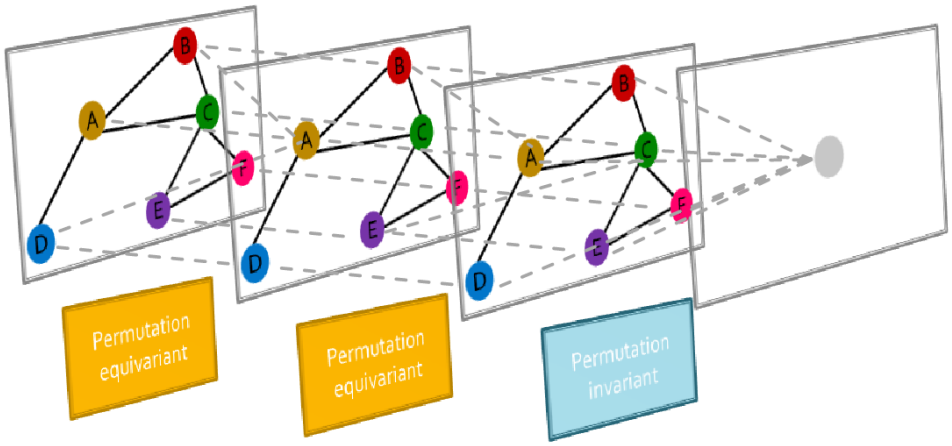
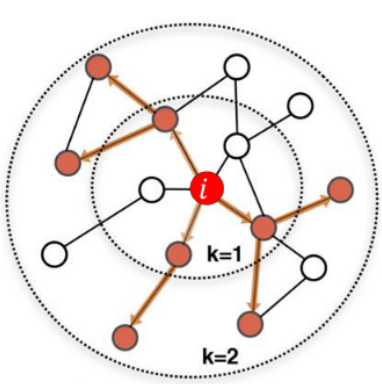
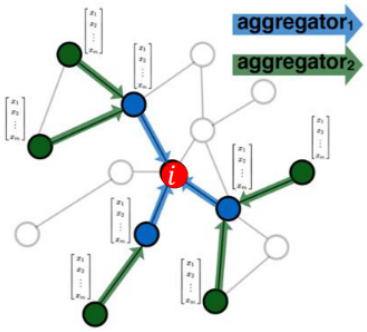


Figure 3.1: General structure of GNN which consists of permutation equivariant *convolutional* layers and permutation invariant *pooling* layers.



(a) Neighbour computation graph

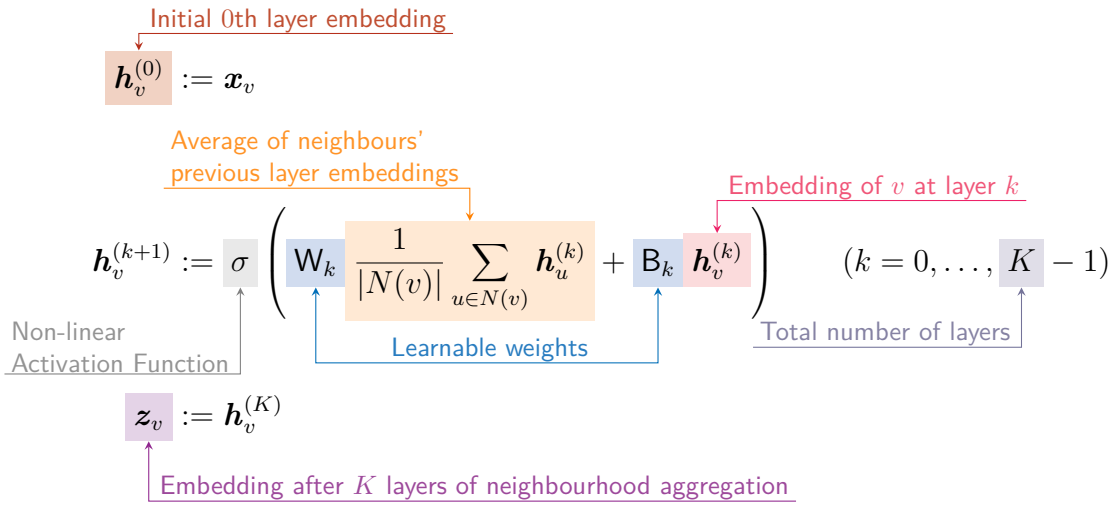


(b) Propagate and transform information

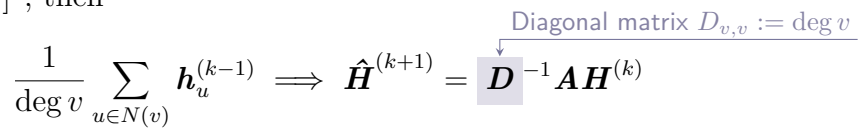
Figure 3.2: Computation graph defined from a node's neighburhood. Each node defines a computation graph based on its neighbourhood and this can change from node to node.

3.3 Graph Convolutional Networks

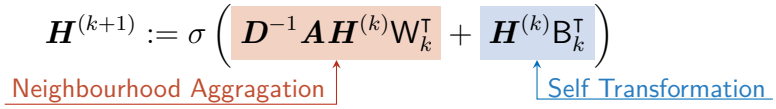
A critical observation is that a node's neighbourhood defines a computation graph. Information can be passed along this computation graph to combine information from different parts of the graph. A model constructed in this fashion can be of arbitrary depth. Layer-0 embeddings of node v is its input feature $\boldsymbol{x}_v$ and the layer-k embeddings are constructed from layer-$(k-1)$ embeddings. **Neighbourhood aggregation** is the approach to aggregate information across layers. A basic approach is to average information from neighbourds and apply a neural network. This leads to the **Deep Encoder**:



GNN computation is permutation equivariant. A GNN can be trained using SGD by updating the weight matrices $\boldsymbol{W}_k, \boldsymbol{B}_k$, for aggregation and transformation, respectively. Many aggregation can be performed efficiently as sparse matrix operations. If $\boldsymbol{H}^{(k)} := [\boldsymbol{h}_1^{(k)}, \ldots, \boldsymbol{h}_{|V|}^{(k)}]^\intercal$, then



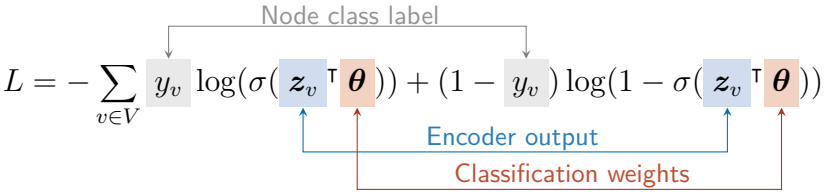
We can re-write the update function in matrix form[2]:



To train a GNN:

- Supervised setting: We could minimise the loss function of node label y_v with the node embedding $\mathsf{f}(\boldsymbol{z}_v)$ by minimising $\min_\theta L(y_v, \mathsf{f}(\boldsymbol{z}_v))$.

  For example, the training of predicting a binary label on the nodes can entail a *binary cross-entropy* or *logistic* loss function of the form

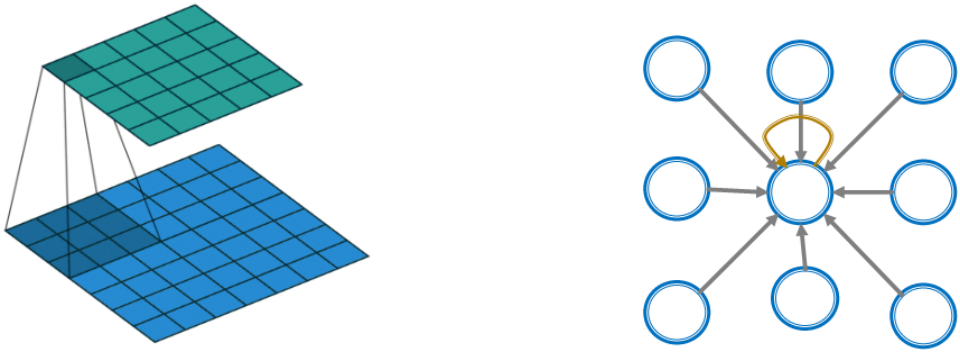


- Unsupervised setting: When no node labels are available we could use the graph's structure as supervision. By requiring similar nodes have similar embeddings. i.e. we optimise

$$L = \sum_{u,v} \mathrm{CE}(y_{u,v}, \mathsf{Dec}(\boldsymbol{z}_u, \boldsymbol{z}_v))$$

  where $y_{u,v}$ is a similarity score of the nodes.

[2] not all GNNs can be expressed in matrix form, especially when aggregation functions are complex.



(a) Image (b) Graph

Figure 3.3: Image and graph comparison of CNN vs GNN

Overall:

1. Define a neighbourhood aggregation function
2. Define a loss function on the embeddings
3. Train on a set of nodes, i.e. a batch of computation graphs.
4. Generate embeddings of nodes (even for nodes that the model never trained on)

A GNN is *inductive* as opposed to transductive. The same mode generalises to unseen nodes.

3.4 GNNs subsume CNNs

GNNs can be viewed as a class of more general architectures compared to CNNs. In a convolutional layer with 3×3 filter, it can be formulated as

$$h_v^{(l+1)} := \sigma\left(\sum_{u \in N(v) \cup \{v\}} \mathsf{w}_{l,u} h_u^{(l)}\right) \qquad (l = 0, \ldots, L-1)$$

$N(v)$ is the 8 neighbour pixels of v.

The key difference is we can learn different $W_{l,u}$ for different "neighbour" u for pixel v on the image. To do this we can pick a particular order of relative positions w.r.t. the centre pixel v.
CNNs can be seen as special GNNs with fixed neighbour size and ordering. CNN is not permutation invariant or equivariant.

4 A General Perspective on Graph Neural Networks

In this section we expand upon the previous construction of GNNs and create a general GNN framework. A GNN consists of 5 parts:

1. Message
2. Aggregation Function

   A GNN Layer is composed of the message and aggregation. Different implementations include Graph Convolutional Networks (GCN), GraphSAGE, and GAT (Graph Attention).
3. Layer Connectivity

   Layer Connectivity refers to the topological connection between layers including skip connections.
4. Graph Augmentation: Feature augmentation, structure augmentation, etc.

   The core idea is that the raw input graph should not be directly used at the computational graph for a number of problems we shall explain later.
5. Learning Objective: Supervised/Unsupervised, Node/Edge/Graph level objectives.

4.1 A Single Layer of GNN

A GNN Layer compresses a (variable sized) set of vectors into a single vector. This involves taking inputs $\boldsymbol{h}_v^{(l-1)}, \boldsymbol{h}_{u \in N(v)}^{(l-1)}$ and outputting node embedding $\boldsymbol{h}_v^{(l)}$.

- The **message function** converts the hidden state of each node into a *message*, which will be sent to other nodes later.

  $$\boldsymbol{m}_u^{(l)} := \text{Msg}^{(l)}(\boldsymbol{h}_u^{(l-1)})$$

  An example is a linear layer

  $$\boldsymbol{m}_u^{(l)} := \mathsf{W}^{(l)} \boldsymbol{h}_u^{(l-1)}$$

  Question: Can a node send different messages to different neighbours?

  Yes. We shall see an example of this, which is *Graph Attention Networks*.

- The **aggregation** function defines how the node's neighbours' messages are combined

  $$\boldsymbol{h}_v^{(l)} := \text{Agg}(\{\boldsymbol{m}_u^{(l)} : u \in N(v)\})$$

  Example: $\boldsymbol{h}_v^{(l)}$ can be summation, mean, or maximum.

  The issue here is that the information from node v itself could get lost, since $\boldsymbol{h}_v^{(l)}$ does not directly depend on $\boldsymbol{h}_v^{(l-1)}$. The solution is to include $\boldsymbol{h}_v^{(l-1)}$ in the computation of $\boldsymbol{h}_v^{(l)}$. We can compute a message for v itself, and then use

  $$\boldsymbol{h}_v^{(l)} := \text{concat}(\text{Agg}(\{\boldsymbol{m}_u^{(l)} : u \in N(v)\}), \boldsymbol{m}_v^{(l)})$$

- A non-linear **activation** function follows message or aggregation to add expressiveness.

Examples:

(1) **Graph Convolutional Networks** (**GCN**), where the message and aggregation functions are

$$\boldsymbol{h}_v^{(l)} := \underbrace{\sigma}_{\text{Activation}}\left(\underbrace{\sum_{u\in N(v)}}_{\text{Aggregation}} \underbrace{\mathsf{W}^{(l)}\frac{1}{\boldsymbol{h}_u^{(l-1)}}\deg v}_{\text{Message}}\right)$$

where

$$\boldsymbol{m}_u^{(l)} = \underbrace{\frac{1}{N(v)}}_{\text{Normalized by node degree}} \mathsf{W}^{(l)}\boldsymbol{h}_u^{(l-1)}$$

$$\boldsymbol{h}_v^{(l)} = \sigma(\sum\left\{\boldsymbol{m}_u^{(l)} : \underbrace{u \in N(v)}_{\substack{\text{GCN Graphs are assumed to have self-edges that are included in the summation.}}}\right\})$$

(2) **GraphSAGE**:

$$\boldsymbol{h}_v^{(l)} := \sigma\left(\underbrace{\mathsf{W}^{(l)}\cdot \text{concat}(\boldsymbol{h}_v^{(l-1)}}_{\text{Stage 2 aggregation with self}}, \underbrace{\text{Agg}(\{\boldsymbol{h}_u^{(l-1)} : u \in N(v)\})}_{\text{Stage 1 aggregation from neighbours}})\right)$$

Message is computed within $\text{Agg}(\cdot)$.

A couple different aggregation functions exist:

- Mean: Weighted average of neighbours

$$\text{Agg}(v) := \sum_{u\in N(v)} \frac{\boldsymbol{h}_u^{(l-1)}}{|N(v)|}$$

- Pool: Transform neighbour vectors and apply symmetric vector function mean or max.

$$\text{Agg}(v) := \text{mean}_{u\in N(v)}\, \mathsf{M}LP(\boldsymbol{h}_u^{(l-1)})$$

- LSTM: Apply LSTM to reshuffled neighbours

$$\text{Agg}(v) := \mathsf{L}STM(\boldsymbol{h}_u^{(l-1)} : u \in \pi(N(v)))$$

Optionally, apply L^2-normalisation to $\boldsymbol{h}_v^{(l)}$ at every layer. Without L^2 normalisation, the embedding vectors have different scales for vectors. In some cases, normalisation results in performance improvements.

(3) **Graph Attention Networks** (**GAT**):

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

GAT assigns different importance to messages coming from different nodes. When $\alpha_{v,u} = \frac{1}{N(v)}$, attention reduces to GCN/GraphSAGE, where $\alpha_{v,u}$ is defined *explicitly* based on the structural properties of the graph, specifically, the node degree $\deg v$. The attention mechanism in GAT is inspired by cognitive attention and focuses on importants parts of the input data.

An **attention mechanism** computes $\alpha_{v,u}$. Define the **attention coëfficients**

$$e_{v,u} := a(\mathsf{W}^{(l)}\boldsymbol{h}_u^{(l-1)}, \mathsf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

Then we normalise $e_{v,u}$ into the attention weight using softmax:

$$\alpha_{v,u} := \frac{\exp e_{v,u}}{\sum_{k\in N(v)} \exp e_{v,k}}$$

In **Multi-head Attention**, multiple attention scores are used and the result of each attention "head" is aggregated:

$$\boldsymbol{h}_v^{(l)}[j] := \sigma\Big(\sum_{u\in N(v)} \alpha_{v,u}[j]\mathsf{W}^{(l)}\boldsymbol{h}_u^{(l-1)}\Big)$$

$$\boldsymbol{h}_v^{(l)} := \mathrm{Agg}(\boldsymbol{h}_v^l[j] : j)$$

The benefits of attention mechanism are:

- Allows for implicitly specifying different importance values of neighbours
- Computationally efficient: Attention can be parallelised across all edges of the graph.
- Storage efficient: Sparse matrix operations do not require more than $\mathrm{O}(V + E)$ entries. The number of parameters is fixed.
- Localised: Only attends over local neighbourhoods
- Inductive capability: Does not depend on global graph structure.

4.2 GNN Layers in Practice

In practice, the classic GNN layers are a starting point.

- We can often get better performance by considering a general GNN layer design
- Concretely, we can include modern deep learning modules. e.g. Batch Normalisation, Dropout[3], Attention/Gating, and others.

  One particular note about activation function: Empirically the **Parametric ReLU** function, defined as

  $$\mathrm{PReLU}(x) := \max(x, 0) + \mathsf{a}\min(x, 0)$$

  performs better than ReLU.

Designing novel GNN layers is a still an active research frontier. You can explore diverse GNN designs or try your own ideas in GraphGym!

Attention weights

[3]In GNNs, dropout is applied to the *linear* layer of the message function.

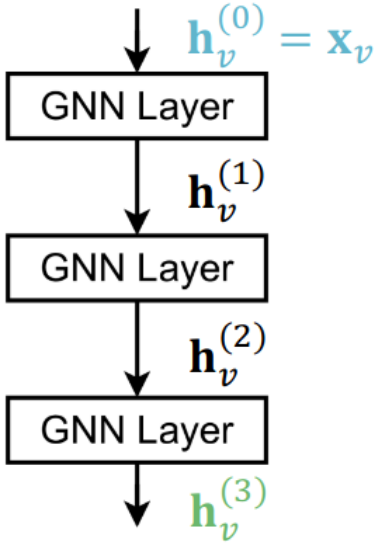


Figure 4.1: Standard way of stacking GNN layers

4.3 Stacking GNN Layers

How to connect GNN layers into a GNN? In the standard way, we stack GNN layers sequentially. This leads to the **over-smoothing problem**. Since each node's receptive field becomes larger and larger, the receptive fields of two different nodes in the graph eventually converge into one, which causes all the node embeddings to become convergent. Hence

1. Be cautious when stacking GNN Layers. More layers do not always help, unlike neural networks in other domains. The layers should be a bit more numerous than the radius of the receptive field, but there should not be too many layers.

   - Solution 1: Increase the expressive power within each GNN layer: We can make aggregation/transformation into a deep neural network.
   - Solution 2: Add layers that do not pass messages. A GNN does not necessarily only contain GNN layers. We can apply MLP layers, applied to each node, before and after GNN layers as *pre-process* and *post-process* layers. In practice adding these layers are beneficial.

2. Add **skip connections** in GNNs: Since earlier GNN layers can sometimes be better in differentiating nodes, we add shortcuts to the GNN.

   A skip connection creates a *mixture of models*. We get a mixture of shallow and deep GNNs using mixed connections. When we have N skip connections, information has 2^N possible pathways of transmission. An example of a skip connection:

$$\boldsymbol{h}_v^{(l)} := \sigma\left(\overbrace{\sum_{u \in N(v)} \mathrm{W}^{(l)} \frac{\boldsymbol{h}_u^{(l-1)}}{|N(v)|}}^{F(x)} + \overbrace{\boldsymbol{h}_v^{(l-1)}}^{x}\right)$$

   Another option is to directly skip to the last layer.

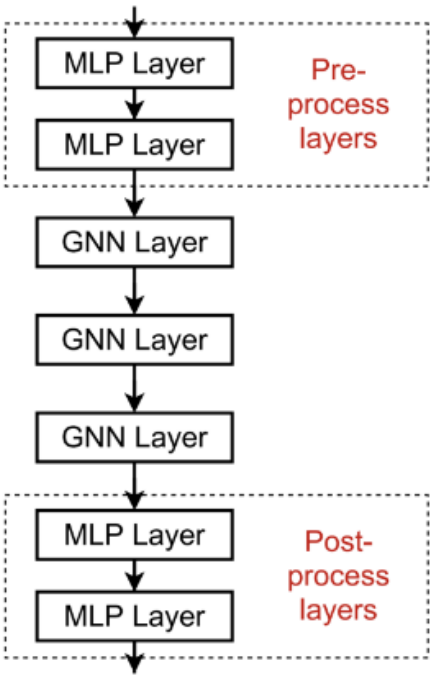
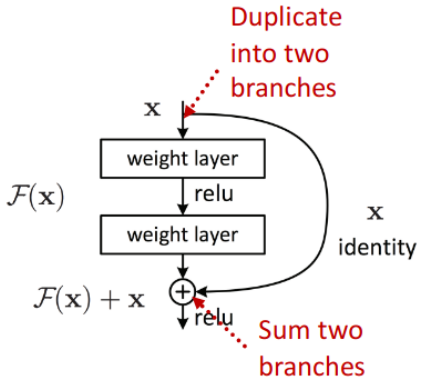
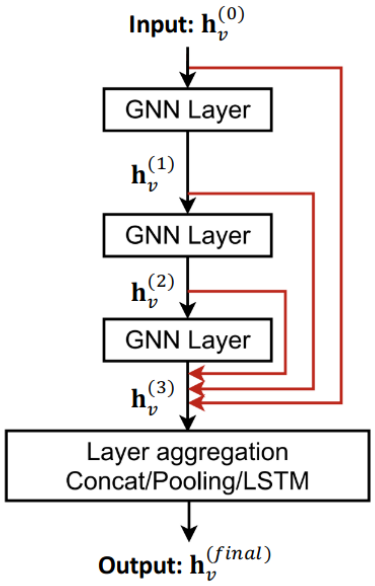


Figure 4.2: GNN with pre- and post-process layers



(a) A shortcut or skip connection



(b) Skip connections to the last layer

4.4 Graph Manipulation in GNNs

The raw input graph is not always suitable for using as a computation graph. The reasons could be:

- Feature Level:

  - Input graph *lacks features*: Feature augmentation
    (a) Assign constant values to nodes: Drawback on expressive power.
    (b) Assign unique ids to nodes: Drawback on inductive learning and computational cost.

  Certain structures are hard to learn by GNN. For example, the cycle count feature (the length of a cycle that v resides in). We could embed the cycle count as a feature. Other commonly used augmented features are clustering coëfficient, PageRank, centrality.

- Structure level:
  - The graph is *too sparse* leading to inefficient message passing
    (a) We could add virtual edges: e.g. Connect 2-hop neighbours via virtual edges. Intuitively, we use $\boldsymbol{A} + \boldsymbol{A}^2$ as the adjacency matrix instead of just $\boldsymbol{A}$. This is useful in bipartite graphs.
    (b) Add virtual node that connects to all nodes in the graph. This greatly improves message passing in sparse graphs.
  - The graph is *too dense* leading to costly message passing
    We could sample neighbours when doing message passing. i.e. during aggregation, only a select random subset of $N(v)$ have their messages passed to v.
  - The graph is *too large* which cannot fit the computational graph into GPU.
    We could sample subgraphs to compute embeddings. This will be covered later in lecture: Scaling up GNNs.

Its unlikely the input graph happens to be the optimal computation graph.

5 GNN Augmentation and Training

5.1 Predictions with GNNs

so far we have covered graph neural networks and node embeddings. With the node embeddings we have created, we can execute different prediction tasks Different levels require different prediction heads.

- **Node-level Prediction**: We can directly make prediction using node embeddings using a linear prediction head

$$\hat{\boldsymbol{y}}_v := \text{Head}(\boldsymbol{h}_v^{(L)}) = \mathsf{W}\boldsymbol{h}_v^{(L)}$$

- **Edge-level Prediction**: Make prediction using pairs of node embeddings

$$\hat{\boldsymbol{y}}_{u,v} := \text{Head}(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)})$$

  This can be implemented using

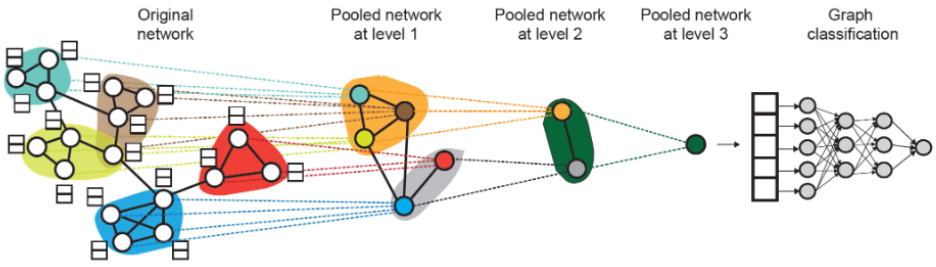


Figure 5.1: DiffPool: A Hierarchical Pooling

– Concatnation and Linear layers:

$$\text{Head}(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)}) = \mathsf{W}_1 \boldsymbol{h}_u^{(L)} + \mathsf{W}_2 \boldsymbol{h}_v^{(L)}$$

– Dot product:

$$\text{Head}(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)}) = \boldsymbol{h}_u^{(L)} \mathsf{W} \boldsymbol{h}_v^{(L)}$$

where W can be $\boldsymbol{I}$, or a set of k matrices for k-way label prediction.

- **Graph-Level Prediction**:

$$\hat{\boldsymbol{y}}_G := \text{Head}(\boldsymbol{h}_v^{(L)} : v \in V(G))$$

This is similar to the aggregation step in GNNs. Global pooling, e.g. mean pooling, max pooling, sum pooling, work great for smaller graphs.

The main issue is global pooling over a large graph loses information. A solution is to aggregate the graph hierarchically. We train two independent GNNs *at each level*.

- GNN-A: Compute node embeddings (embedding task)
- GNN-B: Compute the cluster a node belongs to (clustering task)

For each pooling layer, use clustering assignments from GNN-B to aggregate node embeddings generated by GNN-A.

Question: Is GNN-A trained on the entire graph?

In the first layer of clustering, it is. In the subsequent layers it is trained on the clustered graph.

5.2 Training Graph Neural Networks

The ground truth of training GNNs come from either

- (Supervised learning) External data e.g. predict drug likeness of a molecular graph.
- (Unsupervised learning) Graph features e.g. link prediction, predicting whether two nodes are connected. This is sometimes called **self-supervision**.

Sometimes the differences are blurry.

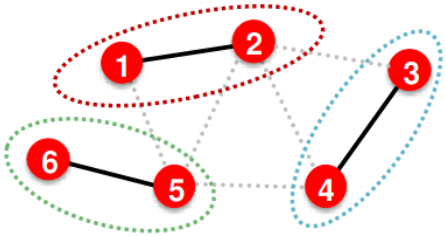


Figure 5.2: A training/validation/test split schematic. In transductive settings, the grey edges would be included. In inductive settings they would not.

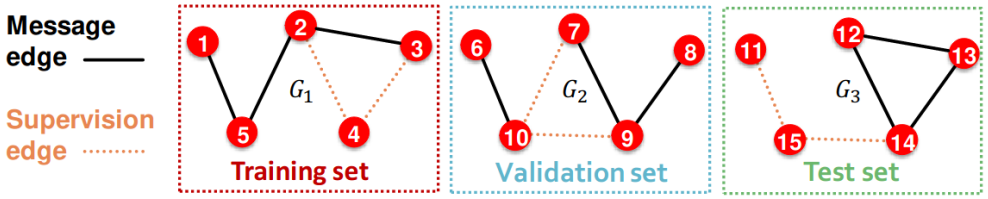


Figure 5.3: An inductive dataset of three different graphs. Each graph is given its own message and supervision edges.

5.3 Setup GNN Prediction

How to split dataset into train/validation/test set? The three datasets are used for:

- **Training set**: Used for optimising GNN parameters
- **Validation set**: Used to develop model and hyperparameters
- **Test set**: Used to report final performance.

Sometimes we cannot guarantee that the test set will be really held out. In which case we could use *random spplit* and report the average performance over different random seeds. Splitting graphs is special and has its own quirkiness compared to image dataset. If we split a graph into different vertices, the nodes are *not* independent. The nodes in the "unseen" validation or test set will affect our prediction on the nodes in the training set, because of message passing mechanics. There are two solutions to this issue:

- **Transductive Setting**: The input graph can be observed in all datasets splits, but only the training sets have visible labels.
- **Inductive Setting**: Break the edges between splits to get multiple graphs. In this case the nodes in different components of the graph are truly independent.

  Only this setting is applicable to *graph classification*.

In a link-prediction task, the setup of the task is tricky. It is a self-supervised task and we need to generate labels and datasets on our own. A practical method is to hide edges from the GNN and let GNN predict if those edges exist. We split edges twice:

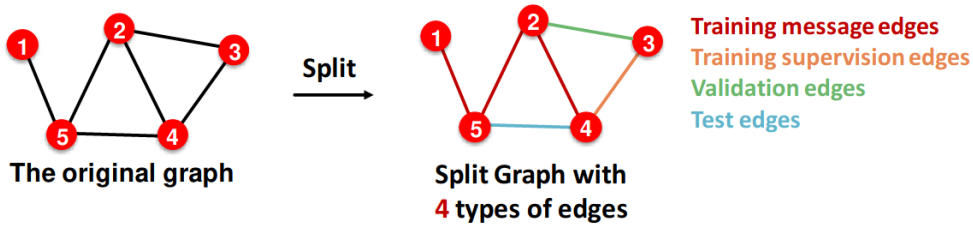


Figure 5.4: A transductive dataset

1. Assign two types of edges in the original graph: Message edges and Supervision edges. Message edges will be visible to the GNN while supervision edges will not.

2. Split edges into train/validation/test. We can either use transductive (Figure 5.3) or transductive (Figure 5.4).

   The transductive setting is the default when people talk about link prediction. In this case there is only one graph, observable in all dataset splits.

Stage	GNN Input	Labels
Training	Training message edges	Training supervision edges
Validation	Training message/supervision edges	Validation edges
Test	Training message/supervision, and validation edges	Test edges

Link prediction is fully supported in PyG and GraphGym.

6 Theory of Graph Neural Networks

A question we wish to answer: How powerful are GNNs? Specifically:

- What is the expressive power of GNNs such as GCN, GAT, GraphSage?
- How to design a maximally expressive GNN model?

Settings

We focus on message passing in GNNs:

1. Message: Each node computes a message

$$\boldsymbol{m}_u^{(l)} := \mathsf{Message}^{(l)}(\boldsymbol{h}_u^{(l)})$$

2. Aggregate: Aggregates messages from neighbours

$$\boldsymbol{h}_v^{(l)} := \mathsf{Aggregate}^{(l)}(\{\boldsymbol{m}_u^{(l)} : u \in N(v) \cup \{v\}\})$$

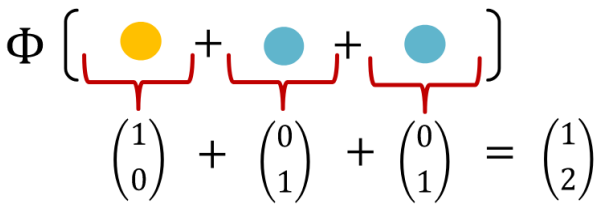


Figure 6.1: When f produces one-hot encodings, the ϕ counts the number of occurrences of each element of the multi-set.

A GNN distinguishes graph structures using the computation graphs induced by the neighbourhood of each node. If the k-hop neighbourhood structures of two nodes are identical and the nodes have the same features, a GNN would not be able to distinguish between them. A computation subgraph is a rooted subtree with root at each node.
We can measure expressive power using injections. The most expressive GNNs should map subtrees to node structure injectively. If each step of GNN's aggregation can *completely* retain the neighbourhood information of each node, the generated node embeddings can distinguish different rooted structures. In other words, the most expressive GNN uses *injective neighbourhood aggregation*.

6.1 Designing the Most Powerful GNN

Observe that the expressiveness of GNNs can be characterised by the neighbourhood aggregation function they use. It can be abstracted as a function over a multi-set

$$\mathsf{Aggregate}(\{\boldsymbol{x}_u : u \in N(v)\})$$

In GCN, this is the mean function, and in GraphSAGE, it is the max pool. For example, both pooling functions will create the same aggregation over the neighbour message multi-sets:

$$\left\{\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}\right\}, \qquad \left\{\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}\right\}$$

In general, the discriminative power decreases in the series

$$\text{sum (multiset)} > \text{mean (distribution)} > \text{max(set)}$$

Theorem 6.1 (Xu et al. ICLR 2019)**.** *Any injective multi-set function can be expressed as*

$$\phi\left(\sum_{x \in S} f(x)\right)$$

where ϕ, f are non-linear functions.

Proof. (sketch) f produces one-hot encodings, and ϕ adds them together. See Figure 6.1. □

To model ϕ and f, we can use a MLP.

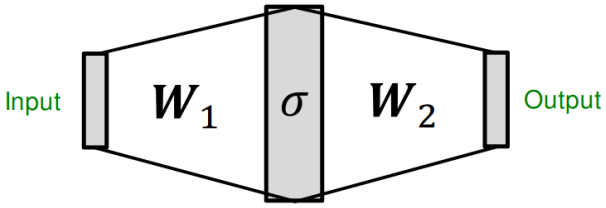


Figure 6.2: MLP with one hidden layer

Theorem 6.2 (Universal Approximation Theorem, Hornik et al., 1989)**.** *1-hidden-layer MLP with sufficiently large hidden dimensionality and non-polynomial activation function σ can approximate any function to arbitrary accuracy.*

Therefore we can use the following structure to model any injective multiset function. Usually a hidden dimension of 100 to 500 is sufficient. This brings us the most expressive GNN: **Graph Isomorphism Network** (**GIN**).

$$\mathsf{MLP}_\phi\left(\sum_{x\in S}\mathsf{MLP}_f(x)\right)$$

GIN uses the two-MLP structure above and it is the most expressive message passing GNN.

Question: Why are we using a MLP as the multiset function?

We could also use hash functions, but the benefit of a neural network is that we will be able to learn depending on the specific task.

Question: Can we just assign randomness to each node to distinguish them?

Yes, but we want to specifically map same computation graphs to the same output. Randomness would distinguish every node.

Question: Why should we map same neighbourhood structure to the same embedding and disregard the individual identity of the node?

In real use cases its very rare to see two nodes with identical computation graphs. Another architecture, position-aware GNN, solves this problem. We shall see it in a future lecture.

Question: Why would we want the aggregation function to be learnable? Why can't we just use a hash function in place of MLP?

The hash function is indeed very expressive. The main motivation of using a hash function is that

1. Each node has features.
2. The neural network is differentiable and can be trained in conjunction.

GIN is comparable to the **Weisfeiler-Lehman Kernel** (WL Kernel) / **Colour-Refinement** algorithm. Given a graph G with nodes V,

1. Assign an initial colour $c^{(0)}(V)$ to each node v.
2. Iteratively refine node colours by
$$c^{(k+1)}(v) := \text{hash}(\{c^{(k)}(v)\} \cup \{c^{(k)}(u) | u \in N(v)\})$$
3. After k steps of colour refinement, $c^{(k)}$ summarises the structure of k-hop neighbourhood.

The WL Kernel is *computationally efficient*. The total time complexity is $O(|E|)$. where the aggregation function is a hash function.
GIN models the hash function in the WL kernel using neural networks.

Theorem 6.3 (Xu et al. ICLR 2019)**.** *Any injective function over the tuple*

$$(\boldsymbol{c}^{(k)}(v), \{\boldsymbol{c}^{(k)}(u)\}_{u \in N(v)})$$

Root feature — Neighbouring features

can be modeled as

$$\begin{aligned}&\mathsf{GINConv}(\boldsymbol{c}^{(k)}(v), \{\boldsymbol{c}^{(k)}(u) : u \in N(v)\}) \\ &:= \mathsf{MLP}_\phi\left((1+\epsilon)\mathsf{MLP}_f(\boldsymbol{c}^{(k)}(v)) + \sum_{u \in N(v)} \mathsf{MLP}_f(\boldsymbol{c}^{(k)}(u))\right)\end{aligned}$$

Question: Why is the ϵ needed here?

It allows differentiating the node itself with its neighbours.

Question: In the hash table, we would not be able to control the output (almost random), but in our case the output seems to be deterministic.

The discussion here is mainly about how to design an injective function over a multi-set.

Question: What happens to nodes in the network with no edges? These edges cannot be distinguished by arbitrarily deep WL kernel hashing or GNN.

In this case, the two nodes are just isolated nodes. These nodes would have identical neighbourhood structure and identical embedding.

If the input features $c^{(0)}(v)$ is one-hot, direct summation is injective. In this case, we only need ϕ to ensure injectivity:

$$\mathsf{GINConv}(\boldsymbol{c}^{(k)}(v), \{\boldsymbol{c}^{(k)}(u) : u \in N(v)\}) := \mathsf{MLP}\left((1+\epsilon)\boldsymbol{c}^{(k)}(v) + \sum_{u \in N(v)} \boldsymbol{c}^{(k)}(u)\right)$$

Advantages of GIN over WL:

	Update Target	Update Function
WL Graph Kernel	Node-colours (one-hot)	hash
GIN	Node embeddings (low-dim vectors)	GINConv

- Node embeddings are *low-dimensional*. Hence they can capture the fine-grained similarity of different nodes.
- Parameters of update function can be learned from *downstream tasks*.

WL Kernel has been both theoretically and empirically shown to distinguish most of the real-world graphs [Cai et al. 1992]. Hence GIN is also powerful enough to distinguish most of the real graphs.

6.2 When things don't go as planned

- Data preprocessing is important! Node attributes should be normalised.
- Optimiser: Adam is relatively robust to learning rate
- Activation Function:
  - ReLU often works well.
  - Other good alternatives: LeakyReLU, PReLU
  - No activation at the output layer
  - Include bias at every layer
- Embedding dimensions: $32, 64, 128$ are good starting points.

Debugging Deep Neural Networks:

- Loss/Accuracy not converging:
  - Check pipeline (e.g. in PyTorch we need to zero the gradients)
  - Adjust hyperparameters such as learning rate
  - Pay attention to weight parameter initialisation
  - Scrutinise loss function
- Important for model development:
  - Overfit on (part of) training data
    With a small training dataset, loss should be essentially close to 0 with an expressive neural network.
  - Monitor the training and validation loss curve.

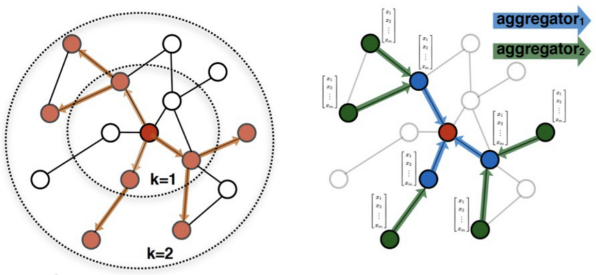


Figure 7.1: A k-layer GNN embeds a node based on the k-hop neighborhood structure

.

Figure 7.2: The two square nodes have the same computational graphs and therefore the same embedding despite having different neighbourhood structures.

7 Limits of Graph Neural Networks

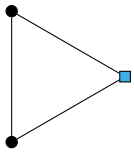
What should a perfect GNN do? Intuitively, a perfect GNN should build an injective function between *neighbourhood structure* and *node embeddings*. Therefore, in a perfect GNN: Therefore, in a perfect GNN:

1. If two nodes have the same neighborhood structure, they must have the same embedding
2. If two nodes have different neighborhood structure, they must have different embeddings

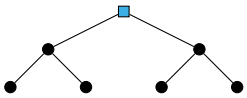
We observe that a perfect GNN should:

1. If two nodes have the same neighborhood structure, they must have the same embedding
2. If two nodes have different neighborhood structure, they must have different embeddings

Observation (2) is often unsatisfiable: There are basic structures that existing GNN frameworks cannot distinguish, such as the length of cycles. GNNs power can be improved in to resolve this problem.
Observation (1) could also be problematic: Sometimes we may want to assign different embeddings to nodes that have different positions in the graph. e.g. In position-aware tasks.
We'll resolve these issues by building more expressive GNNs.



(a) Node on a cycle



(b) Computational graph

Figure 7.3: Failure 1: The computational graph of a node on a cycle is always the same

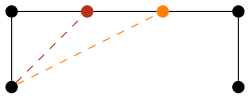


Figure 7.4: Failure 2: The computational graph of • and • are the same, so the link-level prediction on two dashed edges will be identical.

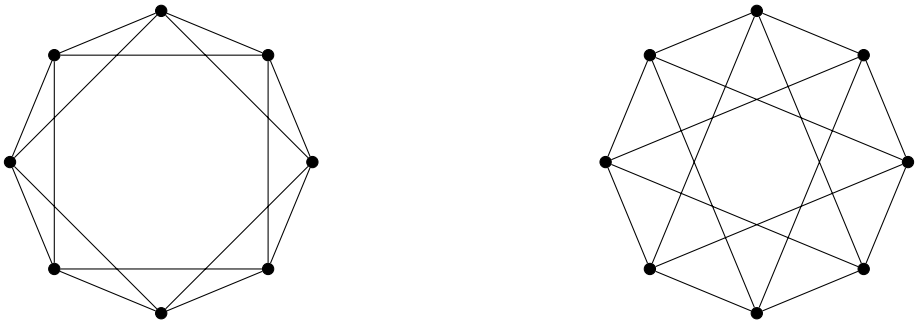


Figure 7.5: Failure 3: Nodes on two different graphs have identical computational graphs

Figure 7.6: The WL Kernel inherits graph symmetries. Symmetric colours are associated with limitations involving spectral decomposition of a graph.

7.1 Spectral Perspective of Message Passing

Due to its high symmetry, GNN cannot perform perfectly in structure aware tasks either. GNNs exhibit three levels of failure in structure-aware tasks:

- Node Level: Different inputs with the same computational graph leads to GNN failure (Figure 7.3)
- Edge Level: Edge prediction tasks may fail since the nodes on the edges have identical computational graphs. (Figure 7.4)
- Graph Level: Same overall computation graphs on different graphs lead to same prediction. (Figure 7.5)

Recall the definition of the Graph Isomoephism Network (GIN)

$$\underbrace{\boldsymbol{c}_v^{(l+1)}}_{\text{Next layer's colour}} := \mathsf{MLP}\left((1+\epsilon)\underbrace{\boldsymbol{c}^{(k)}(v)}_{\text{Root feature}} + \sum_{u\in N(v)} \underbrace{\boldsymbol{c}^{(k)}(u)}_{\text{Neighbouring features}}\right)$$

We can unroll the first MLP Layer

$$\boldsymbol{c}_v^{(l+1)} := \underbrace{\mathsf{MLP}_{-1}}_{\text{All MLP Layers except first}}\left(\sigma\left(\boldsymbol{W}_0^{(l)}(1+\epsilon)\boldsymbol{c}^{(k)}(v) + \sum_{u\in N(v)} \boldsymbol{W}_1^{(l)}\boldsymbol{c}^{(k)}(u)\right)\right)$$

This can be written in matrix form:

$$\boldsymbol{C}^{(l+1)} = \mathsf{MLP}_{-1}\left(\sigma\left(\boldsymbol{C}^{(l)}\,\boldsymbol{W}_0^{(l)} + \boldsymbol{A}\,\boldsymbol{C}^{(l)}\,\boldsymbol{W}_1^{(l)}\right)\right) = \mathsf{MLP}_{-1}\left(\sigma\left(\sum_{k=0}^{1} \underbrace{\boldsymbol{A}^k}_{\text{Adjacency matrix}}\,\underbrace{\boldsymbol{C}^{(l)}}_{\boldsymbol{C}^{(l)}[v,:]=\boldsymbol{c}_v^{(l)}}\,\boldsymbol{W}_k^{(l)}\right)\right) \quad (1)$$

We can compute the *eigen-decomposition* of an adjacency graph

$$\boldsymbol{A} = \boldsymbol{V}\,\underbrace{\boldsymbol{\Lambda}}_{\text{Diagonal matrix of eigenvalues } \lambda_1,\dots,\lambda_N}\,\boldsymbol{V}^\intercal$$

The eigen-decomposition of $\boldsymbol{A}$ is a universal characterization of the graph.

Example

The number of cycles in a graph can be viewed as functions of eigenvalues and eigenvectors, e.g.

$$\operatorname{diag}(\boldsymbol{A}^3) = \sum_{n=1}^{N} \lambda_n^3 \,\|\boldsymbol{v}_n\|^2$$

We can interpret GIN layers as MLPs operating on the eigenvectors. Replacing $\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^\intercal$ in Equation 1:

$$\begin{aligned}
\boldsymbol{C}^{(l+1)} &= \mathsf{MLP}(\boldsymbol{W}) = \mathsf{MLP}_{-1}(\sigma(\boldsymbol{V}\,\boldsymbol{W})) \\
\boldsymbol{W}[n,f] &= \left(\sum_{k=0}^{1} \cancel{\boldsymbol{V}}\boldsymbol{\Lambda}^k\,\boldsymbol{V}^\intercal\,\boldsymbol{C}^{(l)}\,\boldsymbol{W}_k^{(l)}\right)[n,f] \\
&= \sum_{i=1}^{d}\sum_{k=0}^{1} \lambda_n^k\,\boldsymbol{W}_k[i,f]\langle \boldsymbol{v}_n \cdot \boldsymbol{C}^{(l)}[:,i]\rangle
\end{aligned}$$

Thus the weights of the first MLP layer depends on the eigenvalues and the dot product between the eigenvectors $\boldsymbol{v}_n$ and the colours at the previous level $\boldsymbol{C}^{(l)}[:,i]$.
With uniform initial colours, we have $\boldsymbol{C}^{(l)}[:,i] = \boldsymbol{1}$. The new node embeddings only depend on the eigenvectors that are not orthogonal to 1. However, graphs with symmetries admit eigenvectors orthogonal to $\boldsymbol{1}$.

In a nutshell: WL cannot distinguish between symmetric nodes in the graph since the embeddings and graphs structure admit the same symmetry. *The limitations of the WL kernel are limitations of the initial node color.*

Question: How does the dot product relate to the initial colour?

It is a little bit beyond the scope of the course, but the initial colour $\boldsymbol{C}^{(1)}$. In graphs that have symmetries, the inner product $\boldsymbol{v}_n \cdot \boldsymbol{1}$ goes to zero, and the information from $\boldsymbol{C}^{(l)}[:, i]$ is extinguished.

7.2 Feature-Augmentation: Structurally-Aware GNNs

A naïve solution is *one-hot encoding*: Assign each node with a different id. The issues are:

1. Non-scalable: Needs $\mathrm{O}(n)$ feature dimensions
2. Non-inductive: Cannot generalize to new graphs. A graph with a different ordering of nodes but the same structure will have different embedding.

One-hot encoding:

\+ Has high expressive power since each node has a unique id

\- Has low generalizability and cannot generalize to new nodes. New nodes introduce new id's.

\- Has high computational cost

Is mainly used for small graphs and transductive settings.

In comparison, constant node features:

~ In terms of expressive power, all nodes are identical, but the GNN can still learn from structure.

\+ Is simple to generalize to new nodes by assigning constant features to them.

\+ Has low computational cost (1 dimensional feature)

\+ Can be used for any graph and inductive settings.

We can also use the diagonals of the adjacency powers as augmented node features. They correspond to the closed loops each node is involved in.

$$\boldsymbol{C}^{(0)} = [\mathrm{Diag}(\boldsymbol{A}^0), \ldots, \mathrm{Diag}(\boldsymbol{A}^{(D-1)})] \in \boldsymbol{N}^{(N \times D)}$$

Theorem 7.1. *If two graphs have adjacency matrices with different eigenvalues, there exists a GNN with closed-loop initial node features that can always tell them apart.*

GNNs with structural initial node features can produce different representations for *almost all* real-world graphs. Almost all since distinguishing graphs is an open problem. In this case, a GIN with structural initial node features is *strictly more powerful* than the WL Kernel.
Certain structures are hard to learn by GNN. For example, the cycle count feature (the length of a cycle that v resides in). We could embed the cycle count as a feature. Other commonly used augmented features are clustering coëfficient, PageRank, centrality.
Structurally aware node feature:

+ Has high expressive power, so node-specific information can be stored.
+ Is simple to generalize to new nodes. Can count triangles or closed loops.
+ Has low computational cost
+ Can be used for any graph

7.3 Counting Graph Sub-Structures

Can we count graph substructures with GNNs only? It turns out we can. Suppose we assign unique ID to nodes, where each assignment $\mathbf{c}$[4] leads to output $\mathbf{y}$. To maintain equivariance the final output, we take expectation over unique id's over the nodes

$$\boldsymbol{y} := \mathbb{E}[\mathbf{y}]$$

This maintains equivariance. In practice, it is computed as

$$\boldsymbol{y} := \frac{1}{m}\sum_{j=1}^{m} \boldsymbol{y}^{(j)}$$

We allow the GNN to momentarily break equivariance during each individual sample, but equivariance holds after taking expectation.

Question: Is this more computationally efficient than eigen-decomposition

It depends on the size of the graph. Eigen-decomposition is $O(n^2)$. Often sampling random id's is more efficient.

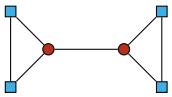
Question: Does the distribution matter

Sometimes. If the distribution is structurally aware, you can compute more stuff. Its a trade-off between complexity of distribution and power of the NN.

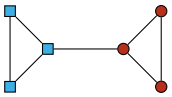
Question: Is this affected by the number of layers in the GNN?

Yes, if you don't normalize.

[4]Following Ian Goodfellow's convention of using upright letters as random variables.



(a) Structure-Aware Task



(b) Position-Aware Task

Figure 7.7: Structure and Position Aware Tasks; GNNs often work well for structure-aware tasks but fail at position-aware tasks.

1. **Positive-aware Tasks**: We may wish to assign different embeddings to nodes with different positions in the network.

   Solution: Position-aware GNNs

   A naïve approach assigns unique one-hot labels to each vertex. This is infeasible since it is

   - Non-scalable: It requires $O(|V|)$ features
   - Non-inductive: Cannot generalise to new graphs

2. The expressive power of GNNs is upper bounded by the WL test. For example, message passing GNNs cannot count the length of a cycle in a graph.

   Solution: Identity-aware GNNs

7.4 Position-Aware GNN

Two types of tasks on graphs:

- **Structure-Aware Tasks**: Nodes are labeled by their *structural roles* in the graph
- **Position-Aware Tasks**: Nodes are labeled by their position in the graph.

GNNs always fail for position aware tasks due to the similarity of computational graphs. We could randomly pick a node s_1 as an **anchor node** and represent other nodes by their relative distances w.r.t. s_1. The anchor node serves as a coördinate axis. We can pick more nodes $s_2, s_3, \ldots$ as anchor nodes to better characterise node positions in the graph.

Theorem 7.2 (Bourgain's Theorem, Informal)**.** *Let (V, d) be the metric space on the graph vertices V and c a constant. Select random nodes $S_{i,j} \subseteq V$ such that each node in v has a probability of 2^{-i} of being included.*
Consider the following embedding function

$$f(v) := [d_{\min}(v, S_{1,1}), d_{\min}(v, S_{1,2}), \ldots, d_{\min}(v, S_{\log n, c \log n})] \in \mathbb{R}^{c \log^2 n}$$

where $d_{\min}(v, S) := \min_{u \in S} d(v, u)$.
Then the embedding distance produced by f under $\|\cdot\|_2$ is "close" to d[5]*.*

[5]See `https://www.cs.toronto.edu/~avner/teaching/S6-2414/LN2.pdf`

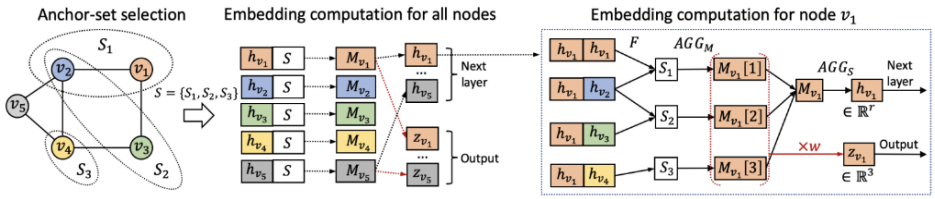


Figure 7.8: Position-aware GNN using permutation invariant NN

P-GNN[6] follows this theory. It samples $O(\log^2 n)$ anchor sets $S_{i,j}$ and embeds each node via f. The embedding positional information can be used:

- Simple solution: Use the position encodings as an augmented node feature. The problem with this is since the encoding is tied to a random anchor set, dimensions of positional encoding can be randomly permuted without changing its meaning.
- Rigorous solution: Use a special NN that can maintain permutation invariant property of position embedding

Question: How is this embedding different from Node2Vec?

Node2Vec boils down to particular matrix factorisation. Bourgain Theorem's embedding is about anchoring high dimensional metric spaces into $\mathbb{R}^n$ and more geometric.

Question: What do we pick for d in Bourgain Theorem?

Usually people use the shortest path distance. This is dependent on the application.

- Training: New anchor sets are re-sampled every time
- Inference: Given a new unseen graph, new anchor sets are sampled.

Question: Why sample new anchor nodes every time?

Although the anchor nodes are randomised, the distances from these anchor nodes follow a relatively stable distribution.

Question: Why is it better to have multiple anchor nodes?

Larger anchor sets reduce the variance of the distribution of node distances.

In some applications it may be better to manually engineer an anchor set.

Question: Do different metrics afford different performances?

There is not a clear answer.

[6]See `https://arxiv.org/abs/1906.04817`

7.5 Identity-Aware GNNs

ID-GNN uses **heterogeneous message passing**. Nodes of different colourings use different message and aggregation functions, i.e.

$$\boldsymbol{h}_u^{(k)} := \text{Aggregate}^{(k)}\left(\left\{\text{Message}^{(k)}_{\mathbb{1}[s=v]}(\boldsymbol{h}_s^{(k-1)}) : s \in N(u)\right\}, \boldsymbol{h}_u^{(k-1)}\right)$$

Depend on whether v is the centre node s

Intuitively, ID-GNN can count cycles originating from a given node but GNN cannot. Based on this, we propose a simplified version **ID-GNN-Fast**, where we use cycle count (the number of times the coloured node is reached) as an augmented node feature.

Question: Is there a case where the node colouring would not be useful in network A and B?

The networks A,B will not converge to the same result since they might but they won't if your objective function forces them to distinguish the nodes.

Question: Is the node colouring similar to node anchors?

Position-Aware is more macroscopic and colouring is more microscopic. These two concepts are orthogonal.

Question: Can we view the coloured graphs as a heterogeneous graph?

Yes.

8 Graph Transformers

We know a lot about the design space of GNNs. What does the design space of graph transformers look like?
A **transformer** is a type of deep learning model featuring:

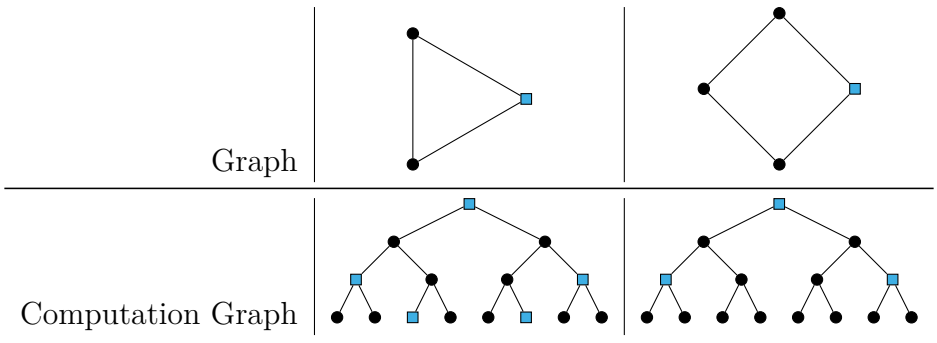


Figure 7.9: Inductive Node Colouring distinguishes computational graphs on cycles of length 3 and 4.

(a) 3-cycle (b) 4-cycle

Figure 7.10: Cycle counts

1. **Self-Attention Mechanism**: An attention layer with the same query, key, and value vectors.
2. **Encoder-Decoder Architecture**: The input to the transformer is first embedded in an embedding space. The output undergoes the reverse transformation during generation.
3. **Positional Encoding**: Attention mechanism has no innate knowledge of position, so a positional encoding helps the mechanism.
4. **Multi-head Attention**: Multiple attention layers are used for each transformer model.
5. **Feed-forward NN**: A feed-forward network at the output end of the attentions processes the output.

The applications are:

- Natural Language: Transformers are the basis of BERT, GPT, and T5
- Vision: Vision Transformers (ViTs) apply the attention mechanism to sections of the input
- Graphs: Transformers can model relationships between nodes

Transformers map 1D vectors of input data to 1D vectors of output data. Each element in the vector is a **token**. Each token represents a "unit" of data e.g. a word. The output of a transformer can either be this token sequence or a *pooled* output by combining all output symbols.
Before we discuss multi-head attention, we have to discuss *single-head attention*.

8.1 Self-Attention

The attention layer processes each input token $\boldsymbol{x}_i$ into three values, the **query** $\boldsymbol{q}_i$, **key** $\boldsymbol{k}_i$, and **value** $\boldsymbol{v}_i$[7], via 3 trainable models.

$$\boldsymbol{q}_i := \mathbf{W}_q \boldsymbol{x}_i, \qquad \boldsymbol{k}_i := \mathbf{W}_k \boldsymbol{x}_i, \qquad \boldsymbol{v}_i := \mathbf{W}_v \boldsymbol{x}_i$$

[7]This terminology comes from search engines where the user inputs a query which gets matched with keys.

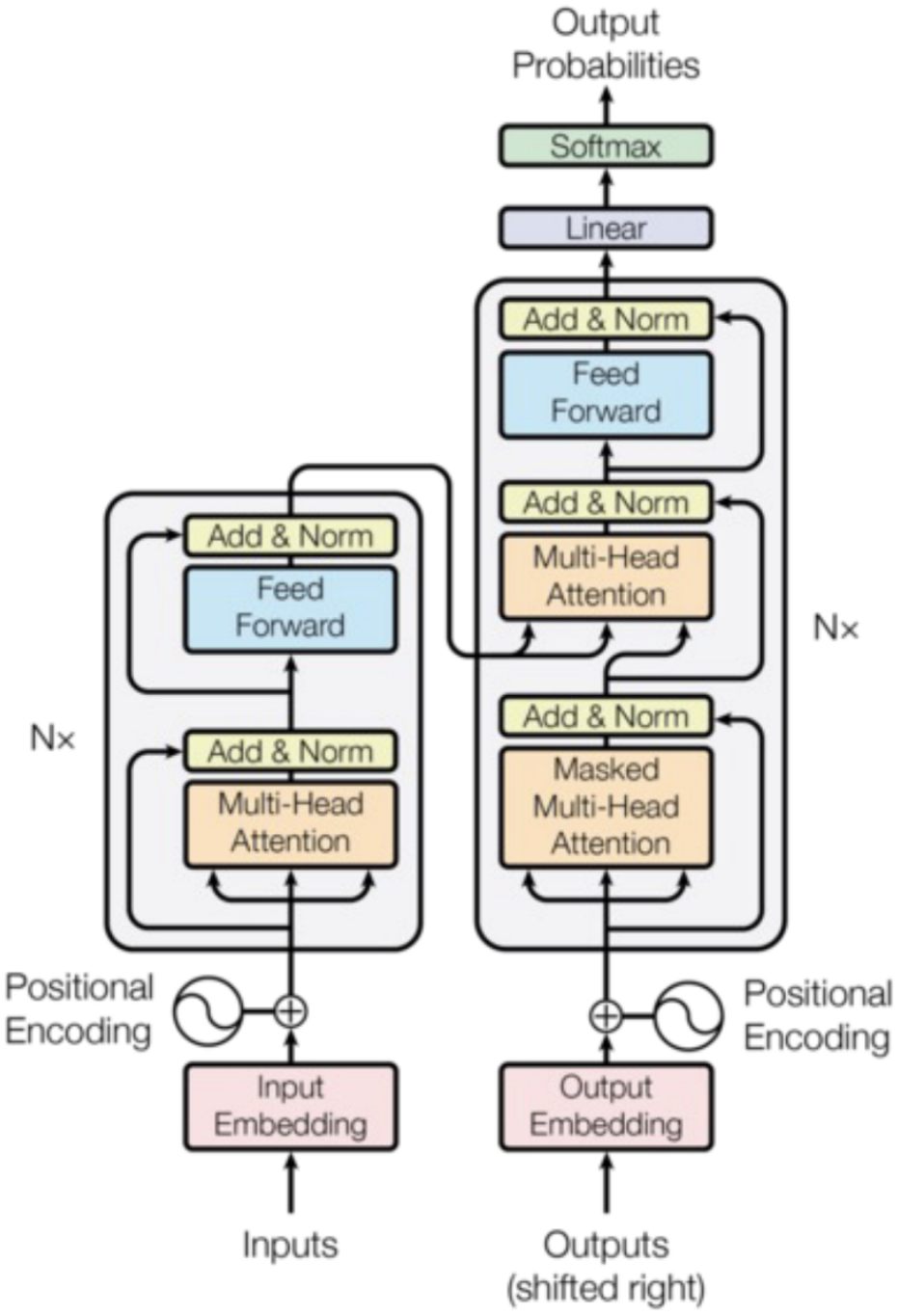


Figure 8.1: A multi-head transformer

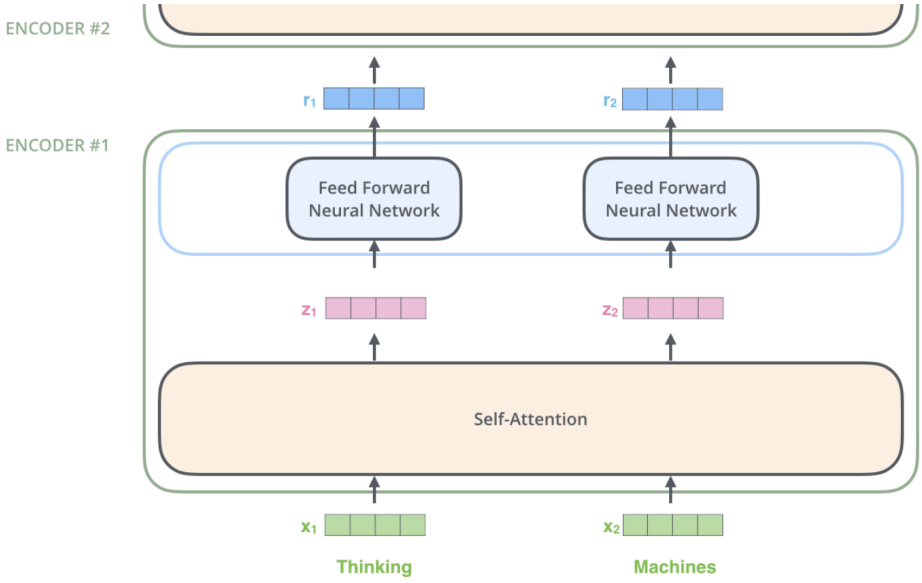


Figure 8.2: Self attention layer in a transformer

$\boldsymbol{q}_i, \boldsymbol{k}_i$ must have the same length d. Then the layer computes a **score** between the query and the key:

$$a_i := \frac{1}{\sqrt{d}} \boldsymbol{q}_i \cdot \boldsymbol{k}_i$$

Finally the output of the layer is a mixture of $\boldsymbol{v}_i$s weighted by the softmax of a_i

$$\boldsymbol{z} := \sum_i \text{softmax}(a_1, \ldots, a_N)_i \boldsymbol{v}_i$$

We can represent the same calculation in matrix form, with the input matrix being $\boldsymbol{X} \in \mathbb{R}^{M \times N}$

$$\underbrace{\boldsymbol{Q}}_{\text{Query}} := \boldsymbol{X}\mathbf{W}_q, \quad \underbrace{\boldsymbol{K}}_{\text{Key}} := \boldsymbol{X}\mathbf{W}_k, \quad \underbrace{\boldsymbol{V}}_{\text{Value}} := \boldsymbol{X}\mathbf{W}_v$$

Then we can compute the score

$$\boldsymbol{Z} := \text{softmax}\left(\underbrace{\frac{1}{\sqrt{d}} \boldsymbol{Q}\boldsymbol{K}^\intercal}_{\text{Attention Score}}\right) \boldsymbol{V}$$

Multi-head Attention is the same as executing many instances of this process in parallel.

Question: Can we take mean pool over the outputs of the heads of the multi-head attention?

Mean pool ignores ordering, so it would not be very useful.

Excursion:

Aniva: I think this would be easier to see using the Einstein summation convention, where if an index appears on only one side, it is assumed to be summed over. Suppose x^i_j is the feature, $q^{i,\mu}$ the query, k^i_μ the key, and v^i_ν the value, then one self-attention layer is

$$q^{i,\mu} := (W_q)^{j,\mu} x^i_j, \qquad k^i_\mu := (W_k)^j_\mu x^i_j, \qquad v^i_\nu := (W_v)^j_\nu x^i_j$$

and one attention layer is

$$z^{i'}_\nu := \mathrm{softmax}_i \left(\frac{1}{\sqrt{d}} q^{i,\mu} k^i_\mu \right)_{i'} v^{i'}_\nu$$

You can replicate this with the Python package `einops`.

Similar to transformers, GNNs also take in a sequence of vectors (in no particular order) and output a sequence of embedding. The difference is when GNN uses message passing, transformer uses attention.

8.2 Self-Attention and Message Passing

Consider the attention output of just one token:

$$\boldsymbol{z}_1 := \sum_i \mathrm{softmax}_j(\boldsymbol{q}_j \cdot \boldsymbol{k}_j) \boldsymbol{v}_i$$

We can represent this as:

1. Compute message from j: $(\boldsymbol{v}_j, \boldsymbol{k}_j) := \mathsf{Message}(\boldsymbol{W}_v \boldsymbol{x}_j, \boldsymbol{W}_k x_j)$
2. Compute query from 1: $\boldsymbol{q}_1 := \boldsymbol{W}_q \boldsymbol{x}_1$
3. Aggregate all messages:

$$\mathsf{Aggregate}(\boldsymbol{q}_1, \{\mathsf{Message}(\boldsymbol{x}_i) : i\}) := \sum_i \mathrm{softmax}_j(\boldsymbol{q}_j \cdot \boldsymbol{k}_j) \boldsymbol{v}_i$$

Question: Does this assume the ordering of tokens is not important.

Yes, and we will have to fix it.

This shows Self-attention can be written as message and aggregation – i.e., it is a GNN! Every node receives information from every other node. In other words, the graph is fully connected.
At the moment the transformer model we have is oblivious to tokens, since the attention mechanism and weighting function softmax ignore index ordering. To fix this issue we need **positional encoding**. For NLP tasks, each token $\boldsymbol{x}_i$ in the input is concatenated with a vector $\boldsymbol{p}$ indicating its position e.g. $[\cos i/N, \sin i/N]$. Then we use the concatenated vector $[\boldsymbol{x}, \boldsymbol{p}]$ as the input to an attention layer instead of $\boldsymbol{x}$.

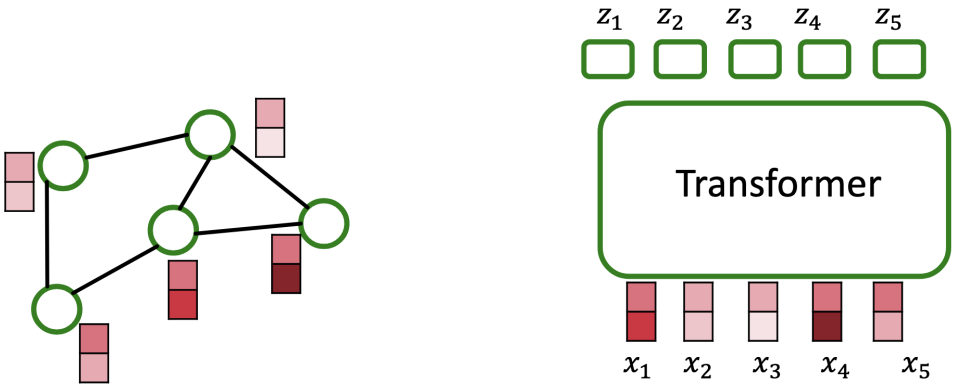


Figure 8.3: Node features on a graph can be used as the inputs features for a transformer

8.3 A New Design Landscape for Graph Transformers

How do we input a graph into a transformer? We need to understand the key components of a transformer: (1) tokenization, (2) positional encoding, and (3) self-attention, and make graph versions of them. A graph transformer must takes the inputs (1) Node features (2) Adjacency information (3) Edge features.
For (1), a sensible choice is the node features. The main problem about this is we completely lose adjacency information. We can add this information back by adding positional encoding based on adjacency information.
A few options for positional encoding exist:

1. Relative distances based on random walks from anchor set: This is particularly strong for tasks that require counting cycles. Pick anchor vertices $v_1, \ldots, v_l$, and each vertex v gets the position encoding

$$\boldsymbol{p} := [d(v, v_1), \ldots, d(v, v_l)]$$

   Relative distances useful for position-aware task but not structural aware tasks.

2. Laplacian Eigenvector Positional Encoding: Each graph has a **Laplacian matrix**

$$\boldsymbol{L} := \boldsymbol{D} - \boldsymbol{A}$$

   Diagonal degree matrix — Adjacency Matrix

   Several Laplacian variants that add degree information differently. Laplacian matrix captures the matrix structure, and its eigenvectors inherit this structure. Eigenvectors with small eigenvalue correspond to global structure, and large eigenvalue correspond to local symmetries. We can calculate the eigen-decomposition of the Laplacian matrix $\boldsymbol{L} = \boldsymbol{\Sigma}\boldsymbol{\Lambda}\boldsymbol{\Sigma}^\intercal$, and use $\boldsymbol{\Sigma}$ (with the indices order [data, feature]) as the position encoding.

   A simple task such as whether a graph has a cycle, could be done with a GNN with the assistance of the Laplacian eigenvectors.

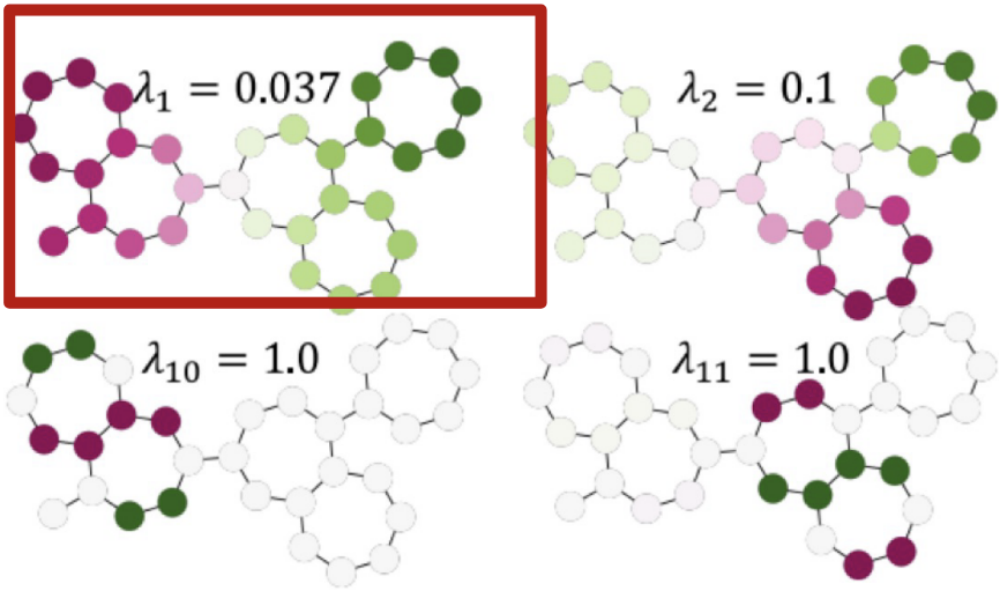


Figure 8.4: Examples of eigenvectors in a Laplacian matrix

Question: Does the Laplacian only encodes structure or both structure and position?

Both. See Figure 8.4.

Finally, we need to find out how to embed the edge features $\boldsymbol{x}_{i,j}$. The only place in the attention mechanism where pairs of vertices come in is during the computation of the attention scores $[a_{i,j}] = \boldsymbol{Q}\boldsymbol{K}^\intercal$. We can adjust this based on the edge features $a_{i,j} \mapsto a_{i,j} + c_{i,j}$. i.e.

$$c_{i,j} := \begin{cases} \boldsymbol{w}_e^\intercal \boldsymbol{x}_{i,j} & \text{edge exists between } i,j \\ \sum_k \boldsymbol{w}_{e_k}^\intercal \boldsymbol{x}_{e_k} & \text{path } e_1,\dots,e_n \text{ exists between } i,j \end{cases}$$

8.4 Positional Encodings for Graph Transformers

Laplacian eigenvectors are not the best that we can do. They have structure that we have been ignoring. For example, if $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{L}$, then $c\boldsymbol{v}$ for any $c \neq 0$ is also an eigenvector of $\boldsymbol{L}$. The iterative algorithms for computing eigenvectors will produce eigenvectors with random signs. This arbitrarity affects the prediction of our GNN.
A simple idea is to randomly flip the signs of eigenvectors during training. The problem is an exponential number of sign choices exist. We can alternatively design an NN which is invariant to sign flips.

Theorem 8.1. *For $f : \mathbb{R}^n \to \mathbb{R}^n$, $f(x) = f(-x)$ if and only if there exists g such that $f(x) = g(x) + g(-x)$*

Proof. If $f(x) = g(x) + g(-x)$ then $f(-x) = g(-x) + g(x)$.
Conversely, define $g = f/2$, then $g(x) + g(-x) = f(x)/2 + f(-x)/2 = f(x)$. □

Hence we can use a neural network structured with the structure.

$$f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) := \mathsf{Aggregate}(\mathsf{g}(\boldsymbol{x}_i) + \mathsf{g}(-\boldsymbol{x}_i))$$

This is known as **SignNet**.

Theorem 8.2. *If f is sign invariant, there are functions g, h such that*

$$f(x_1, \ldots, x_n) = h(g(x_1) + g(-x_1), \ldots, g(x_n) + g(-x_n))$$

With this, we have the full structure of positional encoding for GNN:

1. Compute eigenvectors $\boldsymbol{\Sigma}$
2. Get eigenvector embeddings using SignNet
3. Concatenate SignNet embeddings $\boldsymbol{p}_i$ with feature vectors $\boldsymbol{x}_i$
4. Pass through main GNN/Transformer
5. (Training) Back-propagate gradients to train SignNet and Prediction models jointly.

9 Machine Learning with Heterogeneous Graphs

So far, we have only handled graphs with one edge type. In this section we describe learning algorithms on *heterogeneous graphs*.

9.1 Heterogeneous Graphs

Many real world datasets are naturally described as heterogeneous graphs, graphs whose nodes and edges are of more than one type. For example, in a publication graph, we could have *paper* nodes and *author* nodes, and the edges could be differentiated into *cite* edges and *like* edges.

A **heterogeneous graph** is a graph $G := (V, E, \tau, \phi)$, where

- V is the set of nodes $v \in V$.
- E is the set of edges $e \in E$.
- τ is the map of **node types** $\tau(v) : v \in V$
- ϕ is the map of **edge types** $\phi(e) : e \in E$

The **relation type** for edge e is

$$r(u, v) = (\tau(u), \phi(u, v), \tau(v))$$

Moreover, for edge type r, we define the **r-neighbourhood** of $v \in V$ to be

$$N_r(v) := \{u \in N(v) : \phi(u, v) = r\}$$

We could treat types of nodes and edges as features. For example, we could encode the one-hot feature $[1, 0]$ for author nodes and $[0, 1]$ for paper nodes.

When do we need a heterogeneous graph?

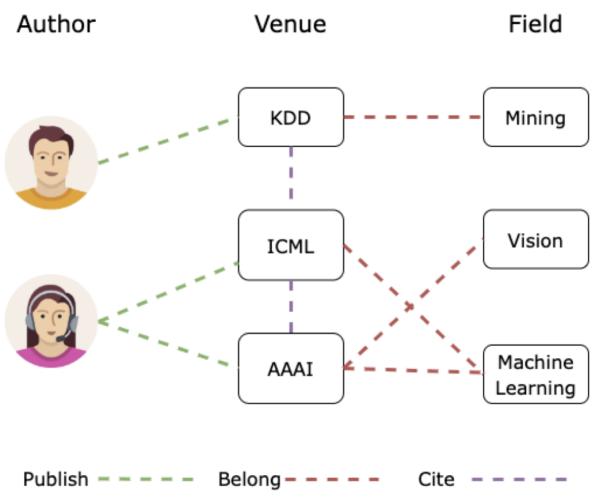


Figure 9.1: People, conferences, and publications in academia can be represented by a hetereogeneous graph.

- Different nodes/edges have different shapes of features
- We know different relation types represent different types of interactions.

Heterogeneous graphs are a more expressive class of graphs, but it comes with drawbacks such as more computation overhead and harder implementation. There are ways of converting a heterogeneous graph to a homogeneous graph.

Question: Can we have multiple edges of different types between two nodes

Yes.

9.2 Relational GCN

We shall extend Graph Convolutional Networks (GCN) so they can operate on heterogeneous graphs.

- Directed Graph $G := (V, \overrightarrow{E})$ with one relation:

  In a directed graph, we could only pass messages along the direction of edges:

$$\boldsymbol{h}_v^{(l)} := \sigma\left(\sum_{u \to v} \mathsf{W}^{(l)} \frac{\boldsymbol{h}_u^{(l-1)}}{\deg_{\text{in}} v}\right)$$

  Activation: σ; Message: $\mathsf{W}^{(l)} \frac{\boldsymbol{h}_u^{(l-1)}}{\deg_{\text{in}} v}$; Aggregation: $\sum_{u \to v}$

- In a graph with multiple relation types, different neural network weights $\mathsf{W}_{R(e)}$ could be used on different edge types:

$$\boldsymbol{h}_v^{(l)} := \underbrace{\sigma}_{\text{Activation}} \left(\underbrace{\sum_{u \to v}}_{\text{Aggregation}} \underbrace{\mathsf{W}_{R(u \to v)}^{(l)} \frac{\boldsymbol{h}_u^{(l-1)}}{\deg_{\text{in}} v}}_{\text{Message}} \right)$$

Question: We would expect some degree of correlation in the edge relations of real-world data. Could we choose whether to use RGCN or convert the graph to a homogeneous one based on correlations in the data?

In practice choose the simpler homogeneous graph models first.

Relational Graph Convolutional Newtork (**RGCN**) introduces a set of neural networks for each relation type on the heterogeneous graph $G := (V, T, R, E)$:

$$\boldsymbol{h}_v^{(l)} := \sigma \left(\sum_{r \in R} \sum_{u \in \underbrace{N^r(v)}_{r\text{-Neighbourhood}}} \underbrace{\frac{1}{c_{v,r}}}_{r\text{-degree } c_{v,r} := |N_r(v)|} \mathsf{W}_r^{(l)} \boldsymbol{h}_u^{(l-1)} + \mathsf{W}_0^{(l)} \boldsymbol{h}_v^{(l-1)} \right)$$

In message-aggregation form:

- Message:

$$\boldsymbol{m}_{u,r}^{(l)} := \frac{1}{c_{v,r}} \mathsf{W}_r^{(l)} \boldsymbol{h}_u^{(l)}$$
$$\boldsymbol{m}_v^{(l)} := \mathsf{W}_0^{(l)} \boldsymbol{h}_v^{(l)}$$

- Aggregation:

$$\boldsymbol{h}_v^{(l+1)} = \sigma\Big(\sum_{u \in N(v)} \boldsymbol{m}_{u,R(u \to v)}^{(l)} + \boldsymbol{m}_v^{(l)} \Big)$$

Each relation has L matrices $\mathsf{W}_r^{(1)}, \ldots, \mathsf{W}_r^{(L)}$. The size of each $\mathsf{W}_r^{(l)}$ is $d^{(l+1)} \times d^{(l)}$. In total this leads to rapid growth of the numbers of parameters w.r.t. the number of relations, so overfitting may become an issue. Two methods of regularisation exist:

- **Block Diagonal Matrices**: Use B block diagonal matrices for W_r:

$$\mathsf{W}_r := \begin{bmatrix} \mathsf{W}_{r,1} & \cdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \mathsf{W}_{r,B} \end{bmatrix}$$

  Limitation: Only nearby neurons could interact via W_r. This reduces the number of parameters from $d^{(l+1)} \times d^{(l)}$ to $\frac{d^{(l+1)}}{B} \times \frac{d^{(l)}}{B}$

- **Basis/Dictionary Learning**: Share weights across different relations.

  We could represent the matrix of each relation W_r as a linear combination of *learnable* basis matrices

  $$\mathsf{W}_r := \sum_{b=1}^{B} a_{r,b} \mathsf{V}_b$$

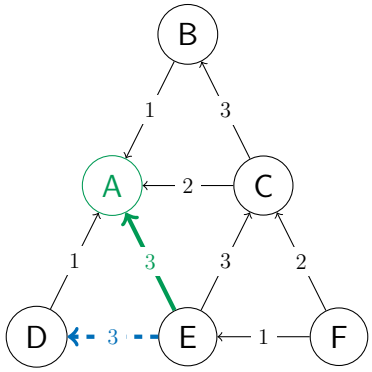
  where V_bs are shared across relations and $(a_{r,b})_{b=1}^{B}$ are learnable scalars. When the number of relation types is large, this requires only $|R|\, B + d^{(l+1)} \times d^{(l)}$ parameters per layer rather than $|R| \times d^{(l+1)} \times d^{(l)}$ parameters per layer, which is an improvement.

Question: How to choose the basis matrices?

One way is to train homogeneous networks and use their weights as bases, or you could make them random.

Example

Consider the following graph:



Two different tasks could be executed on this graph using an RGCN:

- Label Prediction: If there are $|T|$ different node labels possible, the final layer prediction head $\boldsymbol{h}_{\mathsf{A}}^{(L)}$ for node A encodes the probability of classes for node A.
- Link Prediction: Assume (E, A) is a training supervision edge and all other edges are training message edges, we could score the RGCN on (E, A) using a *relation-specific score function*

  $$f_r(\boldsymbol{h}_{\mathsf{E}}, \boldsymbol{h}_{\mathsf{A}}) := \boldsymbol{h}_{\mathsf{E}}^{\mathsf{T}} \mathsf{W}_r \boldsymbol{h}_{\mathsf{A}} \qquad (\mathsf{W}_r \in \mathbb{R}^{d \times d})$$

  This is specific to one relation and the model is tasked with determining the probability of there being an edge between E and A. A more general task head can determine the *existence* of an edge on top of the edge's category, but in many real world cases (e.g. drug discovery, paper citation) the type of the edge is already known.

  A negative supervision edge can be created by perturbing the tail of (E, A) to become for example (E, B). Note that negative supervision edges should not be training message edges, so (E, C) cannot be a negative training supervision edge.

Then, the output from f_1 is used as a logit in a sigmoid function ($\sigma(f_1(\mathsf{E}, \mathsf{A}))$) which is trained against the ground truth label of the supervision edges.

The edges not in the training message and supervision edges can then be ranked by their logits $r_1, \ldots, r_m$. The performance of the model can be measured in a variety of metrics:

- Hits@k: Fix a value k, and $|\{r_i < k : y_i\}|$, where y_i is the ground truth label, can be used to measure the number of relevant hits in the top k ranked edges.
- Reciprocal Rank: $\sum_i y_i / r_i$ (a higher score is better)

Question: How can we generate negative edges for link prediction if the graph is very dense.

The question of link prediction is ill defined when the graph is dense since if there are no places to insert new edges, prediction of new edges cannot fail.

Question: How can we choose the negative edge examples

We can corrupt the tail of existing edges so they point to somewhere not linked in the original graph.

Question: For negative sampling, How do you account for the imbalance of edge types?

The prediction of edges $(u, r, e)$is equivalent to sampling the marginal distribution $v|u, r$ given u, r fixed. This marginal distribution is not affected by the imbalance of edges.

9.3 Heterogeneous Graph Transformer

Graph Attention Networks (GAT) can be adapted for heterogeneous graphs. Introducing a new neural network for each relation type is too expensive for attention.
Heterogeneous Graph Transformer (**HGT**)[8] uses **scaled dot-product attention** from transformer, where the attention weights are defined as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) := \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\intercal}{\sqrt{d_k}}\right)\boldsymbol{V}$$

where $\boldsymbol{Q}$ is the **query**, $\boldsymbol{K}$ is the **key**, and $\boldsymbol{V}$ is the **value**. All 3 matrices have the shape $(\text{batchsize}, d_k)$.
Recall that when applying GAT to a homogeneous graph,

$$\boldsymbol{h}_v^{(l)} := \mathsf{Aggregate}\{\alpha_{v,u} \cdot \boldsymbol{m}_u^{(l)} : u \in N(v)\}$$

Attention weights

[8] Hu et al. WWW '20

Without decomposition, the number of attention parameters can quickly overwhelm the model, since $|T|$ node types and $|R|$ relation types produce $|T|^2 |R|$ different weight matrices.

In **Heterogeneous Mutual Attention**, we decompose homogeneous attention to node and edge type-dependent attention mechanisms. Specifically the attention head is defined as a learned quadratic form

$$\alpha_{v,u} := \mathsf{k}^i_{\tau(u)} \left(\boldsymbol{h}_u^{(l-1)}\right)^\intercal \mathsf{W}^{\mathsf{att}}_{\phi(u \to v)} \mathsf{q}^i_{\tau(v)} \left(\boldsymbol{h}_v^{(l-1)}\right)$$

Note that the edge type directly parameterises $\mathsf{W}^{\mathsf{att}}_{\phi(u \to v)}$ and the node types parameterise $\mathsf{k}_{\tau(u)}, \mathsf{q}_{\tau(v)} : d^{(l-1)} \to d_k$.

Moreover, the message head is also decomposed into node and relation types:

$$\boldsymbol{m}_u^{(l)} := \mathsf{W}^{\mathsf{msg}}_{\phi(u \to v)} \, \mathsf{N}_{\tau(u)} \, \boldsymbol{h}_u^{(l-1)}$$

Weight for each edge type

Linear head for each node type

A layer of HGT is given by

Attention scores

Values V

$$\boldsymbol{h}_v^{(l)} := \mathsf{Aggregate}\left\{ \mathrm{softmax}\left(\frac{1}{\sqrt{d_k}} \alpha_{v,u} : u \in N(v)\right) \cdot \boldsymbol{m}_u^{(l)} : u \in N(v) \right\}$$

On the `ogbn-mag` benchmark to predict paper venues, HGT uses much fewer parameters even though the attention computation is expensive, but it performs better than R-GCN.

Question: What does $\oplus$ mean (in the slides)?

Aggregation.

Question: Why attention weights alone aren't enough to distinguish different relation types?

Attention weights don't take into account the different messages emitted by different relation types.

9.4 Design Space for Heterogeneous GNNs

So far the message aggregation function treats messages on equal footing. In **Heterogeneous Message Aggregation**, a different aggregation can be used for each relation type:

$$\boldsymbol{h}_v^{(l)} := \mathsf{Aggregate}^{(l)}_{r \in R}\left(\mathsf{Aggregate}^{(l)}_r(\{\boldsymbol{m}_u^{(l)} : u \in N_r(v)\})\right)$$

A common case is where the outer aggregation is concatenation and the inner aggregation is summation. Since the number of relations is fixed, this produces fixed sized aggregations.

$$\boldsymbol{h}_v^{(l)} := \bigoplus_{r \in R}^{(l)} \left(\sum_{u \in N_r(v)} \boldsymbol{m}_u^{(l)} \right)$$

where $\oplus$ is concatenation.

Model	Score	Embedding	S.	A.	I.	C.	M.
TransE	$-\|\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|\|$	$\boldsymbol{h},\boldsymbol{t},\boldsymbol{r}\in\mathbb{R}^k$		✓	✓	✓	
TransR	$-\|\|\boldsymbol{M}_r\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{M}_r\boldsymbol{t}\|\|$	$\boldsymbol{M}_r\in\mathbb{R}^{d\times k};\boldsymbol{h},\boldsymbol{t}\in\mathbb{R}^k;\boldsymbol{r}\in\mathbb{R}^d$	✓	✓	✓	✓	✓
DistMult	$\langle\boldsymbol{h},\boldsymbol{r},\boldsymbol{t}\rangle$	$\boldsymbol{h},\boldsymbol{t},\boldsymbol{r}\in\mathbb{R}^k$	✓				✓
ComplEx	$\mathrm{Re}\langle\boldsymbol{h},\boldsymbol{r},\bar{\boldsymbol{t}}\rangle$	$\boldsymbol{h},\boldsymbol{t},\boldsymbol{r}\in\mathbb{C}^k$	✓	✓	✓		✓

Table 10.1: A summary of various knowledge graph completion methods. The last 5 property columns represent the model's capability to model symmetry, antisymmetry, inverse, composition, and 1-to-n relationships, respectively.

10 Knowledge Graph Embeddings

Knowledge Graphs (**KG**) are heterogeneous graphs capture entities, types, and relationships.

- Nodes are entities
- Nodes are labeled by their types
- Edges between nodes capture relationships between entities

Real world applications: FreeBase, Wikipedia, YAGO, etc. They are used for serving information and question answering agents.
Common Characteristics:

- Massive: Millions of nodes and edges
- Incomplete: Many true edges are missing

Enumerating all the possible facts is impossible, but we can predict plausible but missing links.

10.1 Knowledge Graph Completion

The completion task is the following: Given (head, relation), predict missing tails. Note that this is a bit different from link prediction tasks. Edges in KGs are represented as triples (h, r, t).
To solve this task, we would like to have a *shallow* embedding for (h, r) pairs and embeddings for t's such that for plausible edges we have $\boldsymbol{e}_{(h,r)} \simeq \boldsymbol{e}_t$. Note that we do not use a GNN here. There are two design questions:

- How to embed (h, r)?
- How to define close-ness?

Relations in a knowledge graph can have the following properties:

- Symmetric: $r(h,t) \implies h(t,r)$. e.g. Family, Roommate

- Antisymmetric: $r(h,t) \implies \neg r(t,h)$
- Inverse: If $r_1(h,t) \implies r_2(t,h)$ e.g. Advisor/Advisee
- Composition/Transitivity: $r(x,y) \wedge r(y,z) \implies r(x,z)$

  e.g. Subsumes, logical consequence
- 1-to-n: $r(h,t_i) : i = 1,\ldots,n$ are all true.

  e.g. r is "student of".

Question: Are h, r, t vectors learned at the same time?

See the learning TransE algorithm. Each mini-batch updates the embedding. Training is done by updating the embeddings for losses in the criterion function f_r.

Question: How often do we update embeddings?

Knowledge graphs are very stable. Once they're created they're sort of fixed. The embeddings are frozen even if new knowledge comes in.

- **TransE**:

  Intuition: For a triple (h,r,t), $h + r \simeq t$ if the given fact is true. The scoring function is $f_r(h,t) := -\|\boldsymbol{h} + \boldsymbol{r} - \boldsymbol{t}\|$. TransE originated from an observation in Language models that the embeddings of words often have analogous relations

  $$(\text{Washington}) - (\text{USA}) \simeq (\text{Tokyo}) - (\text{Japan})$$

- **TransR**:

  Like TransE, but the entities are morphed by the vector $\boldsymbol{r}$ by a projection matrix $\boldsymbol{M}_r^{k\times d}$.

  Question: Can you make a generalised claim that symmetry implies 1-to-n?

  No. In general these are different properties that do not relate to each other.

  Question: Why use a linear transformation M_r in TransR instead of a non-linear transformation like a MLP?

  You could.

- **DistMult**: We use a bilinear modeling function $f_r(h,t) := \sum_i h_i r_i t_i$. This can be intuitively viewed as the cosine similarity between $\sum_i h_i r_i$ and $\boldsymbol{t}$.

  Intuitively, DistMult defines a hyperplane for each (h,r) pair.

Algorithm 10.1 TransE Training Algorithm

Require: Training set $S = \{(h, r, t)\}$, Entity set E, Relation set L, Margin γ, Embedding Dimension k, batch size b.
  $\boldsymbol{l} \sim U(-\frac{6}{\sqrt{k}}, +\frac{6}{\sqrt{k}})$ for each $l \in L$
  $\boldsymbol{l} \leftarrow \boldsymbol{l} / \|\boldsymbol{l}\|$ for each $l \in L$
  $\boldsymbol{e} \sim U(-\frac{6}{\sqrt{k}}, +\frac{6}{\sqrt{k}})$ for each $e \in E$ ▷ Initialise Relation and Entity embeddings
  loop
    $\boldsymbol{e} \leftarrow \boldsymbol{e} / \|\boldsymbol{e}\|$ for each $e \in E$
    $S_{\text{batch}} \sim \text{MiniBatch}(S, b)$ ▷ Sample Mini-batch
    $T_{\text{batch}} \leftarrow \emptyset$
    for $(h, r, t) \in S_{\text{batch}}$ **do**
      $(h', r, t') \sim \text{Corrupted-Triplet}(S'_{(h,r,t)})$
      $T_{\text{batch}} \leftarrow T_{\text{batch}} \cup \{((h, r, t), (h', r, t'))\}$ ▷ Add positive/negative sample pair
    end for
    $\mathsf{Loss} \leftarrow \sum_{(h,r,t),(h',r,t') \in T_{\text{batch}}} (\gamma + d(\boldsymbol{h} + \boldsymbol{l}, \boldsymbol{t}) - (\boldsymbol{h}' + \boldsymbol{l}', \boldsymbol{t}))_+$ ▷ Contrastive Loss
    Update embeddings using $\nabla \mathsf{Loss}$
  end loop

- **ComplEx**: Uses a similar bilinear modeling function but instead with $\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t} \in \mathbb{C}^k$. $f_r(h, t) = \text{Re} \sum_i h_i r_i \overline{t_i}$.

  This allows modeling of asymmetric relations due to the asymmetry introduced by the conjugate

- **RotatE**: TransE in a complex space

There is not a general embedding that works for all KGs. Use a table to select models.

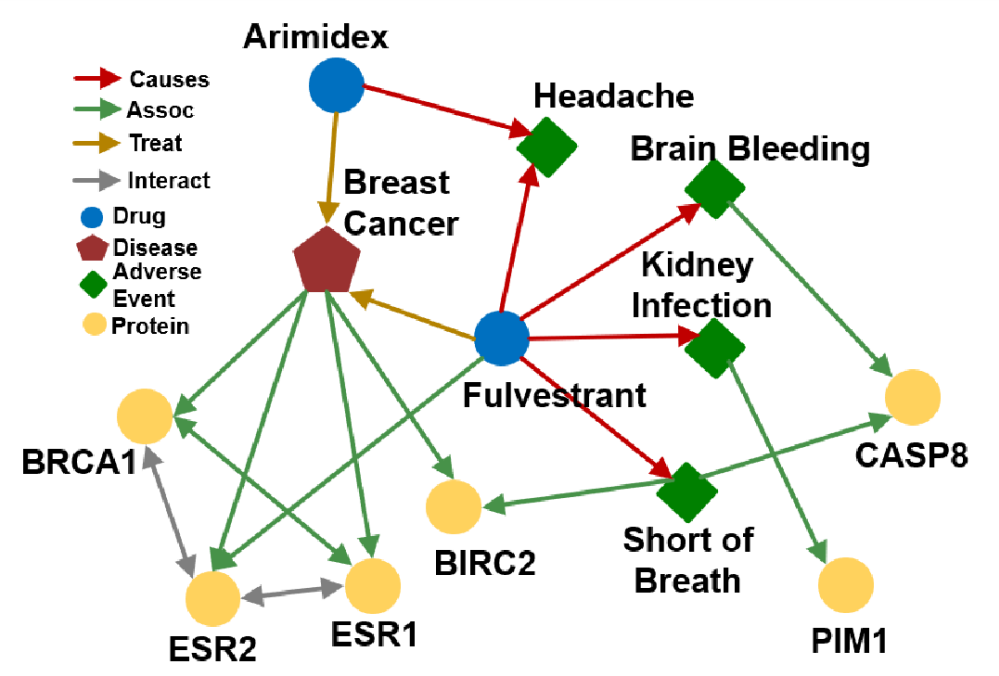


Figure 11.1: Example knowledge graph from biomedicine

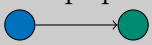
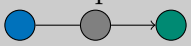
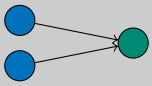
Query Type	Example Natural Language and Query
One-hop queries 	What adverse event is caused by Fulvestrant? (e:Fulvestrant, (r:Causes))
Path queries 	What protein is associated with the adverse event caused by Fulvestrant? (e:Fulvestrant, (r:Causes, r:Assoc))
Conjunctive queries 	What is the drug that treats breast cancer and caused headache? (e:BreastCancer, (r:Treatedby)), (e:Migraine, (r:CausedBy))

Table 11.1: Examples of queries on knowledge graphs

11 Reasoning in Knowledge Graphs

Multi-Hop Reasoning refers to answering a complex query on an incomplete, massive knowledge graph. The 1-hop reasoning problem is simply KG completion:

- KG Completion: Is the link (h, r, t) in KG?
- 1-Hop Reasoning: Is t the answer to (h, r)?

Multi-Hop Reasoning generalises this notion by allowing intermediate steps in the reasoning chain. A n-hop path query q can be represented by

$$q := (v_a, (r_1, \ldots, r_n))$$

Anchor entity — Path relations

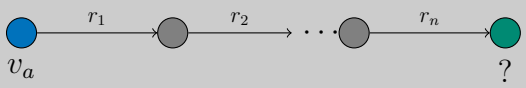


Figure 11.2: Query plan of q

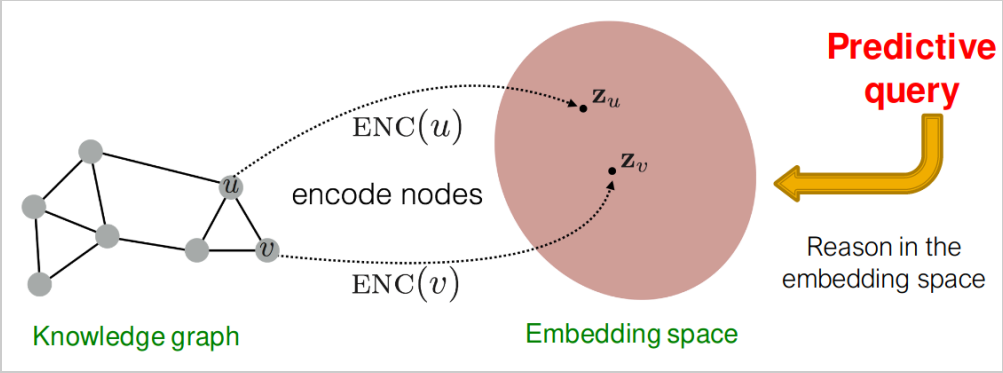


Figure 11.3: Learning to reason in latent space

The **answer** to q in graph G is denoted $[\![q]\!]_g$.
Answering queries seems easy: Just traverse the graph. However knowledge graphs are incomplete and known. Due to this incompleteness, one is not able to identify all the answer entities.
Can we first do KG completion and then traverse the completed probabilistic KG? No, since the probabilistic graph is a dense graph and the time complexity of traversing such a graph is exponential in the number of query path length L.
A solution of this **predictive query** problem would be able to answer arbitrary queries while implicitly accounting for the missing information.

11.1 Answering Predictive Queries on Knowledge Graphs

The key idea is to embed nodes and relations in a graph and learn to reason in the space of embeddings.
Recall that TransE is a method used for Knowledge Graph completions, where the link (h, r, t) is considered positive if $\boldsymbol{h} + \boldsymbol{r} \simeq \boldsymbol{t}$. We can generalise this by using multiple relations. Define the **embedding** of a query q to be

$$\boldsymbol{q} := \boldsymbol{v}_a + \boldsymbol{r}_1 + \cdots + \boldsymbol{r}_n$$

Then, we can query for node embeddings close to $\boldsymbol{q}$ and label them as the answer to the query.

Question: Does the order of the path not matter in TransE since it is a vector addition?

TransE would not be able to model ordered paths.

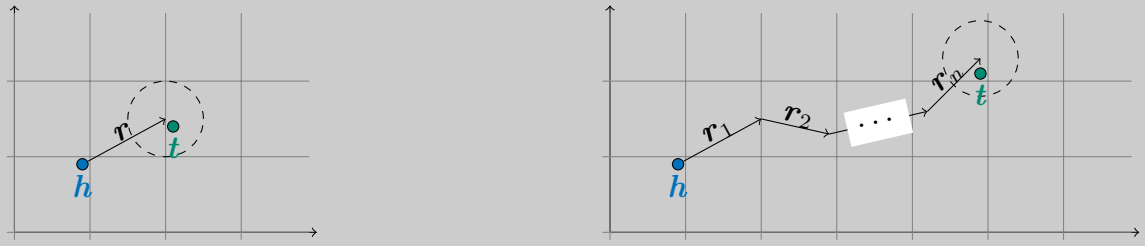


Figure 11.4: TransE for completion and query answering

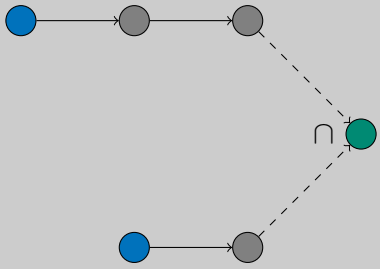


Figure 11.5: A conjunctive query

Since TransE can handle compositional relations, it can handle path queries by translating multiple relations into a composition. TransR, DistMult, ComplEx cannot handle composition and hence cannot be easily extended to handle path queries.
Can we answer more complex queries with *conjunction* operation? If a conjunctive query $q = (q_1, q_2)$, then $[\![q]\!]_G = [\![q_1]\!]_G \cap [\![q_2]\!]_G$.

11.2 Query2Box

We have two problems to solve in conjunctive queries:

- Each intermediate node represents a set of entities. How can we represent it?
- How do we define the intersection operation in latent space when two queries have to be simultaneously satisfied.

In **Query2Box**, each query is embedded as a hyperrectangle (box). A benefit of using boxes is that the intersection of boxes is well-defined. Each box is represented by a **centre** and an **offset** Some other methods use other geometric shapes, e.g. cones, beta distributions.

Question: Can we use the TransE approach and embed the paths separately and take the intersection?

Yes, but representing an entire set with a single point is difficult and we can easily come up with counterexamples.

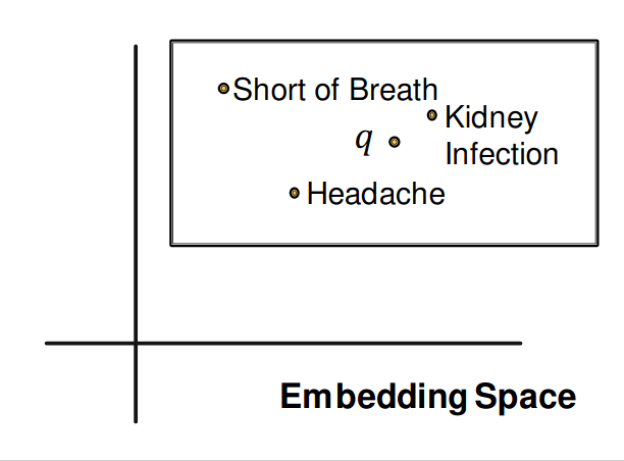


Figure 11.6: Box Embeddings for biomedicine that encloses some answer entities

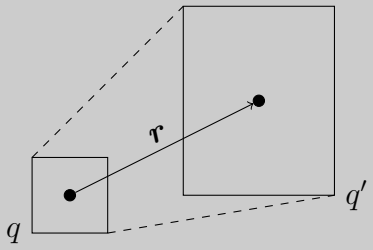


Figure 11.7: Projection operator acting on query q with relation r

Settings

Let d be the out degree, $|V|$ be the number of entities, and $|R|$ be the number of relations.

In Query2Box:

- Entity embeddings ($d\,|V|$ parameters): Entities are seen as zero-volume boxes
- Relation embeddings ($2d\,|R|$ parameters): Each relation takes a box and produces a new box. This is the **projection operator** $\mathcal{P}$ and maps Box $\times\, R \mapsto$ Box. There is one projection operator for each relation type, where the centre and offset are moved by

$$\begin{aligned} \text{centre}(q') &:= \text{centre}(q) + \text{centre}(r) \\ \text{offset}(q') &:= \text{offset}(q) + \text{offset}(r) \end{aligned}$$

- Intersection operator $\mathcal{I}$: Inputs are boxes and output is a box. The centre of the new box should be "close" to the centres of the input boxes, and the offset should *shrink* since the intersected box is smaller than the size of all previous boxes. It does not have to be a strict geometric intersection.

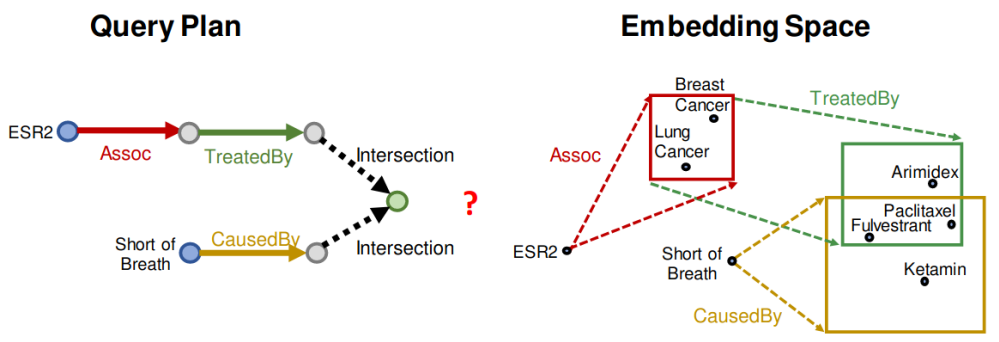


Figure 11.8: Conjunctive query with Query2Box

We can define the result of intersection to be

$$\text{centre}\, q_\cap = \sum_i \boldsymbol{w}_i \odot \text{centre}(q_i)$$

(Hadamard (Elementwise) Product)

$$\boldsymbol{w}_i := \frac{\exp(\mathsf{f}_c(\text{centre}\, q_i))}{\sum_j \exp(\mathsf{f}_c(\text{centre}\, q_i))}$$

$$\text{offset}\, q_\cap := \min\{\text{offset}\, q_1, \ldots, \text{offset}\, q_n\} \odot \sigma\left(\mathsf{f}_o(\text{offset}\, q_1, \ldots, \text{offset}\, q_n)\right)$$

(Ensure shrinking; Sigmoid function)

where $\mathsf{f}_c, \mathsf{f}_o$ are neural networks. f_c assigns attention scores to each box.

Question: Why do we have to learn the intersection operator? Why not just use a fixed one?

Learned operator allows more expressivity for the model.

- Score function $f_q(v)$ (captures relevancy of v w.r.t. q)

  We can define this to be the sum of in-distance and out-distance:

$$d_{\text{box}}(q, \boldsymbol{v}) := d_{\text{out}}(q, \boldsymbol{v}) + \alpha \cdot d_{\text{in}}(q, \boldsymbol{v})$$

(Hyperparameter $0 < \alpha < 1$; Distance outside of the box; Distance inside of the box)

  and the score function $f_q(v)$ would be defined as the inverse distance of v to q: $f_q(v) := -d_{\text{box}}(q, \boldsymbol{v})$.

  The intuition is that the distance inside the box is shrunk relative to the outside.

Question: Does it matter which norm we use to measure the distance of node embedding to box centre?

The distance used in the slides is Manhattan (L^1) distance. It is very natural since balls in L^1 norm are boxes. Other distances can be used as well.

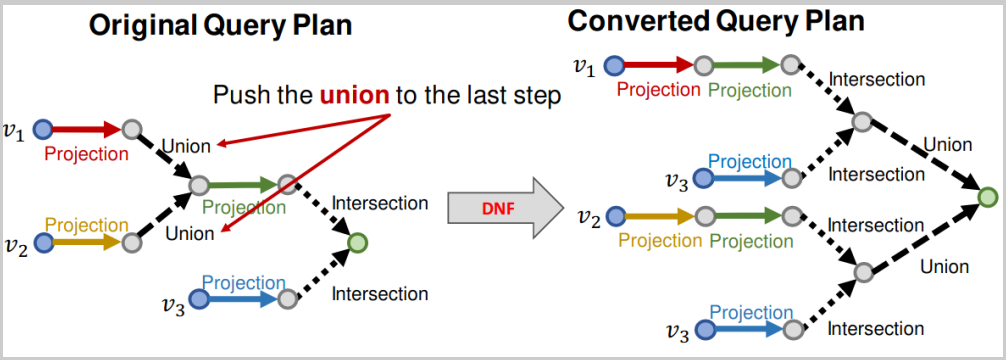


Figure 11.9: Converting a query to disjunctive normal form where unions are done at the last step

Question: How can we modify negative relations as a box?

If a query contains a negation it is not a conjunctive query. This would require things like Beta distributions. See the paper BetaE.

Question: Can we answer disjunctive queries?

The union of boxes is no longer a box, so this can't be done directly, but
You can write a query into *conjunctive normal form* and perform the union at the last step.

Question: How does a box handle the curse of dimensionality?

There is no way around the curse of dimensionality. It can be mitigated by only using tens of dimensions instead of hundreds in real world cases.

Conjunctive queries and disjunction is called **Existential Positive First-Order** (**EPFO**) queries. We will call them **and-or queries**.
Given d queries $q_1, \ldots, q_d$, for any subset $Q \subseteq \{q_1, \ldots, q_d\}$, we can form the query $\bigvee_{q \in Q} q$. We would like the question answering model to include points belonging to $q \in Q$ but exclude the points belonging to $q \notin Q$. When $q_1, \ldots, q_d$ have non-overlapping answers, a dimensionality of $\Theta(d)$ is needed to handle all OR queries.
For arbitrary real world queries, this number d is often very large, so we cannot embed and-or queries in low dimensional space. A solution to this is to leave the union operation to the very last step. This is the **disjunctive normal form** of the query, and any query can be written in the form $q = q_1 \vee \cdots \vee q_n$.
We can use the following metric to measure the distance of an entity embedding to a Disjunctive Normal Form (DNF) query:

$$d_{\text{box}}(q, \boldsymbol{v}) = \min\{d_{\text{box}}(q_i, \boldsymbol{v}) : i = 1, \ldots, n\}$$

Question: Is there a case where knowledge is dynamic or changing over time? How would we deal with that?

Yes. Retrain the embeddings when the knowledge graph gets an updated.

11.3 Training Query2Box

Similar to KG completion, the training goal is for any query embedding $\boldsymbol{q}$, maximise the score $f_q(v)$ for positive answers $v \in [\![q]\!]$ and minimise the score $f_q(v')$ for negative answers $v' \notin [\![q]\!]$. This involves minimising the objective on training graph G

$$\ell := \mathbb{E}_{q \sim \text{Query}(G), v \in [\![q]\!], v' \notin [\![q]\!]} [\, -\log \sigma(f_q(v)) - \log(1 - \sigma(f_q(v'))) \,]$$

Sample over queries q — KL Divergence/Logistic Loss

The queries q are generated from *query templates*. A query template outlines the topological structure of a query. Generating a grounded query from a template involving tracing back from the answer node of the query template and grounding each question edge backwards.[9]

Question: If we have a language based KG where words can have multiple meanings, yet our graph has only 1 embedding for each node, how can we deal with this?

Knowledge graphs should be unambiguous, so the ambiguity should be dealt with outside and not inside.

Question: Do we care about the intermediate step nodes in a query?

In some cases we would not care, but this would be interesting to study what are the intermediate representations.

Question: What if instead of one box, we use beam search to avoid sparsity.

Good idea would be interesting to try out.

12 Fast Neural Subgraph Matching and Counting

Subgraphs are the building blocks of networks. They have the power to discriminate and characterise networks. For example, functional groups are the building blocks of organic molecules.

[9] See the SMORE paper for detail on this subject.

12.1 Subgraphs and Motifs

Settings

Let $G = (V, E)$ be a graph.

- A **node-induced subgraph** $G' = (V', E')$ is a subgraph such that $V' \subseteq V$ and $E' = \{(u, v) \in E : u, v \in V'\}$.
- A **edge-induced subgraph** $G' = (V', E')$ is a subgraph such that $E' \subseteq E$ and $V' = \{v \in V : \exists u.(v, u) \in E'\}$

Question: How do we define the boundary of a subgraph?

Right now we are not worried about the boundary of a subgraph. We will expand on the definition later.

How do we express the statement "G_1's topology is contained in G_2"? We can use *isomorphism*. Two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ are **isomorphic** if there is a bijection $f : V_1 \to V_2$ such that

$$(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$$

f is the **(graph) isomorphism**. Finding the mapping is a computation problem that we do not know if it belongs to NP-hard, but we also do not know any polynomial time algorithm for it.
G_2 is **subgraph-isomorphic** to G_1 if some (node or edge induced) subgraph of G_2 is isomorphic to G_1. We also commonly say G_1 is a subgraph of G_2. Determining subgraph isomorphism is NP-hard.
A **network motif** is a recurring, significant pattern of interconnections.

- Pattern: Small (node-induced) subgraph
- Recurring: Found many times

  How to define frequent?
- Significant: More frequent than expected than e.g. randomly generated graphs

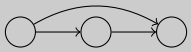
  What is the distribution of such random graphs?

Motifs help us understand how graphs work and make prediction based on presence or lack of presence in a graph dataset.

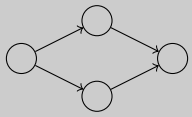
Question: Is it fine for subgraphs to have overlaps during counting?

Yes.

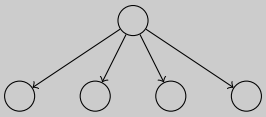
Let G_Q be a small graph and G_T be a target graph. There are two definitions of frequency of G_Q in G_T.



(a) Feed-Forward Loop: Found in networks of neurons, where they neutralise "biological noise"



(b) Parallel Loop: Found in food webs



(c) Single-Input Module: Found in gene control networks

Figure 12.1: Common motifs in graphs

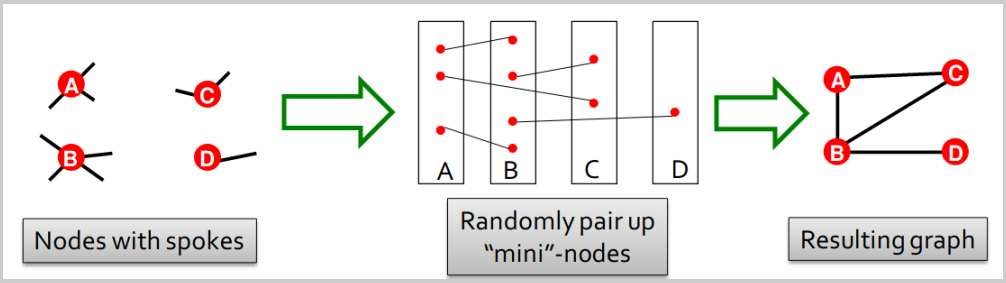


Figure 12.2: Configuration Model

- **Graph-level Frequency**: The frequency of G_Q in G_T is the number of unique subsets of nodes V_T for which the subgraph of G_T induced by V_T is isomorphic to G_Q. Frequency can get very large due to permutations.
- **Node-level Frequency**: In addition to the above we have an anchor $v \in V_Q$ such that the number of nodes $u \in V_T$ for which *some* (counted as 1) subgraph of G_T is isomorphic to G_Q and the isomorphism maps u to v.

If the dataset contains multiple graphs, we can treat the dataset as a giant graph G_T with disconnected components corresponding to individual graphs.
To define significance, we need to have a null-model (point of comparison). Subgraphs that occur in a real network much more often than in a random network have functional significance.
Methods of generating random graphs:

- A **Erdős-Rényi (ER) Random Graphs** is a graph G_n, p where n is the number of nodes and the edges (u, v) appears iid. with probability p.

  The distribution of $\deg v : v \in G_{n,p}$ is binomial, which can be unrealistic for social graphs.

- A **configuration model** graph: Based on a real graph G^{real}, gather its nodes' degree sequence and create nodes with "spokes" corresponding to its degree. Then randomly pair up nodes. This results in a graph G^{rand} with the same degree sequence as G^{real}.

- A **switching** graph: Start from a given graph G and repeat a switching step $Q \cdot |E|$ times, where Q is large (e.g. 100). The switching step selects a pair of edges $(a, b), (c, d)$ at random and exchange their endpoints, giving $(a, d), (c, b)$. Exchange only if no multi-edges or self-edges are generated.

This creates a randomly wired graph with the same node degrees as the original.

Question: If you are rearranging the edges, does it also preserve clustering coëfficient or centrality?

When you generate such a random graph you have to decide what properties not to preserve. The random switching process destroys the local structure so it has no reason to preserve clustering coëfficient.

This is the slowest method.

Motifs are over-represented in a real graph compared to a random graph. The number of motifs in a graph can be measured with statistical tools to evaluate its significance.
We can use statistical methods to evaluate the occurrence significance of a motif. The **Z-score**, defined as

$$Z_i := \frac{N_i^{\text{real}} - \bar{N}_i^{\text{rand}}}{\text{Std}\, N_i^{\text{rand}}}$$

(N_i^{real}: #(Motif i) in graph G^{real}; $\bar{N}_i^{\text{rand}}$: Average #(Motif i) in random graphs; N_i^{rand}: Random variable #(Motif i) in random graphs)

measures the significance. The **network significance profile** (**SP**) is a vector of normalised Z-scores:

$$\boldsymbol{SP}_i := Z_i / \sqrt{\sum_j Z_j^2}$$

The SP vector emphasises relative significance of subgraphs. It is important for comparison of networks of different sizes. Generally, larger graphs display somewhat larger Z-scores.
For each subgraph, the Z-score metric is capable of classifying the subgraph "significance", where negative (resp. positive) values indicate under- (resp. over-)representation.
Variations on the motif concepts:

- Extensions: Directed/Undirected, Coloured/Uncoloured, Temporal/Static
- Variations on the concept: Different frequency concepts, significance concepts, unde-representation (anti-motifs), null models.

Question: How do we choose the motif we use the analyse?

Usually choose subgraphs up to a certain size.
Or if there is domain knowledge, we could use the method described in the next section.

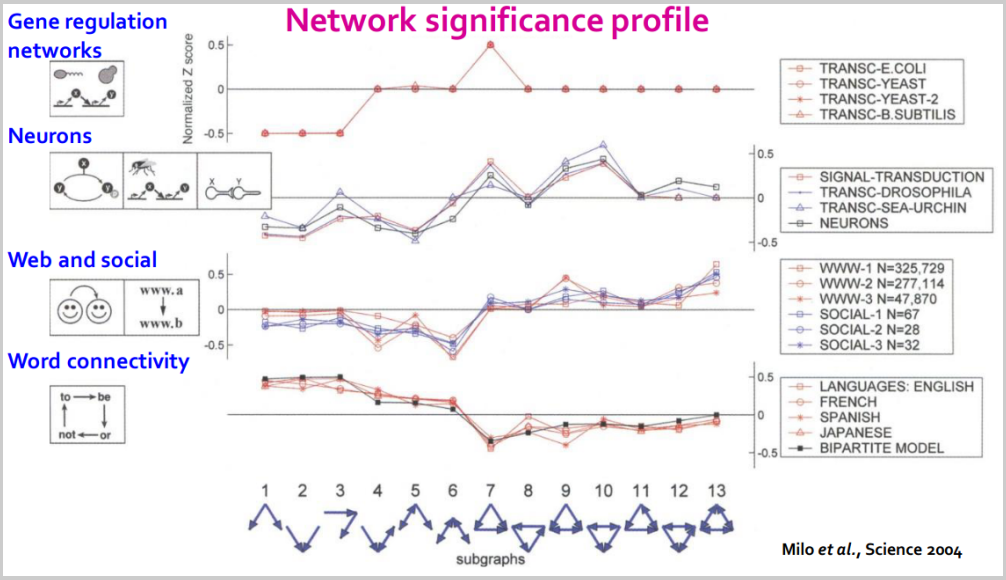


Figure 12.3: Examples of ***SP***: Networks from the same domain have similar significance profiles

12.2 Neural Subgraph Representations

Settings

Subgraph matching: Given a large target graph (can be disconnected) G_T and query graph (connected) G_Q, how can we decide if a G_Q is a subgraph in G_T?

A GNN can be used to predict subgraph isomorphism. We are going to work with *node-anchored* definition of frequency. We wish to generate embedding from the target anchored neighbourhood and the query graph, and the comparison between the embeddings lead to a binary label. The intuition of this method is to exploit the geometric structure of the embedding space to capture properties of subgraph isomorphism.

Question: So far we have only used Euclidean manifold embeddings. Can the subgraphs be embedded into non-Euclidean manifolds?

Would be cool to investigate. There have been experiments of embeddings in hyperbolic space. See. Dr. Chris Ré's research.

The algorithm first should decompose the input graph G_T into neighbourhoods. Then each neighbourhood is embedded into the embedding space, and comparison is executed on this embedding space to determine if G_Q is isomorphic to a subgraph of the neighbourhood. The benefits of using anchored node-level frequency definition is that the embedding of vectors naturally translate to embeddings of anchored subgraphs.
We impose a **partial order** on the embedding space. We define $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ to satisfy $\boldsymbol{u} \leq \boldsymbol{v}$ if $u_i \leq v_i$ for all i. It is trivial to check that $\leq$ defined on $\mathbb{R}^n$ is a partial order, and therefore follows transitivity. If the GNN's output embeddings are in such a way that the

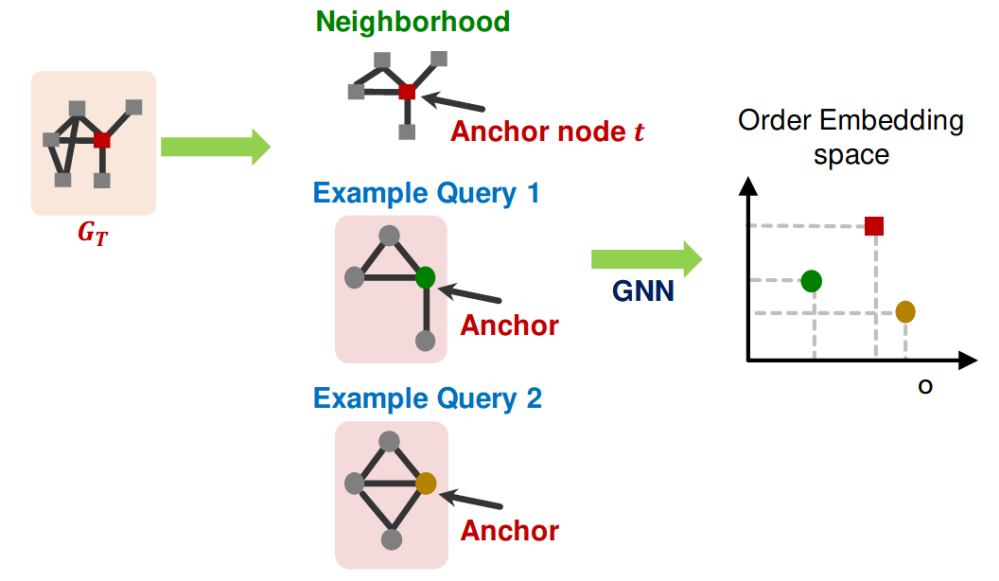


Figure 12.4: By comparing the embedding, we find (Query 1) $\leq G_T$ but (Query 2) $\not\leq G_T$.

subgraph isomorphism relation is mapped to $\leq$ in embedding space, we can quickly determine subgraph embedding by directly comparing the embeddings. Moreover, we force the embedding to be non-negative.

1. For each node v in G_T, Obtain a k-hop neighbourhood around the anchor v (e.g. using BFS)
2. Apply (1) to G_Q to obtain the neighbourhoods
3. Embed neighbourhoods using a GNN.

Question: Is the embedding of the subgraph the same as the embedding of the anchor node?

Yes

Question: How do we choose the anchor of the query?

The user chooses anchor i.e. its task specific.

Question: If my query's diameter is greater than the depth of the GNN, do we have trouble capturing the query?

Yes.

Question: How do we handle real life cases with big queries?

We can several form decompositions of the network based on the structure of the query.

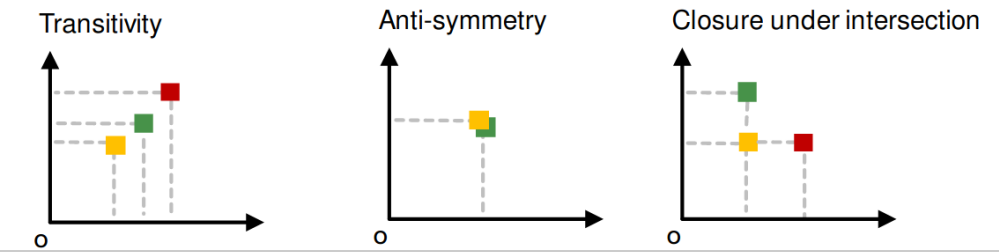


Figure 12.5: Order satisfies transitivity, anti-symmetry, and closure under intersection.

Question: Can we break a big query into smaller anchored queries and find those smaller queries in the engine?

This is an interesting problem to research.

How can we design a loss function to ensure GNN learns the ordering? We design loss functions based on the *order constraint*:

$$\forall i.\ z_q[i] \leq z_t[i] \iff G_Q \subseteq G_T$$

Embedding Dimension; Query Embedding; Target Embedding; Subgraph relation

trained with **max-margin loss**, i.e. the penalty is the square of the amount of violation of the order constraint.

$$E(G_q, G_t) := \sum_{i=1}^{D} \max(0, z_q[i] - z_t[i])^2$$

To learn such embeddings, we generate training examples (G_q, G_t) such that $G_q \subset G_t$ with probability $1/2$, and we minimise:

$$L(G_q, G_t) := \begin{cases} E(G_q, G_t) & G_q \subseteq G_t \\ \max(0,\ \alpha - E(G_q, G_t)) & G_q \not\subseteq G_t \end{cases}$$

Margin hyperparameter $\alpha > 0$

Question: Why is the loss capped at α?

Because if the reward for negative examples can be arbitrarily large, the network may be incentivised to push them infinitely far. In this case no learning can occur.

At each iteration, generate random samples (G_T, G_Q^+, G_Q^-):

- Positive: Sample induced subgraph $G_Q \subseteq G_T$ using BFS Sampling
  1. Initialise $S := \{v\}, V := \emptyset$
  2. Let $N(S)$ be the all neighbours of nodes in S. At every step, sample 10% of the nodes in $N(S) \setminus V$ and put them in S. Put the remaining nodes in V.
  3. After K steps, take the subgraph of G induced by S anchored at q.

Usually use $3-5$ layers of BFS. Trades off runtime and performance.

- Negative: Corrupt G_Q by adding/removing nodes/edges so its no longer a subgraph.

Question: What types of accuracy do you expect from this technique?

There will be some improvement from subgraph embedding methods. This is state of the art for motif counting.

Question: Do you see performance drops in size of the hops?

If you keep query size constant but increase the size of hops (bigger neighbourhoods), performance could drop.

Question: Can you find bijection between the subgraphs?

This method determines the correspondence of the anchors.

During inference, embed anchored query G_Q and target G_T graphs. Output whether the query is a node-anchored subgraph of the target using the predicate $E(G_Q, G_T) < \epsilon$.

12.3 Mining Frequent Motifs

Generally, finding the most frequent size-k motifs requires solving two challenges:

- **Enumerating** all size k subgraphs.
- **Counting** number of occurrences for each subgraph type.

Just knowing if a subgraph exists in a graph is computationally difficult problem. The feasibility of traditional motif counting is relatively small (3 to 7)

Settings

In the problem of **frequent motif mining**, we wish to find among all possible graphs of k nodes, the r graphs with the highest frequency in G_T.

The **SPMiner** algorithm:

1. Decompose input graph G_T into neighbourhoods
2. Embed neighbourhoods into an order embedding space

   The key benefit of order embedding is that we can quickly predict the frequency of a given subgraph G_Q.
3. Estimate the frequency of each motif by counting the set

Randomly sampled node-anchored neighbourhoods

$$S_E := \{ \boldsymbol{z}_Q : \boldsymbol{z}_Q \leq \boldsymbol{z}_N, G_N \subseteq G_T \}$$

Embedding of motif G_Q — Supergraph region

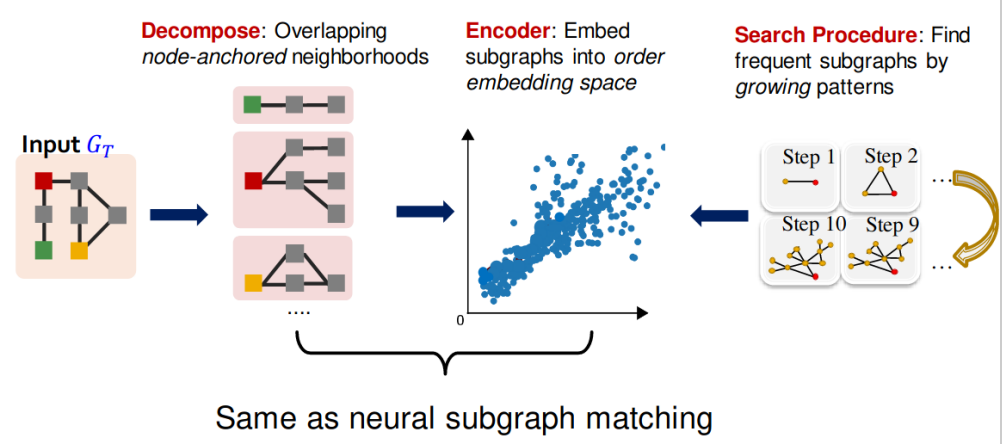


Figure 12.6: SPMiner algorithm

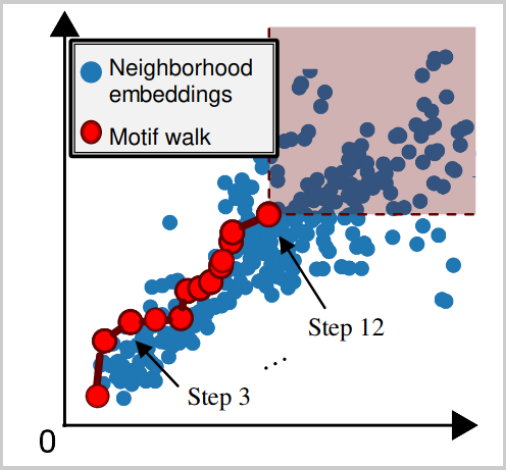


Figure 12.7: Motif Walk and the supergraph region representing the complement of total violation

4. Execute **motif walk**, a kind of *beam search*:

   - Start by randomly picking a starting node u in the target graph G_T. Set $S := \{u\}$
   - Grow a motif by iteratively chooisng a neighbour in G_T of a node in S and add that node to S. Grow the motif S to find larger frequent motifs. At each step, maximise $|S_E|$.

     This maximisation is a greedy procedure and the complement of S_E is the **total violation** of G_Q. The greedy heuristic adds the node that results in the smallest total violation.
   - Terminate upon reaching a desired motif size.

Question: In SPMiner algorithm, what is the goal of the network

Find top k common motifs of a given size. e.g. Out of graphs of size 20, which ones are the most common.

Question: Would exam questions be similar to homework?

We will release some sample exam questions. They will be less involved than the homework questions. No coding on exam questions.

Question: What is the origin of converting space of graphs to space of embeddings

That is the innovation of the entire deep learning field.

13 Label Propagation

Given a graph with labels on some nodes, how can we assign labels to other nodes on the graph? Node embeddings are a partial way to solving this problem.

Settings

Transductive (also called *semi-supervised*) Node Classification: We have labels $Y_v : v \in V$ of some nodes in $G = (V, E)$. We wish to predict the labels of other nodes in the graph.

Today we discuss an alternative framework: Label propagation. The intuition is that correlations exist in the networks, and connected nodes tend to share the same label. Behaviours of nodes are correlated across the links of the network.
There are two explanations of why behaviours of nodes in a network are correlated:

- **Homophily**: The tendency of individuals to associate and bond with similar others.
- **Influence**: Social connections can influence the individual charactersitics of a person.

We will look at three techniques:

- Label propagation:

  Directly propagate known labels to all the nodes

- Correct and smooth:

  First define a base predictor, then correct and smooth the predictions with label propagation

- Masked label prediction:

  Construct a self-supervised ML task, let graph ML model to propagate label information

13.1 Label Propagation

Label Propagation (**LP**) is the following algorithm: Propagate node labels across the network. Suppose $Y_v \in \{0, 1\}$. For labeled nodes, initialise $\hat{Y}_v := Y_v$, and for unlabeled nodes v, let $\hat{Y}_v := \frac{1}{2}$. Then we update all unlabeled nodes until convergence or a number of iterations is reached.

Probability of node having label c

$$P^{(t+1)}(Y_v = c) = \frac{1}{\sum_{(v,u)\in E} A_{v,u}} \sum_{(v,u)\in E} A_{v,u} \, P^{(t)}(Y_u = c)$$

Edge weights

This is iterated until convergence ($\epsilon > 0$ is a hyperparameter):

$$\left| P^{(t)}(Y_v) - P^{(t-1)}(Y_v) \right| \leq \epsilon, \qquad \forall v \in V$$

Question: In the homework, there are over-smoothing problems and bipartite graphs may never converge. Does the same issue happen here?

This algorithm also suffers from the non-convergence issue. We may need to add random noise, but this is a limitation of the algorithm.

Issues:

- Convergence may be very slow and not guaranteed.
- Label propagation does not use node attributes

Question: We say this is fast but slow to converge. How does it compare with GNN?

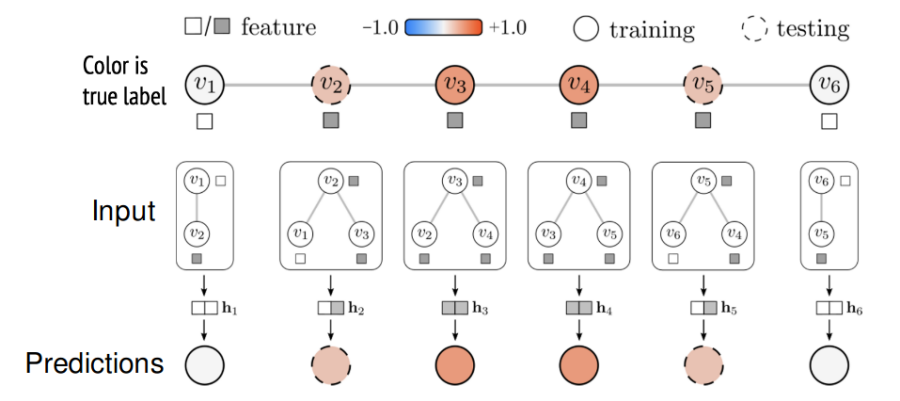
Fast here refers to one-step of label propagation. In most cases, it is still faster than training and applying a deep neural network.
However, inference of GNN is faster than label propagation.

13.2 Correct and Smooth

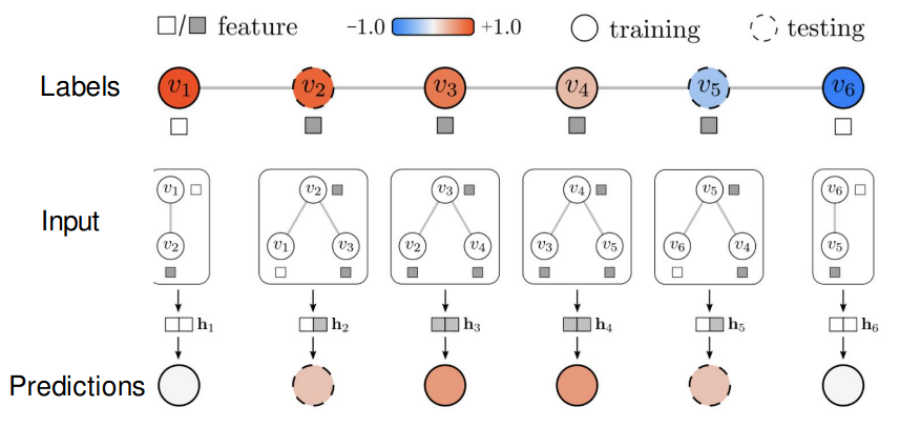
The problem with GNNs is that it fails catastrophically when the node features are *mildly predictive*.

Question: Why can't we include the labels in the messages instead of doing correct and smooth?

The motivation of Correct and Smooth is that its model agnostic, so it can make prediction with your favourite model. Directly leveraging the label is possible too.



(a) Indicative features



(b) Non-indicative features

Figure 13.1: GNN performed on two graphs with different labels and identical node features lead to very different performance since GNN does not take the labels into account. The resulting node embeddings do not have sufficient differentiation power.

The core idea of **Correct and Smooth**[10] is that we expect the *errors* of a base label predictor to be correlated along edges of the graph, so we should spread such uncertainty over the graph. We define the **diffusion matrix** to be

$$\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$$

where $\boldsymbol{D}$ is the diagonal degree matrix $D_{i,i} := \deg i$.

Theorem 13.1. *All the eigenvalues of $\tilde{\boldsymbol{A}}$ are in the range of $[-1, +1]$, and the maximum eigenvalue is always 1 with eigenvector $\boldsymbol{D}^{1/2}\boldsymbol{1}$ so the power of $\tilde{\boldsymbol{A}}$ is well-behaved for any K.*

Proof.

$$\tilde{\boldsymbol{A}}\boldsymbol{D}^{1/2}\boldsymbol{1} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}\boldsymbol{D}^{1/2}\boldsymbol{1} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{1} = \boldsymbol{D}^{-1/2}\boldsymbol{D}\boldsymbol{1} = \boldsymbol{D}^{1/2}\boldsymbol{1}$$

□

[10] See Zhu et al. ICML 2013 for details.

If i, j are connected, then

$$\tilde{A}_{i,j} = \frac{1}{\sqrt{d_i \cdot d_j}}$$

which takes into account the connectivity of both i and j.
We also add self edges to adjacency matrices, i.e. $A_{i,i} := 1$

1. Train *base predictor* which predicts soft labels (class probabilities) over all nodes. Labeled nodes can be used for train/validation data.
2. Apply base predictor to *all* (including the truth) nodes to obtain soft labels.
3. Correct and Smooth:
   - (**Correct**): The degree of the errors of the soft labels are biased. We need to correct for the bias.
     Compute the training errors of nodes, i.e.

     Ground one-hot label / Predicted soft label

     $$\boldsymbol{e}_v := \begin{cases} \boldsymbol{p}_v - \hat{\boldsymbol{p}}_v & v \text{ is labeled} \\ \boldsymbol{0} & v \text{ is unlabeled} \end{cases}$$

     Then, diffuse the training errors $\boldsymbol{E}^{(0)}$ along the edges.

     Hyperparameter

     $$\boldsymbol{E}^{(t+1)} := (1 - \alpha)\boldsymbol{E}^{(t)} + \alpha \tilde{\boldsymbol{A}} \boldsymbol{E}^{(t)}$$

     We add the scaled diffused training errors into the soft labels:

     $$\tilde{\boldsymbol{P}} := \hat{\boldsymbol{P}} + s\boldsymbol{E}^{(T)}$$

     This results in the **corrected labels**

     $$\boldsymbol{z}_v := \begin{cases} \boldsymbol{p}_v & v \text{ is labeled} \\ \tilde{\boldsymbol{p}}_v & v \text{ is unlabeled} \end{cases}$$

   - (**Smooth**): The predicted soft labels may not be smooth over the graph. We need to smoothen the labels.
     Assumption: Neighboring nodes tend to share the same labels.
     Diffuse $\boldsymbol{z}_v$ along the edges:

     $$\boldsymbol{Z}^{(t+1)} := (1 - \alpha)\boldsymbol{Z}^{(t)} + \alpha \tilde{\boldsymbol{A}} \boldsymbol{Z}^{(t)}$$

Question: What does the hyperparameter α control?

It controls the degree of homophily in the network

Question: Could I use it for graph prediction during training?

No because it needs some ground truth.
However, during training stage of graph prediction, a method like this could be used to diffuse the features if you do not trust the features, but this is a different paradigm.

Correct and Smooth:

- Significantly improves the performance of the base model.
- Outperforms the smooth-only baseline.
- Can be combined with GNNs

13.3 Masked Label Prediction

This is inspired by BERT objective in NLP. We treat labels as *additional features*. We concat the label matrix $\boldsymbol{Y}$ with the node feature matrix $\boldsymbol{X}$ and use the partially observed labels $\dot{\boldsymbol{Y}}$ to predict the remaining labels.

- Training: Corrupt $\dot{\boldsymbol{Y}}$ into $\tilde{\boldsymbol{Y}}$ by randomly masking a portion of the node labels to 0's. Then use $[\boldsymbol{X}, \tilde{\boldsymbol{Y}}]$ to predict the masked node labels.
- Inference: Employ all of $\dot{\boldsymbol{Y}}$ to predict the remaining unlabeled nodes.

Masked Label Prediction is also a self-supervised task like link prediction.

14 GNNs for Recommender Systems

Settings

Personalised recommender systems can be naturally modeled as a *bipartite* graph:

- Nodes are users U and items V
- Edges E connect users and items and indicate user-item interaction and are often associated with timestamp.

Given past user-item interactions, we wish to predict new items each user will interact with in the future. This is a link prediction problem. For each $u \in U, v \in V$, the model should generate a score $f(u, v)$ which ranks the recommendations. Since $|V|$ is large, evaluating every user-item pair (u, v) is infeasible. The solution to this problem is to break down the recommender into two stages.

- Candidate generation (cheap, fast):

  Retrieve about 1000 candidate items from embeddings of millions

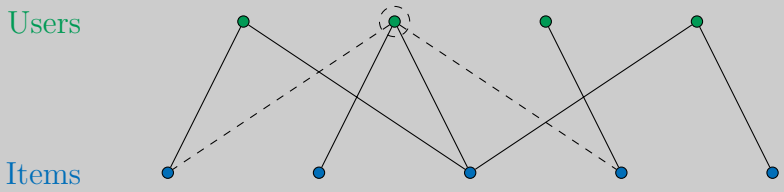


Figure 14.1: Recommendation system involves predicting possible user-item interaction edges given past edges

- Ranking (slow, accurate):

  Score/rank the items using $f(u, v)$.

For each user, we recommend the top K items. For recommendation to be effective, K (usually 10 to 100) has to be much smaller than the total number of items (usually billions). The goal is to include as many **positive items** (items that the user will interact with in the future) in the top-K recommended items as possible.
The evaluation metric is **Recall@K**, defined as follows. Let P_u be a set of positive items that the user will interact with in the future. Let R_u be a set of items recommended by the model. In the top-K recommendation, $|R_u| = K$. The recall is the intersection region in the Venn diagram of P_u and R_u:

$$\text{Recall@}K := \frac{|P_u \cap R_u|}{|P_u|}$$

The final Recall@K is the average of all user Recall@K values.

14.1 Recommender Systems: Embedding Based Models

To get the top-K items, the K itms with the largest scores for a given user u, excluding already-interacted items, are recommended. In an embedding-based model, each user and item correspond to embeddings. If $\boldsymbol{u} \in \mathbb{R}^D$ (resp. $\boldsymbol{v} \in \mathbb{R}^D$) is the embedding of user $u \in U$ (resp. $v \in V$), the score of (u, v) is a parameterised function $f_\theta(\cdot, \cdot) : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$.
The training objective function is to achieve high recall@K on seen (training sample) user-item interactions. This objective is not differentiable, so instead there are two widely used *surrogate loss functions* to enable gradient based training. They are differentiable and align well with the original training objective.

- **Binary Loss**: Define positive/negative edges. The set of positive edges E are observed, and the set of negative edges $E_- := \{(u, v) | (u, v) \notin E\}$

  The binary loss is

  $$-\frac{1}{|E|} \sum_{(u,v)\in E} \log \sigma(f_\theta(\boldsymbol{u}, \boldsymbol{v})) - \frac{1}{|E_-|} \sum_{(u,v)\in E_-} \log(1 - \sigma(f_\theta(\boldsymbol{u}, \boldsymbol{v})))$$

  during training, both sums are approximated with mini-batches of positive and negative edges.

Binary loss pushes scores of positive edges higher than those of negative edges. An issue is that the scores of all positive edges are pushed higher then the scores of all negative edges, but the recommendation for a user u is not affected by the recommendations of another user, so this unnecessarily penalises the model.

Question: Why can't we just use a hinge loss to plateau the loss when the ranking is correct?

The main goal of the BPR loss is to not compare scores for different users, because the scores across users do not matter in the rankings.

Question: What is the problem with the binary loss in practice?

In practice the binary loss is hard to optimise and its harder to make the model focus on the wrong things.

- **Bayesian Personalised Ranking** (**BPR**)[11]: For each user $u^* \in U$, define the **rooted** positive/negative edges as
  - $E(u^*) := \{(u^*, v) : (u^*, v) \in E\}$
  - $E_-(u^*) := \{(u^*, v) : (u^*, v) \in E_-\}$

  Training objective: For each user u^* we want the scores of rooted positive edges to exceed those of rooted negative edges. The loss is

$$L(u^*) := \frac{1}{|E(u^*)|\,|E_-(u^*)|} \sum_{(u^*,v_+)\in E(u^*)} \sum_{(u^*,v_-)\in E_-(u^*)} -\log\sigma(f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_+) - f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_-))$$

  The overall BPR loss is

$$L := \frac{1}{|U|} \sum_{u^*\in U} L(u^*)$$

  In a mini-batch, we sample a subset of users $\hat{U} \subseteq U$. For each $u^* \in \hat{U}$, we sample *one* positive item and a set of sampled negative items V_-. The mini-batch loss is then

$$\mathbb{E}_{\hat{U}\subseteq U, v_+, V_-}\left[\frac{1}{|\hat{U}|} \sum_{u^*\in\hat{U}} \frac{1}{|V_-|} \sum_{v_-\in V_-} -\log(\sigma(f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_+) - f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_-)))\right]$$

Question: Why sample many more negative edges than positive edges?

The problem is inherently imbalanced. Each user only interacts with a tiny subsets of all items, so a lot of negative examples are needed.

[11] The term "Bayesian" is not essential to the loss definition. The original paper Rendle et al. 2009 considers the Bayesian prior over parameters (essentially acts as a parameter regularization), which we omit here.

Question: How to select the negative edges?

See slides at the end. Just picking random edges makes it too easy for the model to learn. The candidate negative edge generator has to generate difficult examples to force the model to pick out the first hundreds out of billions.
We could introduce hard negative examples. We will see *distance based sampling* (evenly across embedding distances) to solve this issue.

Why do embedding models work? The core idea is **collaborative filtering**:
Recommended items for a user are related to preferences of many other similar users. When the embeddings are low dimensional, they cannot simply memorise all user-item interaction data, and they are forced to learn similarities between users and items to fit the data.

Question: Do we consider people who share accounts?

We model this heterogeneity outside. The splitting of one user node into multiple real users should be done before.

14.2 Neural Graph Collaborative Filtering

Conventional collaborative filtering model is based on shallow encoders. There are no user/item features, and use shallow encoders to embed each user u and item v. The score function is just the dot $f(u, v) = \boldsymbol{u} \cdot \boldsymbol{v}$. This model does not explicitly capture graph structure. The structure is only implicitly captured in the training objective.
In shallow encoders, only first-order graph structure (edges) is captured. Higher-order graph structures such as K-hop paths is not explicitly captured.
We want a model that explicitly captures graph structure and captures higher-order graph structure. GNNs are a natural approach to achieve this. In
Neural Graph Collaborative Filtering (**NGCF**), the initial non-graph-aware shallow embeddings are propagated through a GNN to create graph-aware embeddings.
For every user u and item v, set $\boldsymbol{h}_u^{(0)}, \boldsymbol{h}_v^{(0)}$ as the user/item's shallow embedding. Then we iteratively update node embeddings using neighbourhood embeddings:

$$\begin{cases} \boldsymbol{h}_v^{(k+1)} & := \mathsf{Combine}(\boldsymbol{h}_v^{(k)}, \mathsf{Aggregate}(\boldsymbol{h}_u^{(k)} : u \in N(v))) \\ \boldsymbol{h}_u^{(k+1)} & := \mathsf{Combine}(\boldsymbol{h}_u^{(k)}, \mathsf{Aggregate}(\boldsymbol{h}_v^{(k)} : v \in N(u))) \end{cases}$$

after K rounds of neighbour aggregation, we get the final user/item embeddings $\boldsymbol{u} \leftarrow \boldsymbol{h}_u^{(K)}, \boldsymbol{v} \leftarrow \boldsymbol{h}_v^{(K)}$. Score function is the inner product $\boldsymbol{u}^\intercal \boldsymbol{v}$.

14.3 LightGCN

NGCF jointly learns two kinds of parameters: Shallow user/item embeddings, and parameters for the GNN. The embeddings are already quite expressive. They are learned for every user and item node and the total number of parameters in them is $\mathrm{O}(ND)$ (where N is the number of nodes), where as the number of parameters in the GNN is only $\mathrm{O}(D^2)$. The GNN parameters may not be very essential.

We can simplify the GNN used in NGCF. The adjacency and embedding matrices of an undirected bipartite graph can be expressed as

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{R} \\ \boldsymbol{R}^\intercal & \boldsymbol{0} \end{bmatrix}, \qquad \boldsymbol{E} = \begin{bmatrix} \boldsymbol{E}_U \\ \boldsymbol{E}_V \end{bmatrix}$$

User embedding

Item embedding

Recall the diffusion matrix of Correct and Smooth (Section 13.2) method. Let $\boldsymbol{D}$ be the degree matrix of $\boldsymbol{A}$. Each layer of a GCN's aggregation can be written in the form

$$\boldsymbol{E}^{(k+1)} := \mathrm{ReLU}(\, \tilde{\boldsymbol{A}}\, \boldsymbol{E}^{(k)}\, \mathsf{W}^{(k)}\,)$$

Learnable linear transformation

Normalised Adjacency Matrix $\tilde{A} := \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$

LightGCN simplifies this by removing the ReLU non-linearity. Iterating the layers we see that

$$\begin{aligned} \boldsymbol{E}^{(K)} &:= \tilde{\boldsymbol{A}}\boldsymbol{E}^{(K-1)}\mathsf{W}^{(K-1)} \\ &= \tilde{\boldsymbol{A}} \cdots \tilde{\boldsymbol{A}}\boldsymbol{E}^{(0)}\mathsf{W}^{(0)} \cdots \mathsf{W}^{(K-1)} \\ &= \tilde{\boldsymbol{A}}^K \boldsymbol{E} \underbrace{(\mathsf{W}^{(0)} \cdots \mathsf{W}^{(K-1)})}_{\mathsf{W}} \end{aligned}$$

Removing the ReLU part significantly simplifies the GCN. The **LightGCN** algorithm applies $\boldsymbol{E} \leftarrow \tilde{\boldsymbol{A}}\boldsymbol{E}$ K times, and each multiplication diffuses the embeddings to their neighbours. The matrix $\tilde{\boldsymbol{A}}^K$ is dense and not stored in memory. Instead the above multiplication step is executed many times. We could also consider **multi-scale diffusion**:

$$\boldsymbol{E} := \sum_{i=0}^{K} \alpha_i \boldsymbol{E}^{(i)} = \sum_{i=0}^{K} \alpha_i \tilde{\boldsymbol{A}}^k \boldsymbol{E}^{(0)}$$

where $\alpha_0 \boldsymbol{E}^{(0)}$ acts as self connection. For simplicity, LightGCN uses $\alpha_k := 1/(k+1)$.

Question: Do we do all the matrix multiplication in memory?

Multiplying big matrices is not a problem. e.g. using MapReduce.
The bottleneck is storing the embeddings.

Question: Can we do SVD to aid with the matrix multiplication?

We can't do SVD because its cubic. CUR decomposition is fine but its beyond this course.

Intuitively, the simple diffusion propagation encourages embeddings of similar users and items to be similar. LightGCN is similar to GCN and C&S, except that self-loops are not added, and the final embedding is the average of layer embeddings.
LightGCN performs better than shallow encoders but are also more computationally costly. The simplification from NGCF leads to better performance.

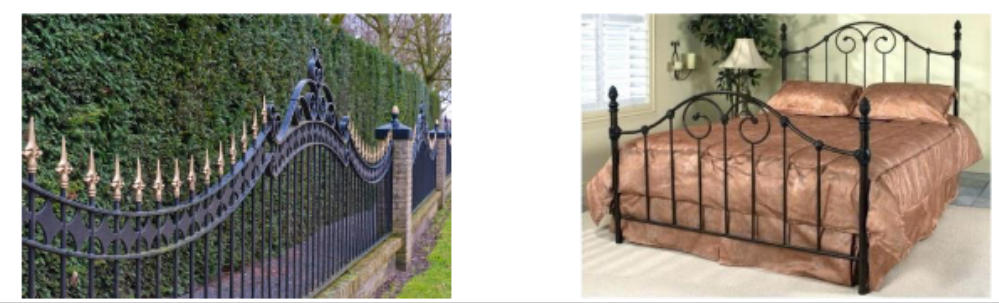


Figure 14.2: "Bed rail" may look like "Garden fence" but they are rarely adjacent in the graph

Question: Is averaging of embedding across layers a general approach not limited to recommender systems?

It is more general but it was originally developed for this purpose.

14.4 PinSAGE

The PinSAGE algorithm was developed for Pinterest to recommend pins and is the largest industry deployment of a GCN. Each pin embedding unifies visual, textual, and graph information. It works for fresh content and is available a few seconds after pin creation. The task of pin recommendation is to learn node embeddings z_i such that the distance of similar pins are shorter than the distance of dissimilar pins. e.g. $d(z_{\text{cake1}}, z_{\text{cake2}}) < d(z_{\text{cake1}}, z_{\text{sweater}})$. There are $1 + B$ repin pairs from related pins surface which captures semantic relatedness.
Graph has tens of billions of nodes and edges. In addition to the GNN model, PinSAGE paper introduces several methods to scale the GNN:

- **Shared negative samples across users**:

  Recall that in BPR loss, for each user $u^* \in \hat{U}$, we sample one positive item and a set of negative items V_-. Using more negative samples per user improves recommendation but is also expensive, where the number of computational graphs is $\left|\hat{U}\right| \cdot |V_-|$.

  The key idea is that the same set of negative samples can be used *across all users* in the mini-batch. This saves computational cost by a factor of $\left|\hat{U}\right|$.

- **Hard negative samples**:

  Industrial recommendation systems need extremely fine-grained predictions. Each user has 10 to 100 items to recommend while there are billions of items in total. A random sample from all items contain mostly easy negatives. We need a way to sample hard negatives to force the model to be fine grained. Hard negatives are obtained as follows:

  1. Compute personalised page rank (PPR) for user u

Figure 14.3: Examples of easy and hard negatives

2. Sort items in the descending order of their PPR scores
3. Randomly sample item nodes that are ranked high but not too high. e.g. 2000th to 5000th.

The paper Wu et al. 2017 describes sampling based on distances so the query-negative distance distribution is about $U[0.5, 1.4]$.

- **Curriculum learning**:

  Make the negative samples gradually harder in the process of training. At the nth epoch, we add $n-1$ hard negative samples. For each user node, the hard negatives are item nodes that are close but not connected to the user node in the graph.

- **Mini-batch training of GNNs**:

  This will be covered in a future lecture.

15 Deep Generative Models for Graphs

We want to generate realistic graphs using *graph generative models*, which we expect to be similar to real graphs. Its applications include

- Drug discovery, material design
- Social network modeling

We study graph generation because

- **Insights**: Understand the formulation of graphs
- **Predictions**: Predict how will the graph further evolve
- **Simulations**: Generate novel graph instances
- **Anomaly detection**: Detect if a graph is normal/abnormal

History of graph generation

1. **Properties of real-world graphs**: A successful Graph Generative Model should fit these properties.
2. **Traditional graph generative models**: Each come with different assumptions on the formulation process
3. **Deep graph generative models**: Learn the graph formulation process from data

In this lecture we will cover Deep Graph *decoders*: A model which produces a graph structure from an embedding.

15.1 Machine Learning for Graph Generation

Two types of generation tasks:

- **Realistic Graph Generation**: Generate graphs that are similar to a given set of graphs.
- **Goal-directed graph generation**: Generate graphs that optimise an objective or constraint.

A **Graph Generative Model** uses the data distribution $p_{\text{data}}(G)$ and learns the distribution $p_{\text{model}}(G)$ from which samples can be generated.

Excursion: Basics of Generative Models

Assume we want to learn a generative model from data points $\{x_i\}$.

- The *data distribution* $p_{\text{data}}(x)$ is not known to us, but we have samples $x_i \sim p_{\text{data}}(x)$.
- The *model* $p_{\text{model}}(x|\theta)$ is parameterised by θ and can be tractably sampled. We wish to make $p_{\text{model}}(x|\theta)$ and $p_{\text{data}}(x)$ similar.

To bring $p_{\text{model}}(x|\theta)$ close to $p_{\text{data}}(x)$, we use the *principle of maximum likelihood*, where we find θ such that the likelihood of drawing $x_i \sim p_{\text{model}}(x|\theta)$ is the greatest.

$$\theta^* := \arg\max_\theta \mathbb{E}_{x\sim p_c data}\left[\log p_{\text{model}}(x|\theta)\right] \simeq \arg\max_\theta \sum_i \log p_{\text{model}}(x_i|theta)$$

To sample from $p_{\text{model}}(x|\theta)$, there are a couple of approaches. The most common approach is to sample from a noise latent distribution $\boldsymbol{z} \sim N(\boldsymbol{0}, \boldsymbol{I})$, and transform the noise with a function f to obtain $\boldsymbol{x} := f(\boldsymbol{z}|\theta)$. The distribution of $\boldsymbol{x}$ is the pushforward of $N(\boldsymbol{0}, \boldsymbol{I})$. In deep generative models, $f(\cdot)$ is a neural network.
To generate a sequence, we can use **auto-regressive models**, where the chain rule is used to generate a join distribution of $x_1, \ldots, x_n$:

$$p_{\text{model}}(\boldsymbol{x};\theta) = \prod_{t=1}^{n} p_{\text{model}}(x_t|x_1, \ldots, x_{t-1};\theta)$$

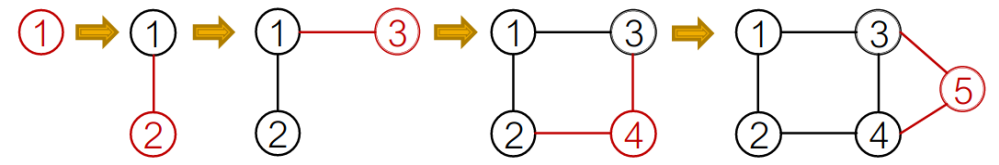


Figure 15.1: Generation process of a graph

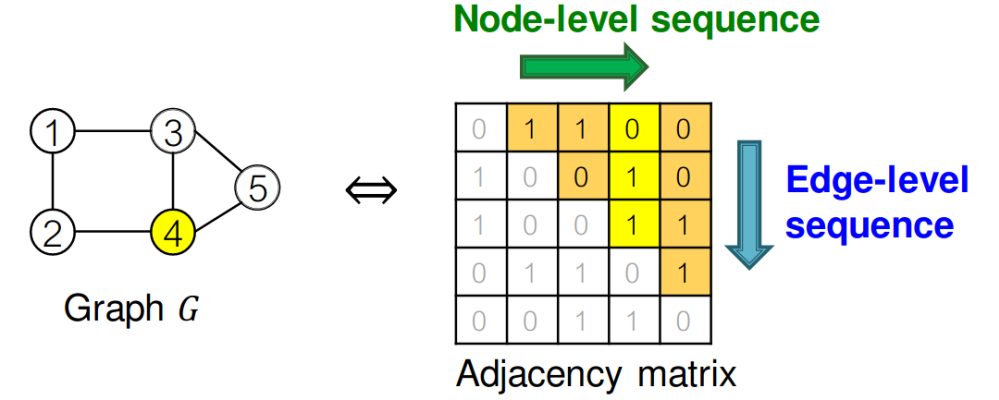


Figure 15.2: Generating a graph using GraphRNN is plotting its adjacency matrix column by column

15.2 Generating Realistic Graphs

A graph can be generated by successively adding nodes an edges. This is the core idea of **GraphRNN**. For a graph with node ordering π, we have the sequence $(S_i^\pi)_{i=1}^n$. The sequence S^π has two levels:

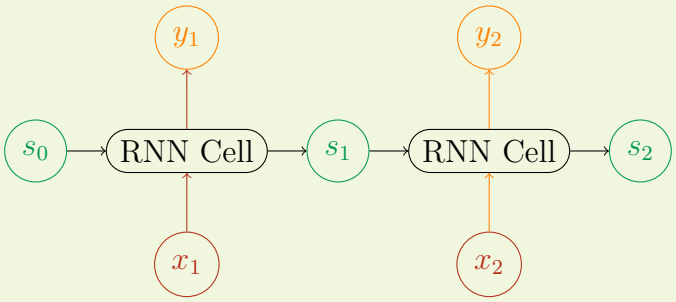
- Node-level generation: Add nodes, one at a time
- Edge-level generation: Add edges between existing nodes.

  For example $S_i^\pi = [S_{i,1}^\pi, \ldots, S_{i,m}^\pi]$ where each $S_{i,j}^\pi$ indicates whether an edge exists in between nodes i, j.

A graph and a node ordering is a sequence of sequences. The node ordering is randomly selected. This generates an adjacency matrix column by column. One drawback of this approach is the huge modeling capacity required to generate longer and longer columns We have transformed a graph generation problem into a sequence generation problem.

Excursion: Recurrent Neural Networks

Recurrent Neural Networks(**RNN**) is a neural network designed for sequential data. An RNN sequentially takes an input sequence to update its hidden states. The hidden states are passed between time steps of the RNN and the update is conducted via RNN cells.



An RNN cell takes inputs s_{t-1} (previous hidden state) and x_t, and outputs y_t and s_t (next hidden state). It is defined as

$$s_t := \sigma(W \cdot x_t + U \cdot s_{t-1}), \qquad y_t := V \cdot s_t$$

More expressive RNN cells such as GRU and LSTM have been developed to combat vanishing gradient problems.
An RNN can be used to generate sequences by feeding back $x_{t+1} := t_t$ (i.e. using the previous output as input). The sequence is initialised by a special `SOS` (start of sequence) symbol and `EOS` (end of sequence) symbol is emitted (as an extra RNN output) to signal halting the generation process.
An RNN modeled in this fashion is completely deterministic. To introduce stochasticity in the model, each $y_i = p_{\mathsf{model}}(x_t | x_1, \ldots, x_{t-1}; \theta)$ could be a categorical probability distribution and we can sample $x_{i+1} \sim y_i$.
A sequence generation RNN can be trained using **teacher forcing**, where the inputs to the RNN is forced to be the ground truth sequence and the loss is computed between the (shifted) ground truth sequence and RNN output.
RNNs are trained using **Backpropagation Through Time** (**BPTT**) which accumulates gradients across time steps.

GraphRNN has a node-level RNN and an edge-level RNN. Node-level RNN generates the initial state for edge-level RNN, and edge-level RNN sequentially predict if the new node will connect to each of the previous nodes. The edge-level RNN at node-level step i outputs a binary label $\hat{y}_j$ of whether nodes i, j have an edge, and it is trained using *binary cross-entropy*

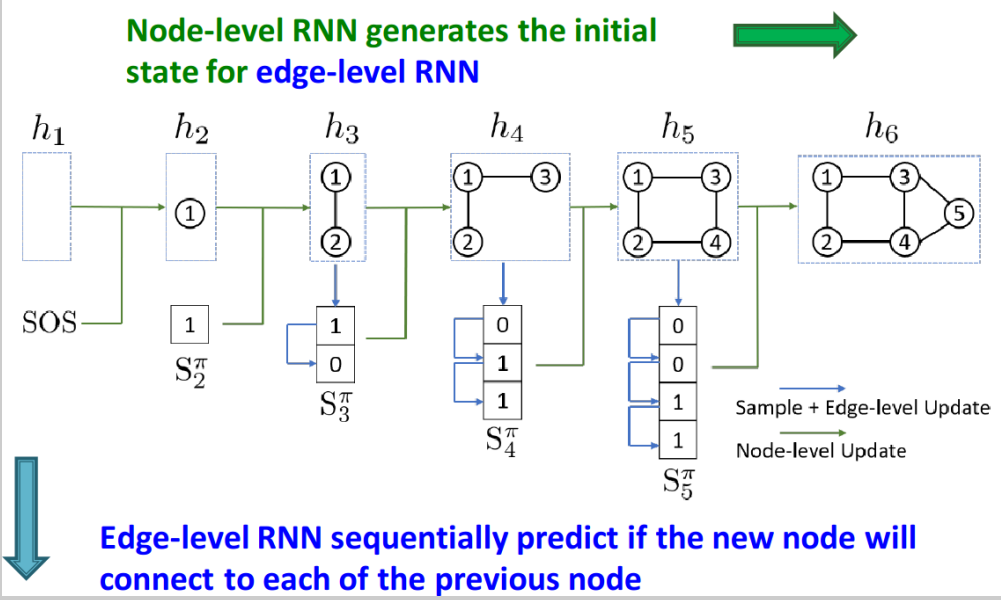
$$L := -(\hat{y}_j \log(y_j) + (1 - \hat{y}_j) \log(1 - y_j))$$

Question: Could we have vanshing gradients in GraphRNN?

If the generation sequence is too long this could be a problem.

Question: Does the generation process depend on the ordering of the nodes?

Yes.



Question: If the graph grows in multiple directions, how can the model build one part of the graph and then another part of the graph?

(#610)
Not sure if you have a specific context in mind but one way to mitigate forgetting other branches of a graph could be to use attention networks. It would definitely help in improving the forgetting part and depending on the problem, you can tweak the network to be more useful for any particular kind of forgetting you are facing.

Question: Sequence generation order agnostic supervision, would this penalise the model unnecessarily? Could we feed in stochastic inputs when generating cliques as teacher forcing?

Some models can generate larger graph structures (e.g. a clique) at a time and this mitigates some of the issue.

15.3 Scaling Up and Evaluating Graph Generation

The generation steps of GraphRNN could become intractable since any node can connect to any prior node. This is too many steps for edge generatio and requires complex long range dependencies. To mitigate this issue, we could use
Breadth-First Search node ordering.
The nodes are ordered using a BFS algorithm. If the algorithm is concerned with the generation of two nodes $i < j$ and no edges to j were produced at step i, at step j there is no need to consider an edge to node i. Using BFS ordering, the number of "look-back" steps is reduced from $n - 1$ to $\max_v \deg v$.

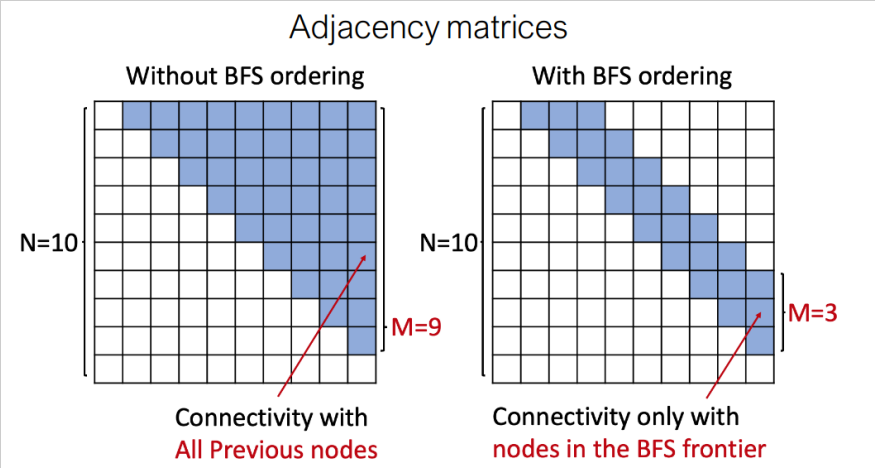


Figure 15.3: Ordering the nodes using BFS reduces the number of memory steps required to generate the graph

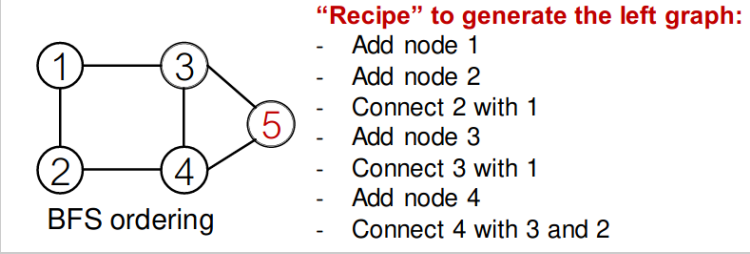


Figure 15.4: Example of generating a graph using BFS node ordering

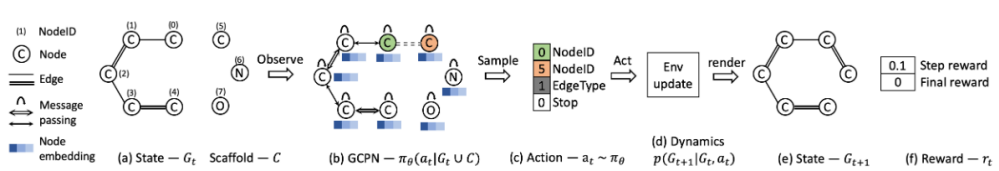


Figure 15.5: Overview of GCPN

Question: Why could 5 not have link to 1?

If 5 is connected to 1, in the BFS ordering 5 would be traversed before 4, so it would be 4 instead.

Question: Could you have edge RNN generate stop token?

Yes.

15.4 Graph Convolutional Policy Network

Sometimes we would like to generate graphs that:

- **Optimise a given objective** (High-score): e.g. drug-likeness

  This introduces a black-box to graph generation. Objectives like drug-likeness are governed by physical laws which is assumed to be unknown to us.

- **Obey underlying rules** (Valid): e.g. Chemical validity

- **Are learned from examples** (Realistic): Imitates a molecule graph dataset.

Excursion: Reinforcement Learning

In **Reinforcement Learning** (**RL**), a ML agent observes the *environment* and takes *action* to interact with the environment. The agent receives positive or negative *reward*. The ML agent is trained from this loop. The environment is a black box to the agent but the agent can directly learn from it.

Graph Convolutional Policy Network (**GCPN**) combines graph representation and reinforcement learning. GCPN generates graphs sequentially like GraphRNN but has some differences:

- GCPN uses GNN to predict the generation action (more expressive, but takes longer time to compute)

- GCPN uses RL to direct graph generation

Steps in GCPN:

(a) Insert nodes

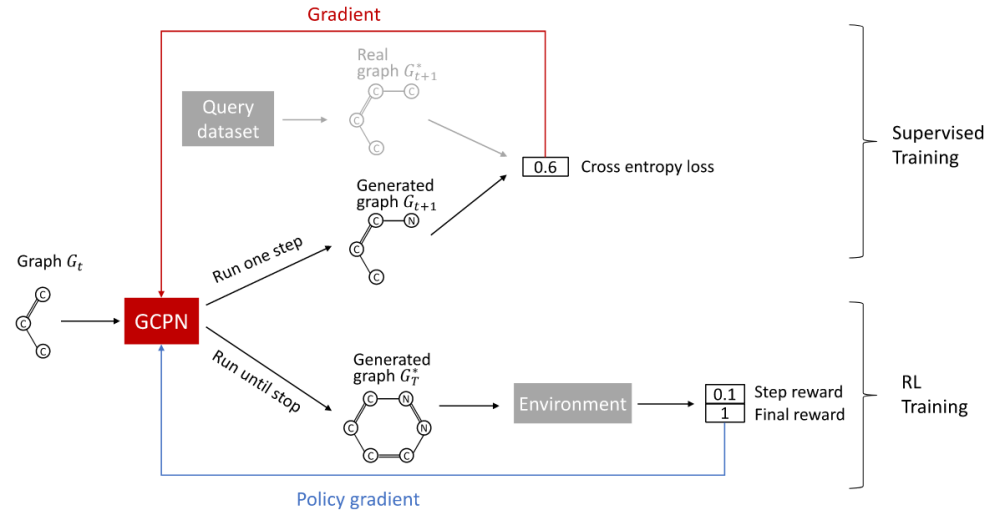


Figure 15.6: Training process of GCPN

(b,c) Use GNN to predict which nodes to connect

(d) Take an action

(e,f) Compute reward

- **Step Reward**: Learn to take valid action
- **Final Reward**: Optimise desired properties

GCPN is trained in two parts:

1. Supervised training: Train policy by imitating the action given by real observed graphs.
2. RL training: Train policy to optimise rewards using a policy gradient algorithm.

Question: Can graph completion task delete edges?

Not sure how this could be useful.

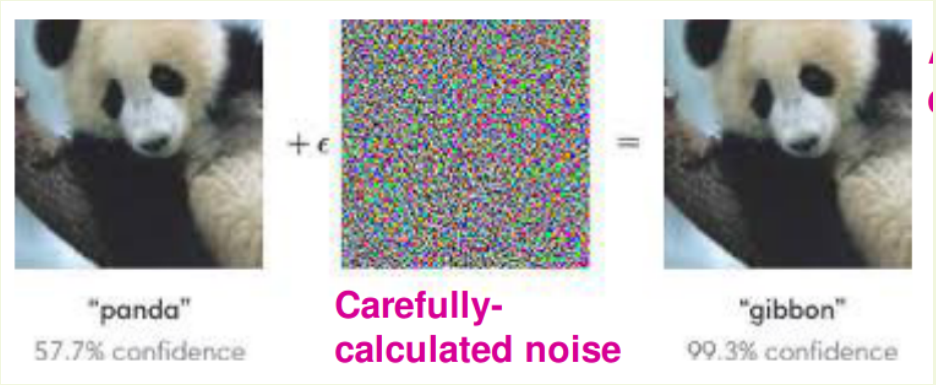
16 Advanced Topics in Graph Neural Networks

16.1 Robustness

Deep convolutional neural networks are vulnerable to *adversarial attacks*.

Excursion: Adversarial Attacks

Imperceptible changes can be added to neural network input to change the prediction.



Adversarial examples are also reported in natural language processing and audio processing.
The existence of adversarial examples prevent the reliable deployment of deep learning models to the real world. Adversaries may try to actively interfere the neural networks. Deep learning models are often not robust.

How robust are GNNs?

Settings

- Task: Semi-supervised node classification
- Model: GCN
- Target node $t \in V$: Node whose label prediction we want to change
- Attacker nodes $S \subseteq V$: Nodes the attacker can change

  The attacker can modify node feature, add connections, or remove connections.
- The attacker has access to $\boldsymbol{A}$ (the adjacency matrix), $\boldsymbol{X}$ (the feature matrix), $\boldsymbol{Y}$ (the label matrix), and the learning algorithm.
- The attacker can modify $(\boldsymbol{A}, \boldsymbol{X})$ to $(\boldsymbol{A}', \boldsymbol{X}')$ with the assumption $(\boldsymbol{A}', \boldsymbol{X}') \simeq (\boldsymbol{A}, \boldsymbol{X})$. The manipulation is unnoticeably small.
- $\boldsymbol{\theta}$ (resp. $\boldsymbol{\theta}'$) is the model parameter learned over $(\boldsymbol{A}, \boldsymbol{X}, \boldsymbol{Y})$ (resp. $(\boldsymbol{A}', \boldsymbol{X}', \boldsymbol{Y})$)
- c_v (resp. c'_v) is the class label of node v predicted by GCN with parameters $\boldsymbol{\theta}$ (resp. $\boldsymbol{\theta}'$).
- The attacker wants to make $c'_v \neq c_v$, i.e.

Change of prediction on target node v

$$\Delta(v; \boldsymbol{A}', \boldsymbol{X}') := \log f_{\boldsymbol{\theta}'}(\boldsymbol{A}', \boldsymbol{X}')_{v,c'_v} - \log f_{\boldsymbol{\theta}'}(\boldsymbol{A}', \boldsymbol{X}')_{v,c_v}$$

Predicted log probability of newly predicted class
Attacker wants to increase

Predicted log probability of originally predicted class
Attacker wants to decrease

- The optimisation objective of the attacker is

$$\arg\max_{\boldsymbol{A}',\boldsymbol{X}'} \Delta(v; \boldsymbol{A}', \boldsymbol{X}') \text{ subject to } (\boldsymbol{A}', \boldsymbol{X}') \simeq (\boldsymbol{A}, \boldsymbol{X})$$

Change of target node label prediction

Graph manipulation is small

Question: Why we assume only the adjacency matrix and feature matrix can be changed, but not the labels?

If the attacker can change the label the adversary problem becomes very easy.

Question: Does the attacker need to know the structure of the model itself?

Yes.

Attack possibilities:

- **Direct attack**: Node is the target $S = \{t\}$
- **Indirect attack**: The target node is not in the attacker nodes: $t \notin S$

Optimising the objective function $\Delta(v; \boldsymbol{A}', \boldsymbol{X}')$ is challenging since $\boldsymbol{A}'$ is a discrete object and for every modified graph $\boldsymbol{A}', \boldsymbol{X}'$ the GCN needs to be retrained. The solution [Zügner et al. KDD2018] is to follow a iterative locally optimal strategy:

1. Sequentially manipulate the most promising element/entry from $\boldsymbol{A}$
2. Pick one which obtains the highest difference in the log-probabilities indicated by the score function

GCN is not robust to *direct* adversarial attacks but it is somewhat robust to *indirect* and *random* attacks.

Question: Has there been work of using Reinforcement Learning to train the attacker to attack a black box?

Yes, in this work they assume the overall architecture of the model but not its weights.

17 Scaling Up GNNs

In modern applications, graphs are very large.

- In **recommender systems** used by Amazon/Youtube/Pinterest etc., the number of users is on the order of $10^8 to 10^9$, and the number of products/videos is on the order of 10^7 to 10^9.

  Tasks: Recommend items, Classify users/items

- In **social networks** used by Facebook/Twitter/Instagram etc., the number of users is on the order of 10^8 to 10^9

  Tasks: Friend recommendation, User property recommendation

- In **academic graphs** such as Microsoft Academic Graph

  Tasks: Paper categorisation, author collaboration recommendation citation recommendation

- In **knowledge graphs** such as Wikidata or Freebase, there are about 10^8 entities.

  Tasks: KG Completion, Reasoning

Why is training GNN difficult?

Excursion: Training Machine Learning Models

The objective of training machine learning models is usually to minimise some averaged loss:

$$\ell(\theta) := \frac{1}{N}\sum_{i=0}^{N-1} \ell_i(\theta)$$

We perform *stochastic gradient descent* by randomly sampling batches $B$of size $M \ll N$, calculate the loss $\hat{\ell}(\theta)$ over B, and update using $\theta \leftarrow \theta - \alpha\nabla\hat{\ell}(\theta)$.

Naïve full-batch processing iterates the GCN layers on the entire graph:

$$\boldsymbol{H}^{(k+1)} := \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(k)}\mathsf{W}_K^\intercal + \boldsymbol{H}^{(k)}\mathsf{B}_K^\intercal\right)$$

is infeasible in large graphs since the memory on a GPU is extremely limited (10-20 GB). We introduce 3 methods to scaling up GNNs.

- Two methods sample smaller subgraphs:

  Neighbour Sampling and Cluster GCN

- One method simplifies GNN into a feature preprocessing operation:

  Simplified GCN

17.1 Neighbour Sampling

Recall that GNNs generate node embeddings via neighbour aggregation. To compute the embedding of a single node n, all we need is the K-hop neighbourhood. We can sample $M \ll N$ nodes, construct their K-hop neighbourhoods, and use the resulting computational graph to train a GNN. The issue with this method is the size of K-hop neighbourhoods can be exponentially large, especially when it hits a **hub node** (a node with a high degree).
In **Neighbourhood Sampling**, each hop only samples at most H_k neighbours at the kth level. We then use the pruned computational graph to train the GNN. A K-layer GNN will involve at most $\prod_{k=1}^{K} H_k$ leaf nodes in the computational graph.
Benefits and Drawbacks

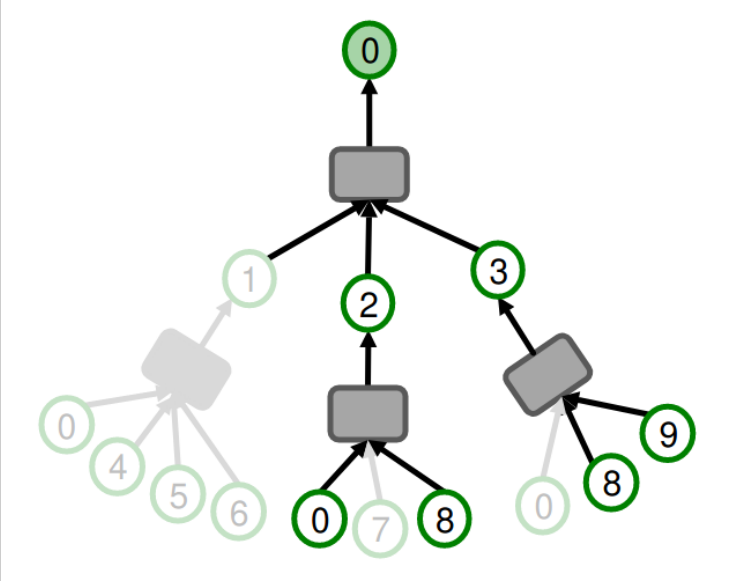


Figure 17.1: Neighbourhood Sampling with $H = 2$

- H is the trade off between aggregation efficiency and variance. A smaller H leads to more efficient computation but increases the variance which gives more unstable training results.
- The size of the computation graph is still exponential w.r.t. K. One more GNN layer makes computation H times more expensive.

The neighbours can be sampled randomly, which is fast but might not be optimal. In natural graphs. We could use **random walks with restarts** and sample nodes with the highest restart scores. This works better in practice.
Overall time complexity: $M \cdot H^K$.

17.2 Cluster GCN

Observe that when nodes share many layers, computation of the embeddings of these neighbours become redundant. In a full-batch GNN, only $2\,|E_M|$ (where E_M is the set of edge messages) messages need to be computed, and overall only $2K\,|E_K|$ messages are needed in a K-layer GNN.
Layer-wise node embedding update allows the re-use of embeddings from the previous layer. This is infeasible in large graphs due to limited GPU memory.
In Cluster-GCN, we can sample a small subgraph of the large original graph and perform efficient layer-wise node embedding updates in the smaller graph. What subgraphs are good for training GNNs? Since GNNs pass messages along edges, the subgraphs should retain as much connectivity from the original graph as possible. Real-world graphs exhibit *community* structure, and we can sample a community as a subgraph, each of which retains essential connection patterns.
Cluster-GCN:

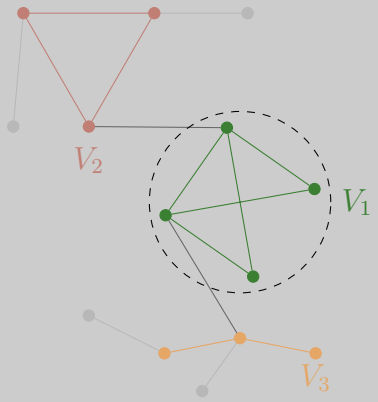


Figure 17.2: Cluster GCN

1. **Preprocessing**: Given a large graph, partition it into groups of nodes

   Partition V of a large graph $G = (V, E)$ into C groups $V_1, \ldots, V_C$ using a community detection algorithm e.g. Louvain, METIS. Each V_i induces a subgraph G_i. Note that between group edges in G *are dropped*.

2. **Mini-batch training**: Sample node group V_i at a time and apply GNN on node group G_i.

The issues with this algorithm are:

- Between group edges are dropped

  This leads to systematically biased gradient estimates.

- Sampled node group tends to only cover the small-concentrated portion of the entire data.

- Sampled nodes are not diverse enough to be represent the entire graph structure

  This leads to very different gradients across clusters, which translate to high training variance and slow convergence of SGD.

The solution is to aggregate multiple node groups per mini-batch. This is **Advanced Cluster-GCN**:

1. **Preprocessing**: Given a large graph, partition it into groups of nodes

   Partition V of a large graph $G = (V, E)$ into C groups $V_1, \ldots, V_C$ using a community detection algorithm. They have to be small enough such that multiple node groups can be loaded into the GPU at a time.

2. **Mini-batch training**: Sample node groups $V_{i_1}, \ldots, V_{i_q}$ and apply GNN to the induced graph of $V_{i_1} \cup \cdots \cup V_{i_q}$.

Question: What about nodes that have no communities?

Assign the isolated nodes into some community

Question: Can we train different models on different subgraphs?

Its better to train one model to prevent models from overfitting on a subgraph.

Question: Induced subgraphs lose connection between subgraphs. Can we treat induced subgraphs as hypernodes?

That is an interesting research problem. It is unclear if you can represent the entire subgraph using one hypernode.
See GNNAutoScale.

Question: Is the computational trade off of selecting a large number of small communities vs. small number of larger communities worth it?

Yes. Many community selection algorithms are linear in $|V|$.

Question: Can Cluster-GCN be easily adapted to heterogeneous graphs?

This might introduce bias for the edge types. This could be a good research question.

Overall time complexity of Cluster-GCN: $K \cdot M \cdot D_{\text{avg}}$, where D_{avg} is the average node degree. This linear growth is much more efficient than neighbourhood sampling.

17.3 Simplified GCN

We start from Graph Convolutional Network (GCN). Recall that the GCN aggregation function is of the form $\boldsymbol{E}^{(k+1)} := \text{ReLU}(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(k)}\mathsf{W}^{(k)})$.
Simplified GCN uses self-loops on the adjacency matrix $\boldsymbol{A}$: $\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{I}$. It also assumes the node embeddings $\boldsymbol{E}$ are given as features. These are the main differences from LightGCN. Since the node features are fixed, the matrix $\tilde{\boldsymbol{E}} := \tilde{\boldsymbol{A}}^K \boldsymbol{E}$ can be calculated only once, as a preprocessing step.
$\tilde{\boldsymbol{E}}$ already has very rich features and can be fed to existing machine learning models.

Question: Since we are removing the activation function and the performance is still strong, does this mean a lot of things we learned are linear?

Yes.

The drawback is that Simplified GCN is *less* expressive.
Despite being less expressive, Simplified GCN performs comparably to the original GNNs. The reason for this is **homophily** in many node classification tasks, i.e. nodes connected by edges tend to share the same target labels. As a result, nodes connected by edges tend

to have similar pre-processed features. This is due to the high dimensionality of embedding space.

18 Geometric Graph Learning

Settings

A graph $G = (\boldsymbol{A}, \boldsymbol{S})$ is a set V of n nodes connected by edges. Each node has *scalar attributes* (e.g. atom type). $\boldsymbol{A}$ is the adjacency matrix and $\boldsymbol{S} \in \mathbb{R}^{|V| \times f}$
A **geometric graph** is a graph $G = (\boldsymbol{A}, \boldsymbol{S}, \boldsymbol{R})$, where each node is embedded in d-dimensional Euclidean space, i.e. $\boldsymbol{R} \in \mathbb{R}^{|V| \times d}$.

Molecules can be represented as a graph G with node features $\boldsymbol{s}_i$ (atom type, charges) and edge features $a_{i,j}$ (valence bond type). Sometimes we also know the 3D positions of each node $\boldsymbol{r}_i$.
Geometric graphs lead to a variety of GNN models: Geometric GNN, Geometric Generative model.
To design a GNN which processes geometric graphs, we need to overcome some obstacles. The coördinate system used to describe graph geometry transform the node coördinates. The output of traditional GNN will be affected by this transformation, so we would like the GNN to be aware of symmetry based on the coördinates system.
A function $F : X \to Y$ is

- **Equivariant** if it commutes with a rigid transformation. i.e. for a transformation ρ it satisfies $F \circ \rho_X = \rho_Y \circ F$.
- **Invariant** if $F \circ \rho_X = F$.

For example, force on a molecule is equivariant, and energy in a molecule is invariant.
For geometric graphs, we consider the group of 3D special Euclidean symmetries SE(3), which consists of rotations and translations.
For ML models with no explicit handling of symmetry, we can use **data augmentation**: Create more training data by randomly generating rigidly transformed original data. This is expensive. An alternative is to develop a GNN which respects the symmetry in the input. This substantially shrinks the search space of the training process.
There are two classes of Geometric GNNs:

- Invariant GNNs for learning invariant scalar features
- Equivariant GNNs for learning equivariant tensor features

18.1 Invariant GNNs

Continuous-Filter Convolutions refers to convolutions executed on a continuous grid. Such convolution filters may be useful when the continuous position of the irregular grid

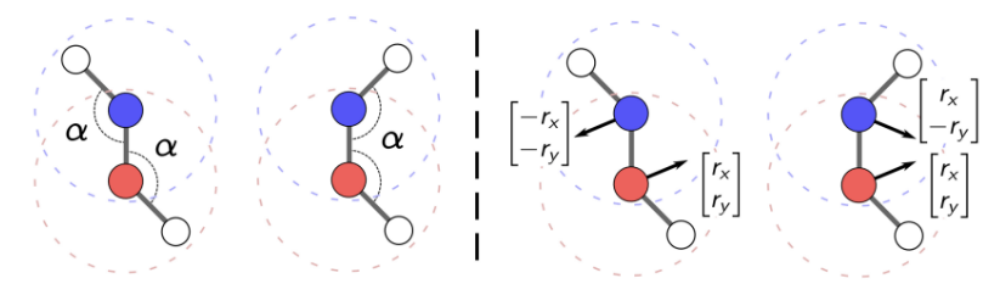


Figure 18.1: A pair of molecules exhibiting geometric isomerism, having identical scalar quantities (distance and angle) but are distinguished by directional (normal) and geometric information.

becomes important, e.g. positions of atoms in a molecule. A continuous-filter convolution on the features $\boldsymbol{H}^{(l)}$ is

Element-wise Multiplication

$$\boldsymbol{h}_i^{(l+1)} := (\boldsymbol{H}^{(l)} * \mathsf{W}^{(l)})_i = \sum_j \boldsymbol{h}_j^{(l)} \odot W^{(l)}(\boldsymbol{r}_i - \boldsymbol{r}_j)$$

Filter-Generating Function $\mathsf{W}^{(l)} : \mathbb{R}^d \to \mathbb{R}^f$ — Relative Position

SchNet is a class of invariant GNNs, the filter-generating function $\mathsf{W}^{(l)}$ is a learnable function (MLP) of the radial bases functions

Interatomic distance $d_{i,j} := \|\boldsymbol{r}_i - \boldsymbol{r}_j\|$

$$e_k(\boldsymbol{r}_i - \boldsymbol{r}_j) := \exp(-\gamma \left| d_{i,j} - \mu_k \right|^2)$$

Radial basis centres

This is to curb the initial training difficulties. [12] SchNet makes W invariant by *scalarising* relative positions. This gives up the angular information in the edges.
DimeNet improves upon SchNet by using both distance and angle features during message passing.
Still, distances/angles are incompletely descriptors for uniquely identifying geometric structure (e.g. cis-1,2-dichloroethene and trans-1,2-dichloroethene).
Limitations of invariant GNNs: You have to guarantee that your input features already contain any necessary equivariant interactions.

18.2 Equivariant GNNs

PaiNN is a class of equivariant GNNs. PaiNN still takes learnable weights W conditioned on the relative distance but each node has two features, a scalar feature s_i and vector feature $\boldsymbol{v}_i$. They are defined by

$$s_i^{(0)} := 0 \qquad s_i^{(l+1)} := s_i^{(l)} + \Delta s_i^{(l)}$$

$$\boldsymbol{v}_i^{(0)} := \boldsymbol{e}_i \qquad \boldsymbol{v}_i^{(l+1)} := \boldsymbol{v}_i^{(l)} + \Delta \boldsymbol{v}_i^{(l)}$$

Atom embedding

[12] See SchNet

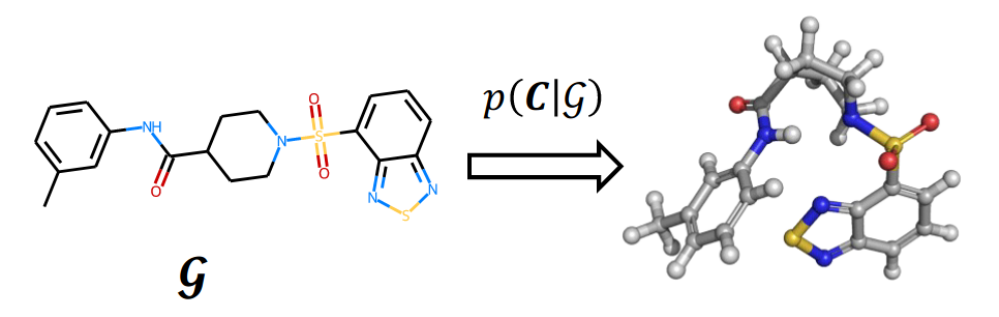


Figure 18.2: Molecular conformation generation

where the updates $\Delta s_i, \Delta \boldsymbol{v}_i$ are defined by

$$\Delta s_i^{(l)} := (\phi_s(s^{(l)}) * \mathsf{W}_s)_i = \sum_j \phi_s(s_j^{(l)}) \odot \mathsf{W}_s(\|\boldsymbol{r}_{i,j}\|)$$

$$\Delta \boldsymbol{v}_i^{(l)} := \sum_j \boldsymbol{v}_j^{(l)} \odot \phi_{vv}(s_j^{(l)}) \odot \mathsf{W}_{vv}(\|\boldsymbol{r}_{i,j}\|) + \sum_j \phi_{vs}(s_j^{(l)}) \odot \mathsf{W}_{vs}(\|\boldsymbol{r}_{i,j}\|) \frac{\boldsymbol{r}_{i,j}}{\|\boldsymbol{r}_{i,j}\|}$$

where ϕ, W are neural networks. This passes invariant scalar messages and equivariant vector messages through each layer, thus keeping the equivariant properties.

18.3 Geometric Generative Models

Geometric Generative Models have application in molecule/protein design, biomolecule structure prediction, and protein-molecule interaction simulation.
The problem of **Molecular Conformation Generation** is to generate stable conformations from molecule graph. The model converts a molecular graph (atom-bond graph) $\mathcal{G}$ to a conformation $\boldsymbol{C}$ with 3D coördinates. $\boldsymbol{C}$ follows the Boltzmann distribution $\boldsymbol{C} \propto \exp(-E(\boldsymbol{C})/T)$.

Excursion: Diffusion Models

A *forward diffusion process* destroys data by gradually adding noise and **diffusion models** learn to reverse this noise i.e. *denoise*.
Let $t_1 = 0, \ldots, t_n = T$ be time steps. Corrupt the data x_0 by adding noise $x_t := \mu_t x + \sigma_t \epsilon$. The diffusion model is trained to reverse every step of this process.

Geometric Diffusion (**GeoDiff**) samples by an equivariant denoising procedure. The noise generation process is a distribution $q(\boldsymbol{C}^{(t)}|\boldsymbol{C}^{(t-1)})$ and GeoDiff learns its inverse $p_\theta(\boldsymbol{C}^{(t-1)}|\mathcal{G}, \boldsymbol{C}^{(t)})$. The denoising network is parameterised by an equivariant GNN.

19 Trustworthy Graph AI

Trustworthy AI/GNN include explainability, fairness, robustness, privacy, etc. Previously the role of graph topology is unexplored in these problems. This lecture covers robustness and explainability.

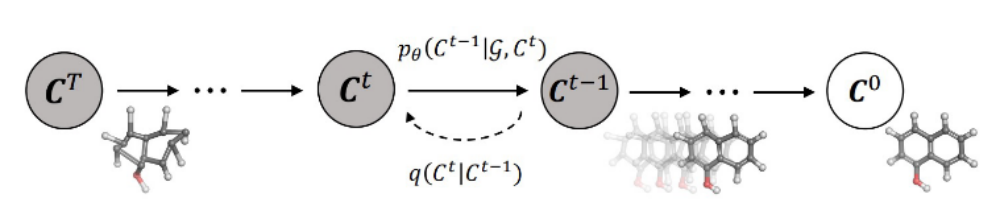


Figure 18.3: Geometric Diffusion

19.1 Explainability

Deep learning models are black boxes which makes it a major challenge to explain and extract insight from. **Explainable Artificial Intelligence** (**XAI**) is an umbrella term for any research trying to solve the black-box problem for AI.
It is useful since it enables

- **Trust**: Explainability is a prerequisite for humans to trust and accept the model's prediction.
- **Causality**: Explainability (e.g. attribute importance) conveys causality to the system's target prediction: attribute X causes the data to be Y
- **Transferability**: The model needs to convey an understanding of decision- making for humans before it can be safely deployed to unseen data.
- **Fair and Ethical Decision Making**: Knowing the reasons for a certain decision is a societal need, in order to perceive if the prediction conforms to ethical standards.

Excursion: Explainable Models

A model is **explainable** when

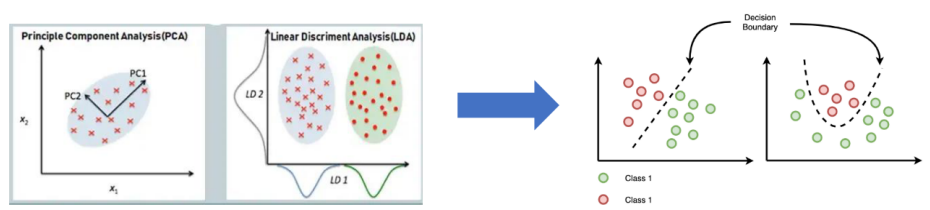
- **Importance values**: Scores for input features
- **Attribution**: straightforward relationships between prediction and input features

Examples:

- Linear Regression: The most simple model is linear regression. In a model specified by

$$\underbrace{y}_{\text{Prediction}} := \overbrace{w_1}^{\text{Weights}} \overbrace{x_1}^{\text{Features}} + \cdots + w_d \, x_d$$

  The slope is explainable as the amount of effect a variable has on the prediction.
- Dimension Reduction: Allows us to visualise the training data distribution geometrically with a boundary characterising different classes.



- Decision Trees: A very explainable set of models where each node represents a logical decision. We can compute statistics for each decision node.

What makes models explainable?

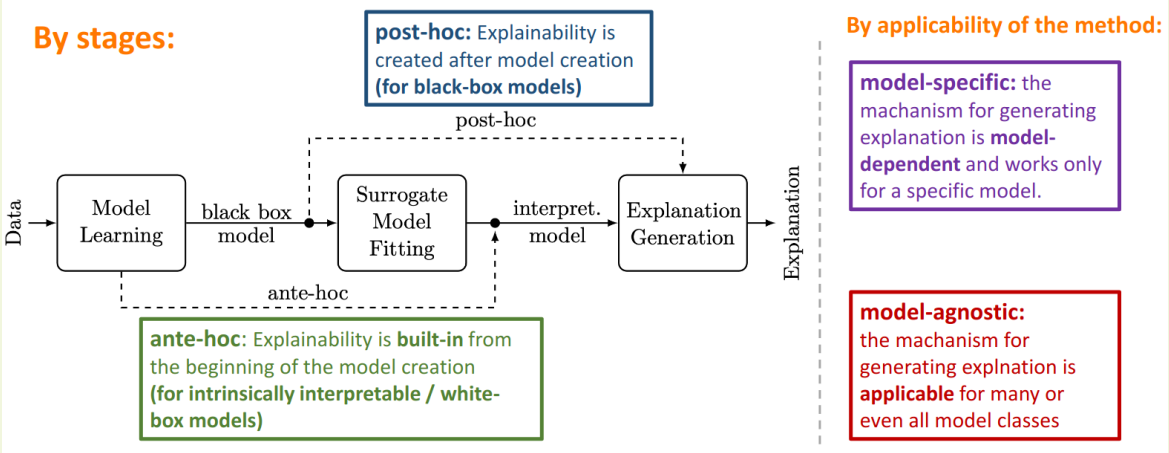
- **Importance** values: Assigning importance to features, pixels, words, nodes, etc.
- **Attributions**: Straightforward relationships between prediction and input features.
- **Concepts** and **Prototypes**

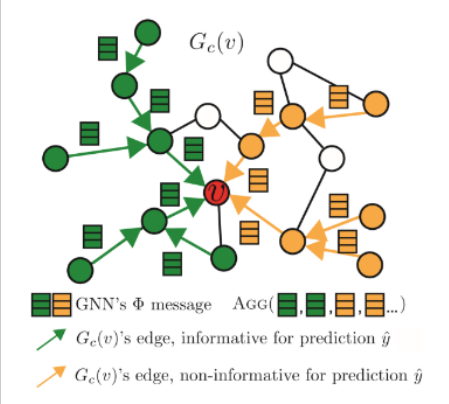
Some architectures provide explainability:

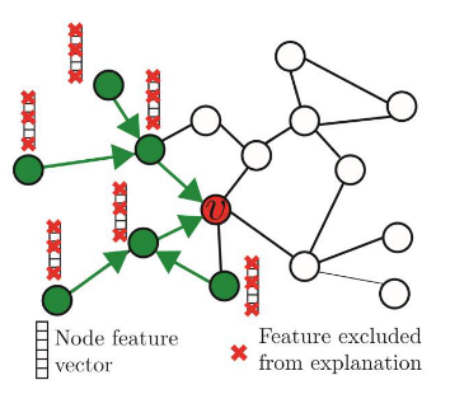
- **Proxy Model**: Learn an interpretable model that locally approximates the original model.
- **Saliency Map**: Compute the gradients of outputs w.r.t. inputs
- **Attention**: Visualise attention weights in attention models such as transformers and GAT.

Explainability settings can be classified into:

- **Instance-level**: Local explanation for a single input x and prediction $\hat{y}$
- **Model-level**: A global explanation of a specific dataset D or classes of D







Explanation in Graph Learning: an important subgraph structure and a small subset of node features that play a crucial role in GNNs prediction. Examples:

- Explaining ground truth phenomenon: What are the characteristics of a toxic molecule?
- Explaining model predictions: Why does the model recommend no loan for person X?

Question: Can we make an explainer of the post-hoc explainer model and so on?

Yes. This is done for CV and it often gives better performance.

19.2 GNNExplainer

GNNExplainer is a post-hoc, model agnostic explanation method for GNNs.

- Training time: Optimise GNN on training graphs and save the trained model
- Test time: Explain the prediction of GNN on unseen instances

GNNExplainer can explain different tasks, including Node classification, Link prediction, and Graph classification. It can be adapted to GAT, Gated Graph Sequence, Graph Networks, GraphSAGE, etc.
In a general message-passing framework, we can produce *structural explanations* and *feature explainations*. GNNExplainer explains both aspects simultaneously.

Settings

Without loss of generality, consider the node classification task. The input is

- The computation graph $G_c(v)$
- Adjacency matrix $\boldsymbol{A}_c(v)$
- Node features are given by $X_c(v) := \{x_u : u \in G_c(v)\}$

- GNN model ϕ learns a distribution $P_\phi(Y|\boldsymbol{A}_c(v), X_c(v))$.
- GNNExplainer outputs $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{X}}^F)$. Graph $\hat{G}$ with adjacency matrix $\hat{\boldsymbol{A}}$ is a subgraph of $G_c(v)$
- $\hat{\boldsymbol{X}}^F := \{\boldsymbol{x}_u^F : u \in \hat{G}\}$
- F is a mask which masks out unimportant features.

Excursion: Mutual Information

The **mutual information** of two random variables X, Y measure the correlation It is defined as

$$\begin{aligned}\mathrm{I}(X,Y) &:= \mathrm{D}_{\mathsf{KL}}(P_{(X,Y)} \| P_X \otimes P_Y) \\ &= \mathrm{H}(X) - \mathrm{H}(X|Y) = \mathrm{H}(Y) - \mathrm{H}(Y|X) \\ &= \mathrm{H}(X) + \mathrm{H}(Y) - \mathrm{H}(X,Y) \\ &= \mathrm{H}(X,Y) - \mathrm{H}(X|Y) - \mathrm{H}(Y|X)\end{aligned}$$

where P_X, P_Y are the marginal distributions of X, Y, resp., and $P_{(X,Y)}$ is the joint distribution of $P_{(X,Y)}$. $\mathrm{I}(X,Y)$ measures the amount of deviation the product distribution has relative to the hypothetical case of when X, Y have no correlation.
$\mathrm{I}(X,Y) = \mathrm{I}(Y,X)$

A good explanation should have a high correlation with model prediction, so the GNNExplainer's goal is

$$\max_{\hat{G}} \mathrm{I}(\underbrace{Y}_{\text{Label}}, \underbrace{(\hat{\boldsymbol{A}}, \boldsymbol{X}^F)}_{\text{Explanation}}) = \mathrm{H}(Y) - \mathrm{H}(Y|\underbrace{\boldsymbol{A} = \hat{\boldsymbol{A}}}_{\text{Subgraph}}, \underbrace{\boldsymbol{X} = \boldsymbol{X}^F}_{\text{Feature subset}})$$

Finding $\hat{\boldsymbol{A}}$ that minimises the conditional entropy $\mathrm{H}(Y|\cdots)$ is computationally expensive due to the exponentially many possible $\hat{\boldsymbol{A}}$. The solution in GNNExplainer is to treat explanation as a distribution of *plausible explanations* instead of a single graph. This has two benefits:

- Captures multiple possible explanations for the same node
- Turns discrete optimisation to continuous

GNNExplainer instead optimises the *expected adjacency matrix* $\hat{\boldsymbol{A}}$:

$$\begin{aligned}\min_{\mathcal{A}} \mathbb{E}_{\tilde{\boldsymbol{A}} \sim \mathcal{A}} \mathrm{H}(P_\phi(Y = y | \boldsymbol{A} = \tilde{\boldsymbol{A}}, \boldsymbol{X} = \hat{\boldsymbol{X}}^F)) &\simeq \min_{\mathcal{A}} \mathrm{H}(P_\phi(Y = y | \boldsymbol{A} = \mathbb{E}_{\mathcal{A}}[\tilde{\boldsymbol{A}}], \boldsymbol{X} = \hat{\boldsymbol{X}}^F)) \\ &\simeq \min_{\mathcal{A}} \mathrm{H}(P_\phi(Y = y | \boldsymbol{A} = \boldsymbol{A}_c \odot \sigma(\boldsymbol{M}), \boldsymbol{X} = \hat{\boldsymbol{X}}^F))\end{aligned}$$

Note the approximation used here since H is not a convex function. GNNExplainer uses $\boldsymbol{A}_c \odot \sigma(\boldsymbol{M})$, where $\sigma(\boldsymbol{M})$ is a mask, to approximate $\mathbb{E}_{\mathcal{A}}[\tilde{\boldsymbol{A}}]$. σ is the sigmoid function which squashes $\boldsymbol{M}$ into $[0, 1]$, representing whether an edge should be kept or dropped.

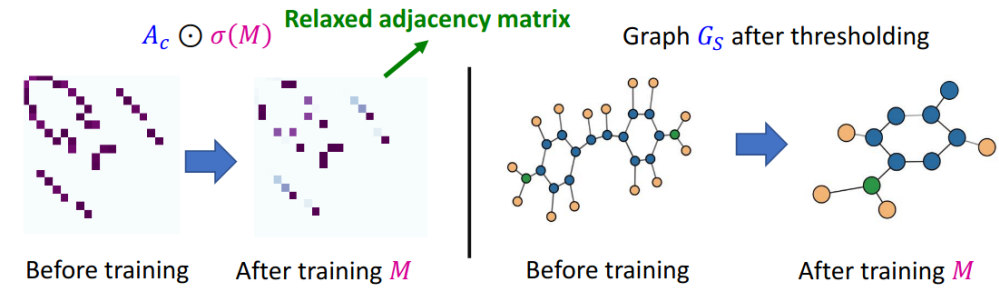


Figure 19.2: GNNExplainer thresholding graphs using the relaxed adjacency matrix

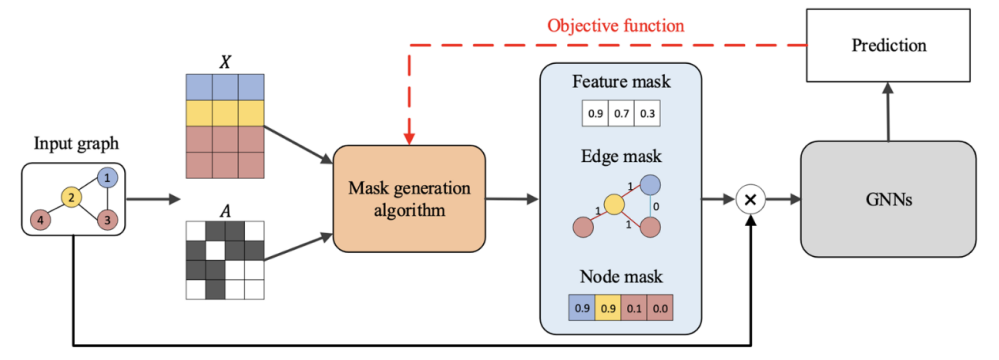


Figure 19.3: GNN Post-hoc explanation pipeline

After optimisation, threshold on $\boldsymbol{A}_c \odot \sigma(\boldsymbol{M})$ to obtain $\hat{\boldsymbol{G}}$. Similarly, select features by optimising for feature mask $\sigma(\boldsymbol{F})$. Finally, we add regularisation terms and this forms the training objective of GNNExplainer.

$$\min_{\boldsymbol{M},\boldsymbol{F}} - \mathrm{H}(P_\phi(Y = y | \boldsymbol{A} = \boldsymbol{A}_c \odot \sigma(\boldsymbol{M}), \boldsymbol{X} = \hat{\boldsymbol{X}}^F)) + \underbrace{\lambda_1 \sum \sigma(\boldsymbol{M}) + \lambda_2 \sum \sigma(\boldsymbol{F})}_{\text{Regularisation}}$$

- Node classification: Optimise mask $(\boldsymbol{M}, \boldsymbol{F})$ on the node's neighbourhood
- Edge prediction: Optimise mask $(\boldsymbol{M}, \boldsymbol{F})$ on two nodes' neighbourhoods
- Graph classification: Optimise mask $(\boldsymbol{M}, \boldsymbol{F})$ on the entire graph

Alternative approaches:

- **GNN Saliency Map**: Record of the gradients of output scores w.r.t. inputs on a GNN
- **Attention** from GAT: Attention scores provide some explanation.

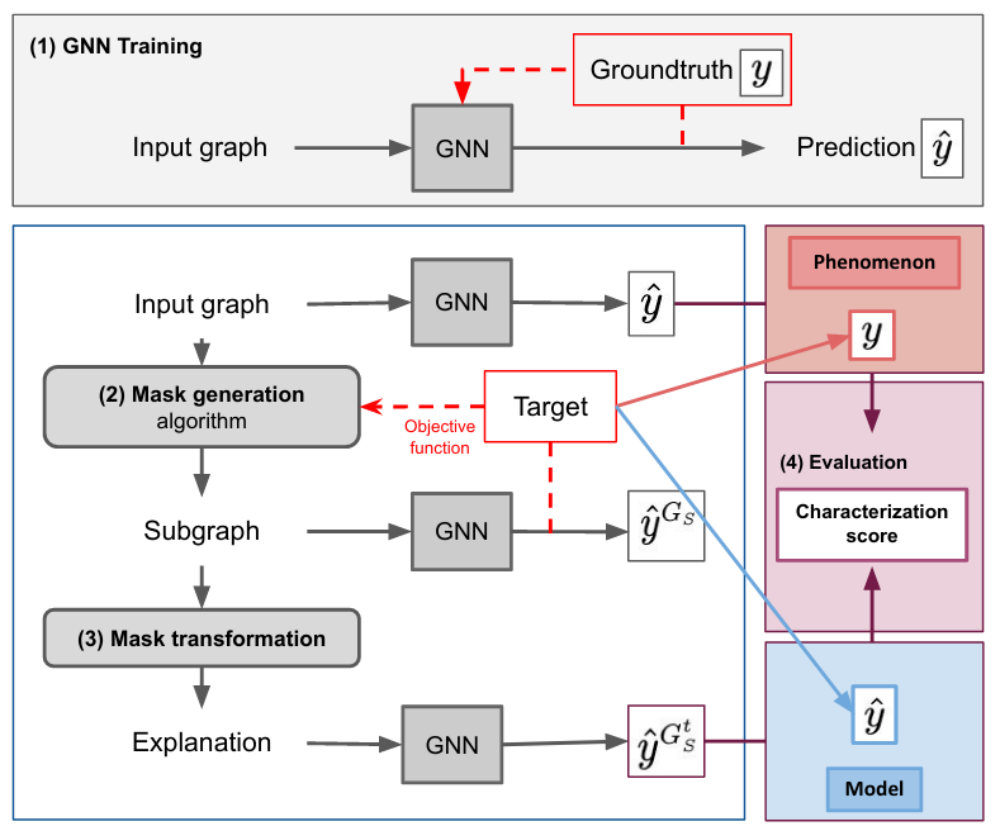


Figure 19.4: GraphFramEx explanation framework focuses on the phenomenon and the model.

19.3 Explainability Evaluation

During evaluation, the ground truth might not always be available, and evaluation is multi-dimensional: It can be done on the **goal** (phenomenon/model), **masking**, or **type** (sufficiency/necessity)

- **Phenomenon explanation** refers to explaining the underlying reasons for the ground truth phenomenon
- **Model explanation** refers to explaining why model makes a particular prediction

The **fidelity metrics** are $\mathsf{fid}_{\pm}$, corresponding to removing important subgraphs and using only the important subgraphs.

$$\mathsf{fid}_+ := \frac{1}{N}\sum_{i=1}^{N} \left| \mathbb{1}(\hat{y}_i = y_i) - \mathbb{1}(\hat{y}_i(G_{C\setminus S}) = y_i) \right|$$

$$\mathsf{fid}_- := \frac{1}{N}\sum_{i=1}^{N} \left| \mathbb{1}(\hat{y}_i = y_i) - \mathbb{1}(\hat{y}_i(G_S) = y_i) \right|$$

Remove important subgraph S

Original prediction probability/confidence

Keep only important subgraph S

Evaluation criteria are multidimensional:

- **Quality**: High fidelity/characterisation scores
- **Stability**: Consistency across random optimisation seeds
- **Complexity**: Explanation should be concise and easy to understand

Types of explanations:

- **Sufficiency**: An explanation is sufficient if it leads by its own to the initial prediction of the model explanation ($\mathsf{fid}_- \to 0$)
- **Necessity**: An explanation is necessary if the model prediction changes when removing it from the initial graph. ($\mathsf{fid}_+ \to 1$)

The **characterisation score** summarises the explanation quality

$$c := \frac{w_+ + w_-}{\frac{w_+}{\mathsf{fid}_+} + \frac{w_-}{\mathsf{fid}_-}} = \frac{(w_+ + w_-)\cdot \mathsf{fid}_+ \cdot (1 - \mathsf{fid}_-)}{w_+(1-\mathsf{fid}_-) + w_- \cdot \mathsf{fid}_+}$$

where $w_\pm$ are the weights of both fidelity metrics and usually $w_\pm = 1$.
Other types of explanations:

- **Counterfactual explanations**: What makes an instance belonging to a different class? What perturbation is needed to move an instance to a different class?
- **Model-Level explanations**: What are the general characteristics of all instances belonging to a certain class?

20 Conclusion

20.1 GNN Design Space and Task Space

How to find a good GNN design for a specific GNN task? Redo hyperparameter grid search for each new task is not feasible.

- **Design**: is a concrete model instantiation

  e.g. 4-layer GraphSAGE

- **Design Dimension**: Characterises a design

  e.g. The number of layers $l \in \{2, 4, 6, 8\}$

- **Design choice**: The actual selected value of the design dimension

  e.g. The number of layers is $l := 2$

- **Design space**: Cartesian product of design dimensions

- **Task**: A specific task of interest

  e.g. Node classification on Cora, Graph classification on ENZYMES.

- **Task Space**: Consists of all the tasks we care about

THe GNN Design space consist of

- **Intra-Layer Design**: GNN Layer is transformation and aggregation

  e.g. Batch Normalisation, Dropout $(0, 0.3, 0.6)$, Activation (ReLU, Swish), Aggregation

- **Inter-Layer Design**: Explore different ways of organizing GNN layers

  e.g. Layer connectivity (Skip connection), Pre-process layers $(1, 2, 3)$, Message passing connections $(2, 4, 6, 8)$, Post-process layers $(1, 2, 3)$

- **Learning-Configuration**: Batch size, Learning rate, Optimiser, etc.

  e.g. Batch size, Learning Rate, Optimiser, Training epochs

Overall there are $300,000$ possible designs for an assorted combination of parameters. The total size of the design space is huge ($> 10^5$". We cannot cover all possible designs.
GNN tasks can be categorised into node/edge/graph level tasks. This is not precise enough, since for example "predicting clustering coëfficient" and "predicting a node's subject area in citation networks" are completely different.

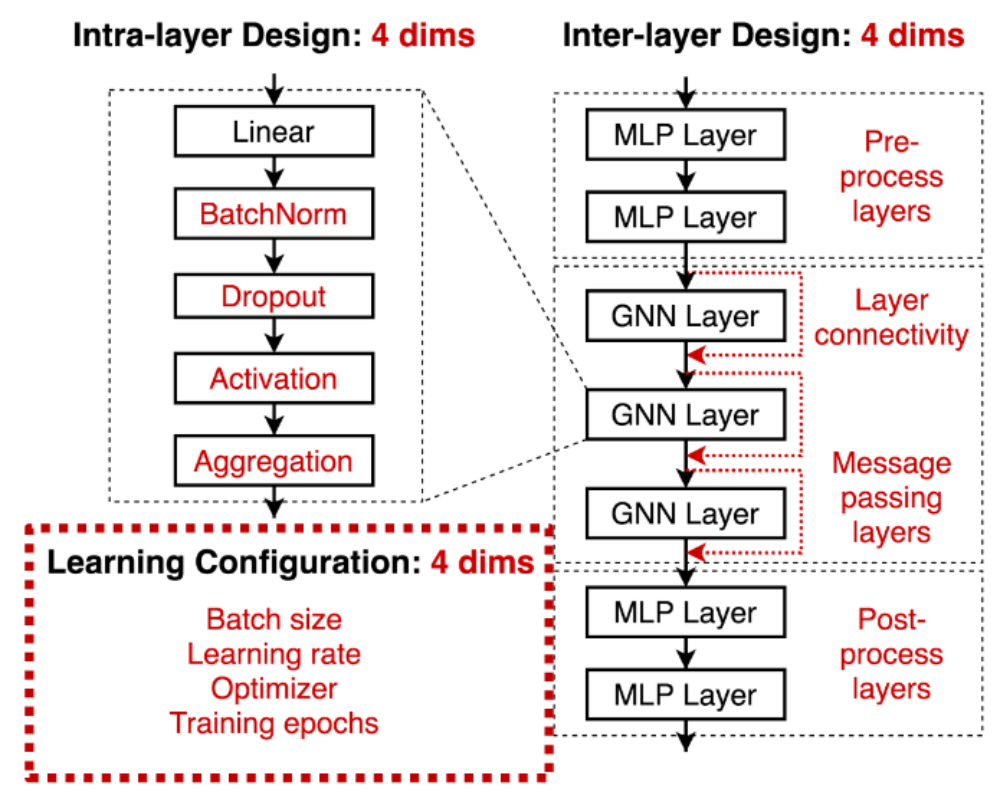


Figure 20.1: The design space of GNNs

20.2 GraphGym

GraphGym is a platforms for exploring different GNN architectures.
In GraphGym, a quantitative *task similarity metric* is defined as

1. Select anchor models $(M_1, \ldots, M_5)$
2. Characterise a task by ranking the performance of anchor models
3. Tasks with similar *rankings* are similar.

Anchor models can be selected from

1. Select a small dataset (e.g. node classification on Cora)
2. Randomly sample N models from our design space (e.g. sample 100 models)
3. Sort these models based on their performance (e.g. sample 12 models in the experiments)

	Anchor Model Performance ranking					Similarity to Task A
Task A	M_1	M_2	M_3	M_4	M_5	1.0
Task B	M_1	M_3	M_2	M_4	M_5	0.8
Task C	M_5	M_1	M_4	M_3	M_2	-0.4

Figure 20.2: Task similarities of 3 tasks

Example

Evaluating a design dimension: If we want to inquire about "is Batch Normalisation generally useful for GNNs"? The common practice is to select one model and compare it with batch normalisation on and off.
In GraphGym, the process is

1. Sample from $10^7 \simeq 32_{(\text{tasks})} \cdot 315000_{(\text{model})}$ model-task combinations
2. Rank the models with Batch Normalisation on and off. Here the computational budget of the models are controlled.

   The lower the ranking the better.
3. Plot the average/distribution of the ranking

Question: Is there any autoencoder task in the task space?

In the set of 32 tasks its all node/structure prediction tasks.

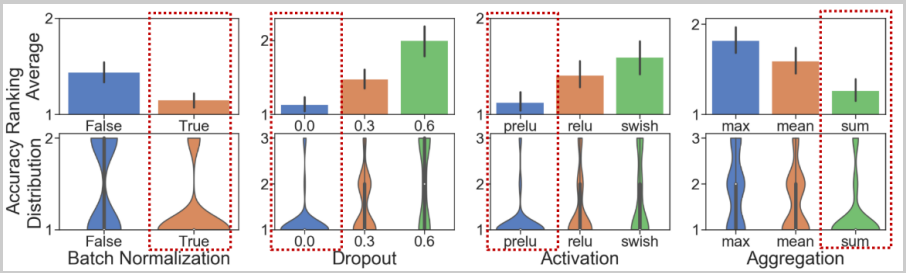
To apply this paradigm to novel task:

1. Measure 12 anchor model performance on the new task
2. Compute similarity between new task and existing tasks
3. Recommend the best designs from existing tasks with high similarity

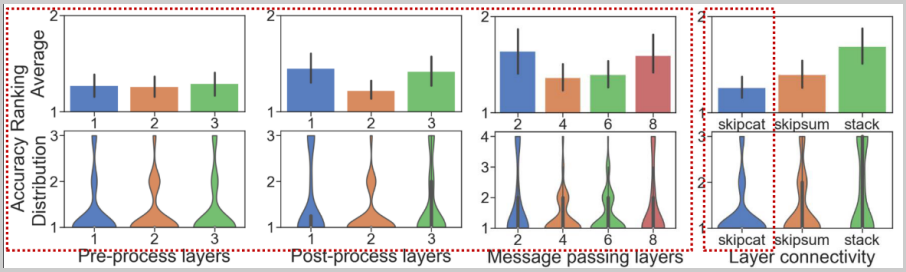
20.3 Pre-Training Graph Neural Networks

Challenges of applying ML to scientific domains

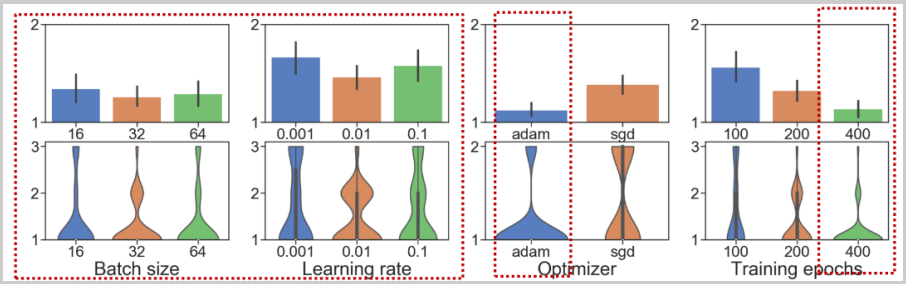
1. Scarcity of labeled data
2. Out-of-distribution prediction



(a) Intra-layer designs



(b) Inter-layer designs



(c) Learning configuration designs

Figure 20.3: Comparison of designs on a selection of 32 GNN tasks

Excursion: Pre-Training and Fine-Tuning

Deep learning models are extremely prone to overfitting on small labeled data and extrapolate poorly.
To improve a model's out-of-distribution prediction performance even with limited data, we can inject domain knowledge into a model before training on scarcely labeled tasks.

- **Pre-training**: Training a model on relevant tasks with large amounts of data
- **Fine-tuning**: Training a pre-trained model on a small number of out-of-distribution downstream task.

Settings

In this section we investigate a setting in molecule classification.

- **Task**: Binary classification of molecules

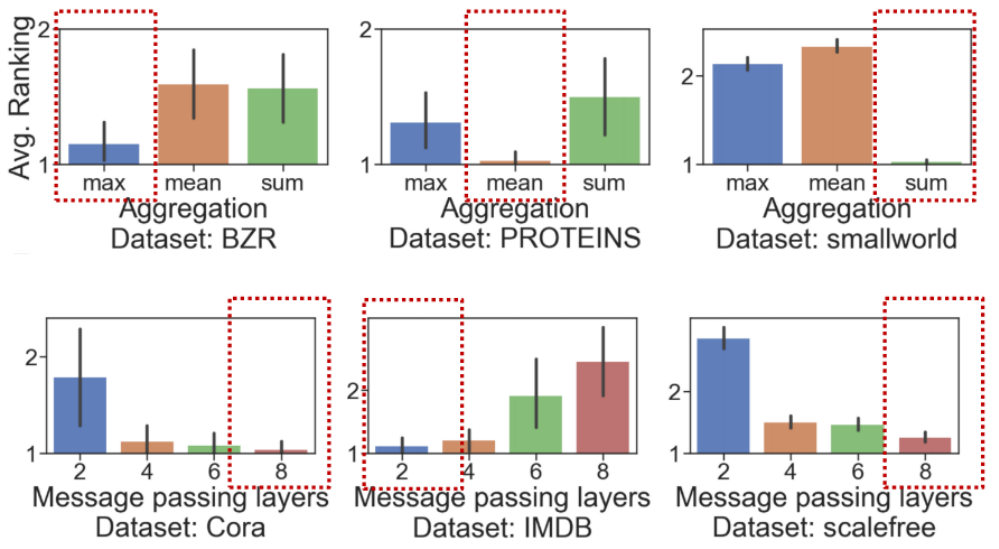
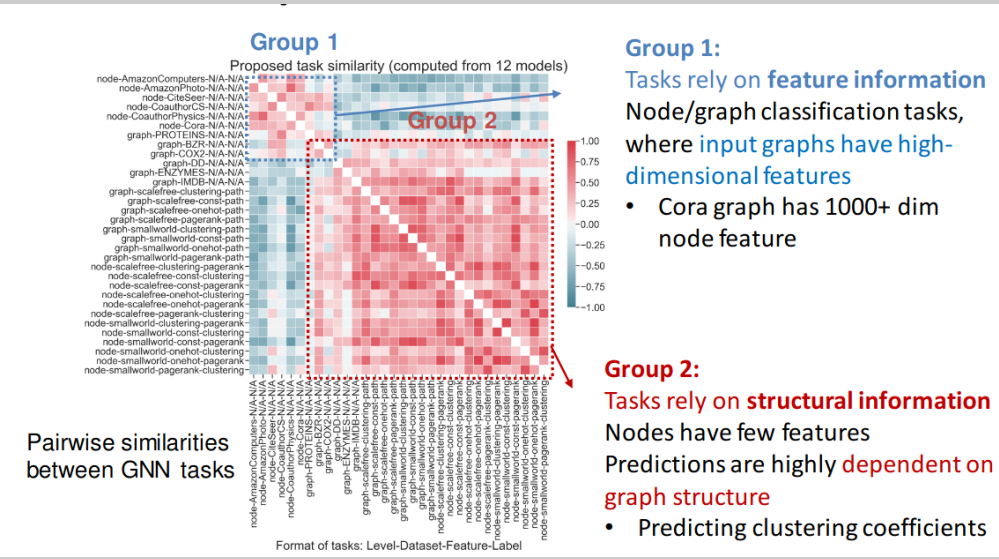


Figure 20.4: Best GNN designs in different tasks vary



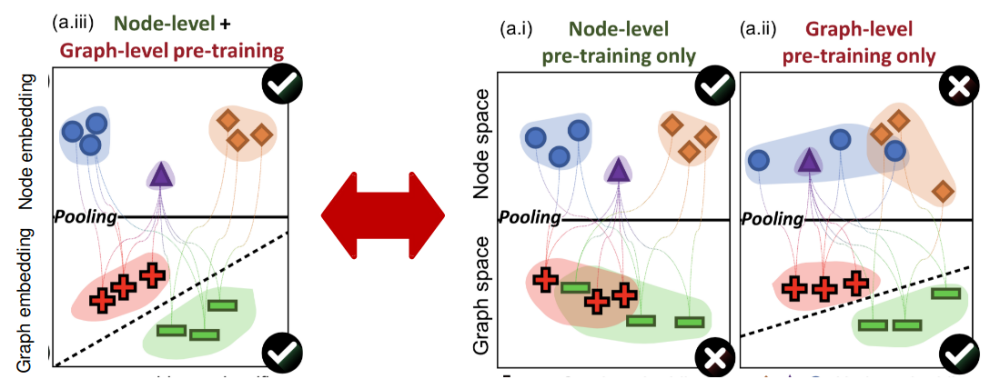


Figure 20.5: Pre-training both node and graph embeddings compared to training them individually

- **Evaluation metric**: ROC-AUC
- **Supervised pre-training data**: 1310 diverse binary bioassays annotated over 450, 000 molecules.
- **Downstream task (target)**: 8 molecular classification datasets, each with 1, 000 to 100, 000 molecules
- **Data split**: Scaffold (test molecules are out of distribution)

The Naïve strategy is multi-task supervised pre-training on relevant labels. This has limited performance on downstream tasks and often leads to negative transfer.
The key idea to improving this is to *pre-trian both node and graph embeddings.*
Pre-training methods:

- **Attribute Masking** (Node-lvel, self-supervised):

  1. Mask-node attributes and use GNNs to generate node embeddings.
  2. Use these embeddings to predict masked attributes (e.g. molecule)

  Intuitively, this forces the GNN to learn domain knowledge.

- **Context Prediction** (Node-level, self-supervised):

  1. For each graph, sample one centre node
  2. Extract neighbourhood and context
  3. Use GNNs to encode neighbourhood and context graphs into vectors
  4. Maximise/minimise inner product between true/false (neighbourhood, context) pairs

  Intuitively, subgraphs that are surrounded by similar contexts are semantically similar.

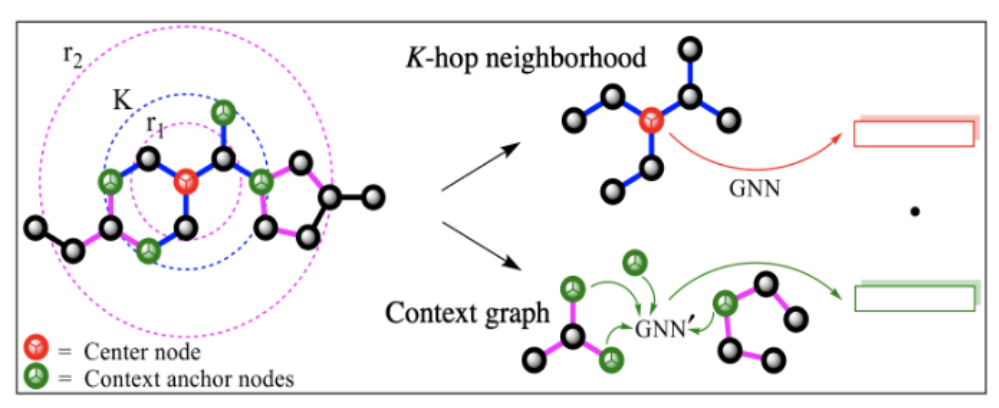


Figure 20.6: Context prediction

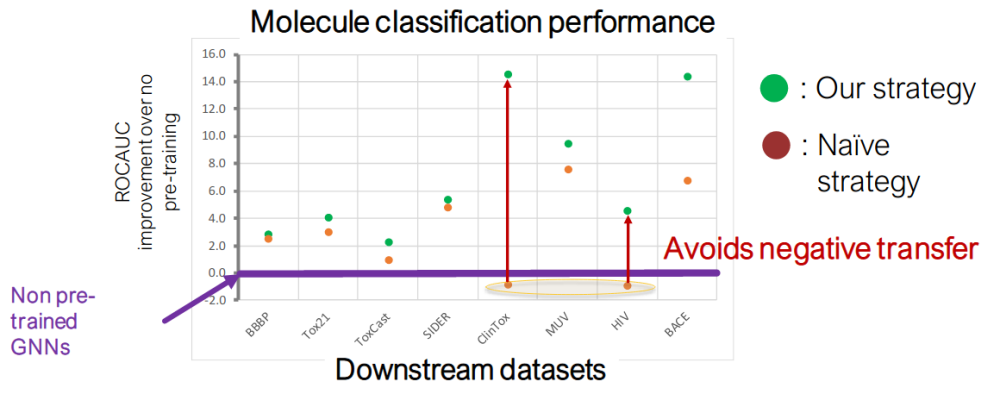


Figure 20.7: Comparison of naïve and effective graph pre-training

- **Supervised Attribute Prediction** (Graph-level):

  Multi-task supervised training on many relevant labels.

- **Structural Similarity Prediction** (Graph-level)

When different GNN models are pre-trained, the most expressive model (GIN) benefits the most from pre-training.