Graphs represent the relations between objects (nodes) and can have different attributes such as vertex attributes, edge attributes, and global attributes.

**V**

Vertex (or node) attributes

e.g., node identity, number of neighbors

**E**

Edge (or link) attributes and directions

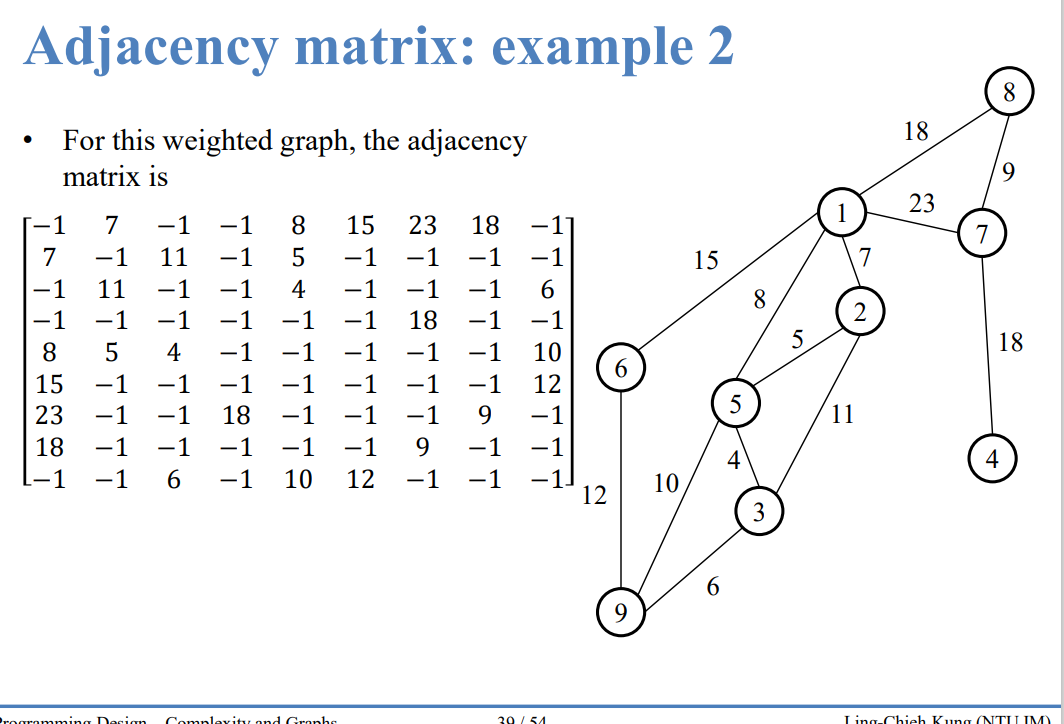
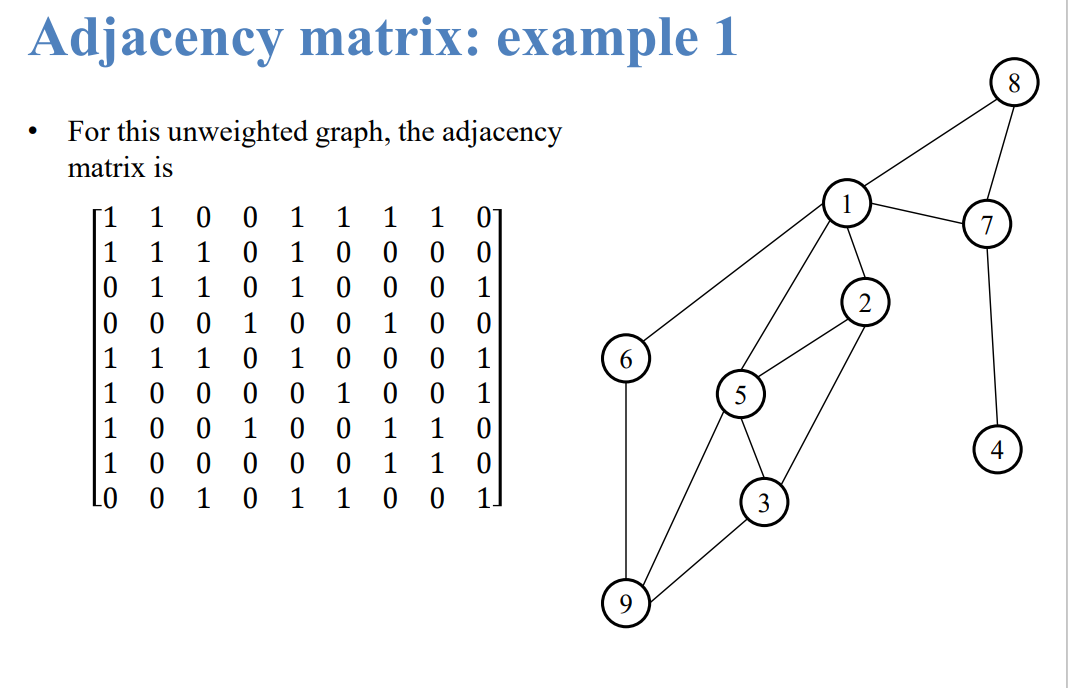
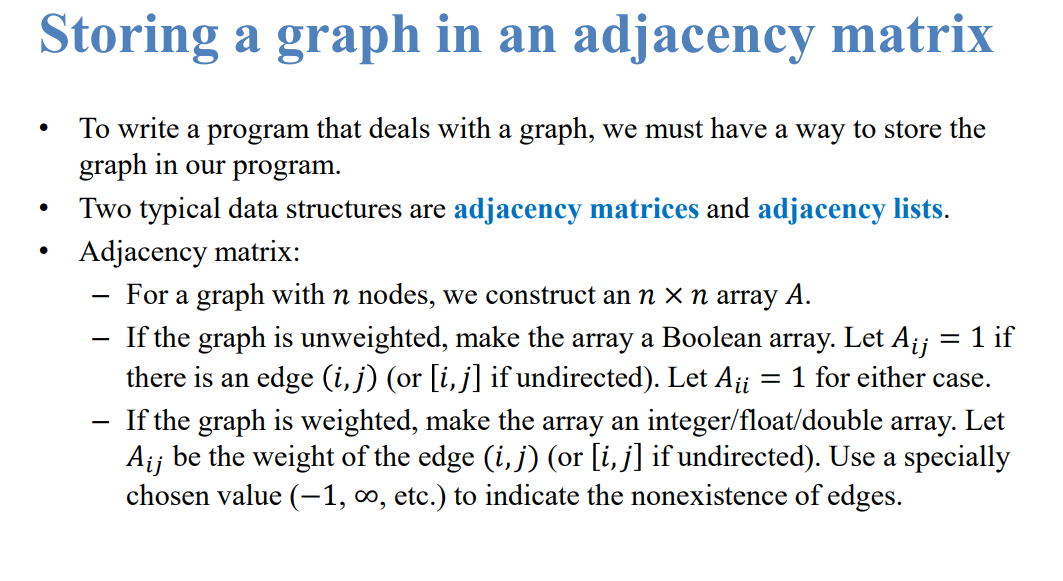
e.g., edge identity, edge weight

**U**

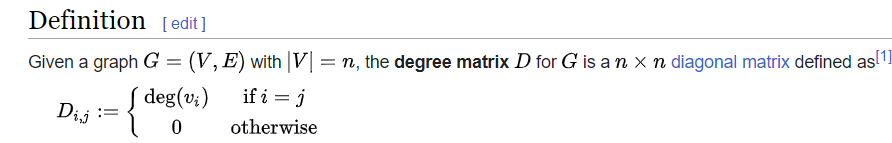
Global (or master node) attributes

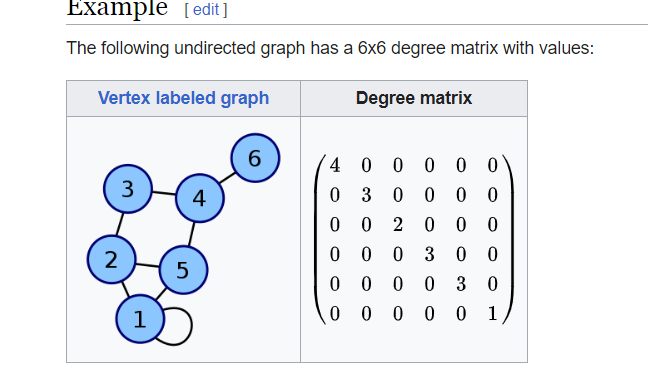
e.g., number of nodes, longest path

1.图的表示形式Adjacency matrix



1. 圖的表示形式——度矩陣( Degree matrix)





In Graph Neural Networks (GNNs), the degree matrix, adjacency matrix, and Laplacian matrix are derived from the graph's structure and used as input or intermediate representations during GNN computations.

There are several common representations used in Graph Neural Networks (GNNs) to capture and process graph-structured data. Some of the most commonly used representations in GNNs include:

Node Features: Each node in the graph can have associated feature vectors or attributes. These features capture information about the nodes themselves, such as node attributes, node embeddings, or node labels. Node features serve as input to GNNs and are typically represented as matrices or tensors.

Adjacency Matrix: The adjacency matrix is a binary or weighted matrix that represents the connections between nodes in the graph. It is commonly used in GNNs to define the graph's structure and capture pairwise relationships between nodes.

The adjacency matrix represents the connections between nodes in the graph. It is a square matrix where each element represents the presence or absence of an edge between two nodes. The adjacency matrix is used to define the graph structure and can be transformed or normalized for GNN computations.

Degree Matrix: The degree matrix is a diagonal matrix that contains the degrees of each node in the graph. The degree of a node is the number of edges connected to it. The degree matrix is typically used to normalize the adjacency matrix or Laplacian matrix to account for varying degrees of nodes in the graph.

Laplacian Matrix: The Laplacian matrix is derived from the adjacency matrix and represents the graph Laplacian. It provides information about the graph's connectivity and can be used for spectral analysis or to define graph convolutions in GNNs.

Laplacian Matrix: The Laplacian matrix is derived from the adjacency matrix and represents the graph Laplacian. It provides information about the graph's connectivity and can be used to analyze the graph's spectral properties. In GNNs, variants of the Laplacian matrix, such as the normalized Laplacian, are used to perform graph convolutions and propagate information between nodes.

Message Passing: GNNs often use message passing mechanisms to propagate information across nodes in the graph. Nodes exchange messages based on their neighboring nodes' features, allowing information to flow and be aggregated throughout the graph.

Spatial GNN

how to represent a picture with graph in GNN?

Representing a picture as a graph in Graph Neural Networks (GNNs) typically involves constructing a graph where the nodes and edges capture the spatial relationships between pixels or image regions. Here's a common approach for representing a picture as a graph in GNNs:

Nodes: Each pixel or image region in the picture is represented as a node in the graph. The number of nodes corresponds to the total number of pixels or image regions in the picture.

Edges: The edges represent the spatial relationships between nodes (pixels or image regions). There are different strategies for defining edges:

a. Grid Structure: If the picture has a regular grid structure, such as in a typical image, you can define edges between adjacent pixels. For example, each pixel can be connected to its neighboring pixels (e.g., up, down, left, right).

b. Delaunay Triangulation: In some cases, you may want to define edges based on the Delaunay triangulation of the pixel coordinates. Delaunay triangulation creates triangles connecting the pixels in a way that maximizes the minimum angle between edges.

c. Superpixels: Instead of using individual pixels as nodes, you can use superpixels as nodes. Superpixels group neighboring pixels into coherent regions, reducing the graph size and complexity.

Node Features: Each node (pixel or superpixel) can have associated features. These features can include color values, texture information, position coordinates, or other visual attributes. These features are typically represented as feature vectors or embeddings.

GCN(有max pooling等aggregation的圖神經網絡)

when applying Graph Neural Networks (GNNs) to obtain node features from images, it is common to incorporate Convolutional Neural Networks (CNNs) into the GNN architecture. This combination is often referred to as Graph Convolutional Neural Networks (GCNs).

The process typically involves the following steps:

Initial Node Features: Each node in the graph represents a pixel or an image region. The initial node features can be derived from the input image, such as the color values, texture information, or other visual attributes associated with each pixel or region.

Graph Construction: The graph structure is established by connecting nodes based on spatial relationships, as discussed earlier. This can be done using a grid structure, Delaunay triangulation, or superpixels, depending on the specific representation chosen.

用途：

1.Relationship of items or people

2. a sequence of speech

3.citation

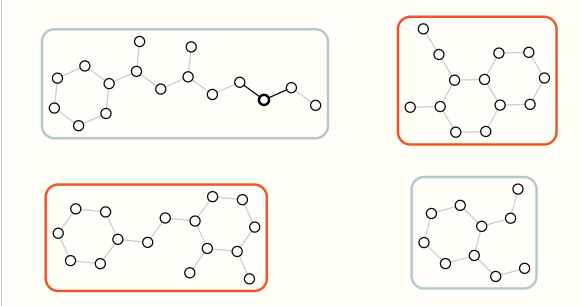
4.分子結構

4.knowledge graph

What task do we want to perform on this data?what problem do we want to solve?

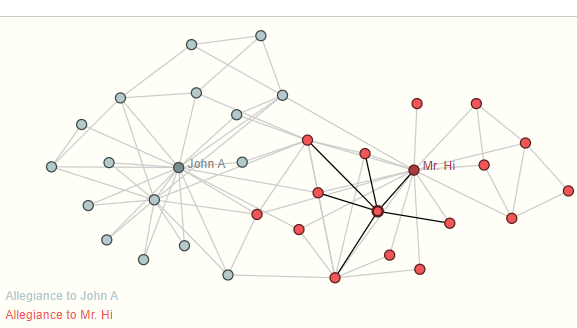
1. graph-level task

In a graph-level task, our goal is to predict the property of an entire graph. For example, for a molecule represented as a graph, we might want to predict what the molecule smells like, or whether it will bind to a receptor implicated in a disease.



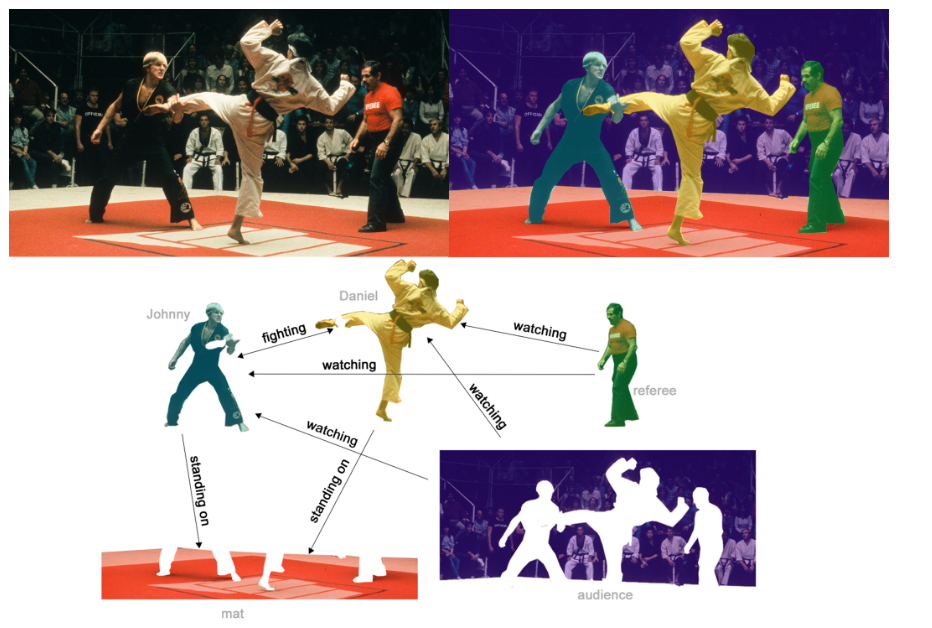
1. Node-level task

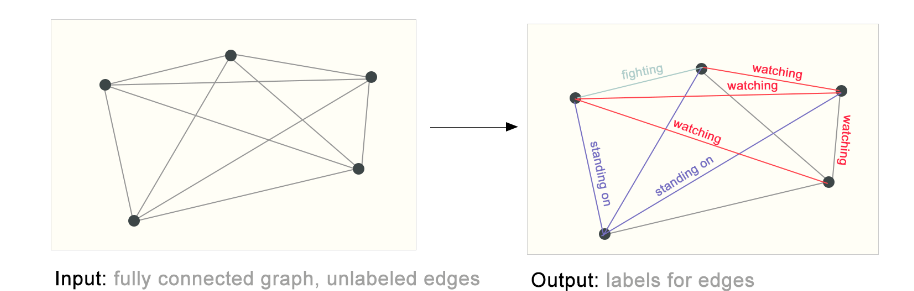
Node-level tasks are concerned with predicting the identity or role of each node within a graph.node classification



1. Edge-level task

The remaining prediction problem in graphs is edge prediction.





## **The challenges of using graphs in machine learning**

- Representing graphs for compatibility with neural networks:

- Nodes, edges, global context, and connectivity are key components of graph representation.

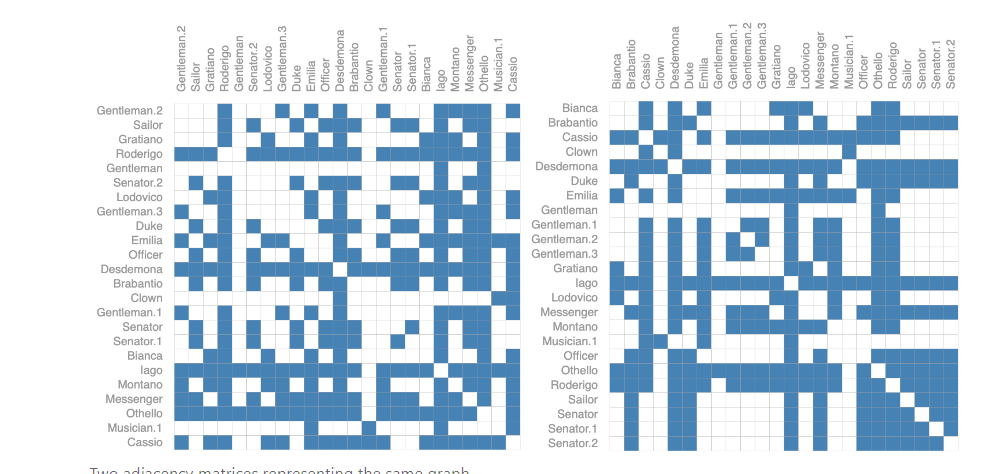
- Node features can be stored in a node feature matrix.

- Connectivity representation is more complex and often done using adjacency matrices.

- Challenges with adjacency matrix representation:

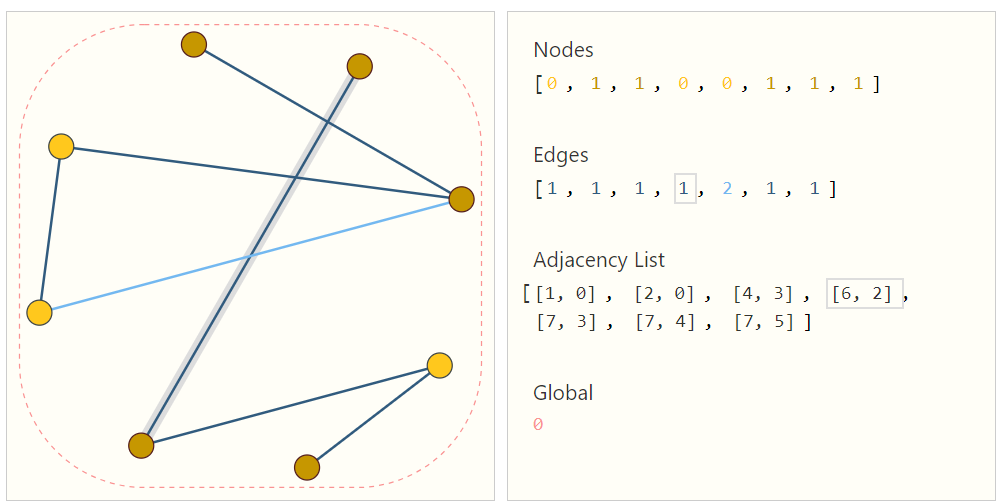
1. Sparse adjacency matrices can be space-inefficient for large graphs.

2.Different adjacency matrices can encode the same connectivity, leading to non-permutation invariance.把任何行的順序和列的順序交換都不會影響它（The problem arises because there can be different permutations or reorderings of the nodes that maintain the same underlying connectivity. ）



- Recent research focuses on learning permutation invariant operations.

如果想不受順序影響又想表達connectivity的話可以用以下方法：（adjacency list）



每個node可以是scalar也可以是vector

The length of Adjacency list is same as the number of edge

- Graph representation for images in GNNs:

- Grid Structure: Define edges between adjacent pixels in a regular grid structure.

- Delaunay Triangulation: Define edges based on the Delaunay triangulation of pixel coordinates.

- Superpixels: Use superpixels as nodes to group neighboring pixels into coherent regions.

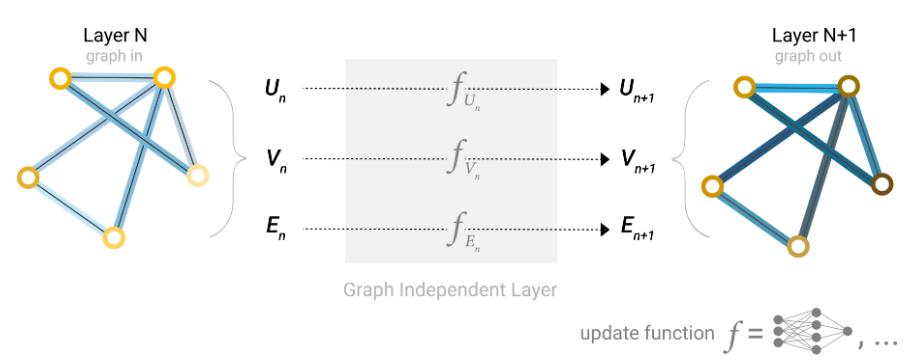
- Node features in image-based graphs:

- Each node (pixel or superpixel) can have associated features such as color values, texture information, or position coordinates.

- Features are typically represented as feature vectors or embeddings.

**A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances)**

GNNs adopt a “graph-in, graph-out” architecture meaning that these model types accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph.

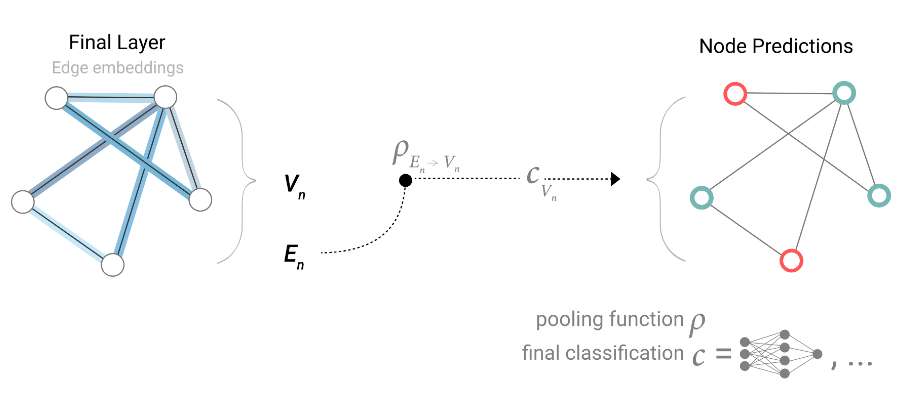


### **The simplest GNN,layer to layer**

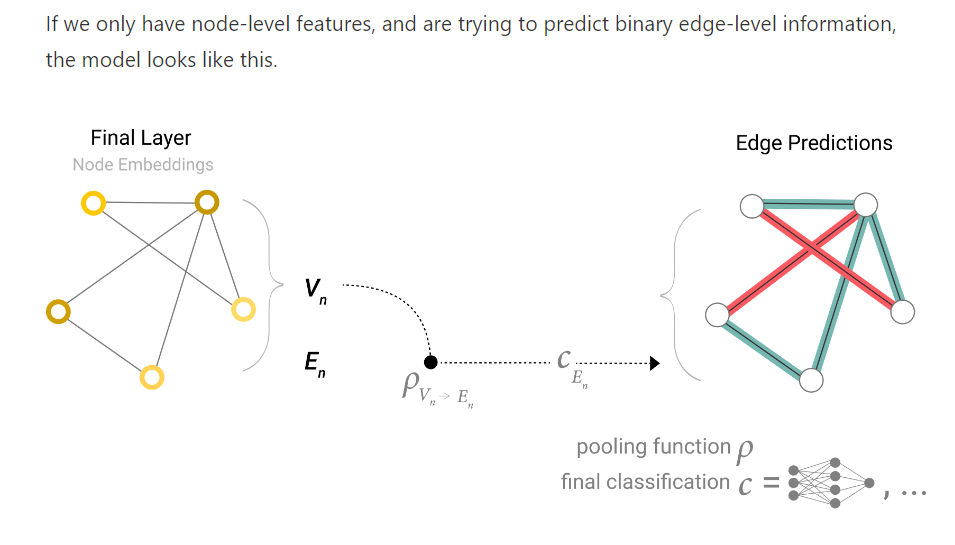
With the numerical representation of graphs that [we’ve constructed above](https://distill.pub/2021/gnn-intro/?fbclid=IwAR0z3N0TA9XB9SoFVYyDDZu1eYQ2QvmPYHrmmoQDHIAHxqAM7ronH72gQ-Y" \l "graph-to-tensor) (with vectors instead of scalars), we are now ready to build a GNN. We will start with the simplest GNN architecture, one where we learn new embeddings for all graph attributes (nodes, edges, global), but where we do not yet use the connectivity of the graph.

For simplicity, the previous diagrams used scalars to represent graph attributes; in practice feature vectors, or embeddings, are much more useful.

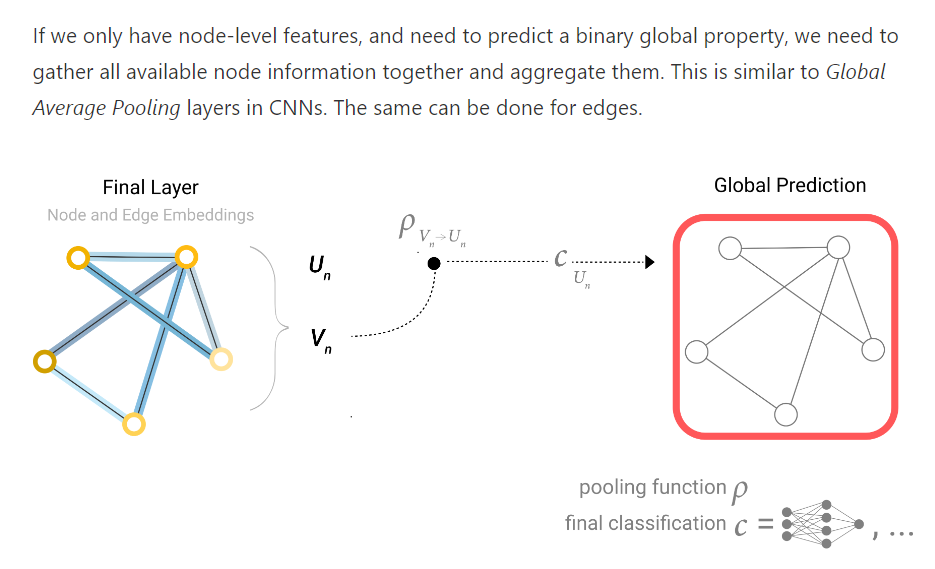
This GNN uses a separate multilayer perceptron (MLP) (or your favorite differentiable model) on each component of a graph; we call this a GNN layer. For each node vector, we apply the MLP and get back a learned node-vector. We do the same for each edge, learning a per-edge embedding, and also for the global-context vector, learning a single embedding for the entire graph.



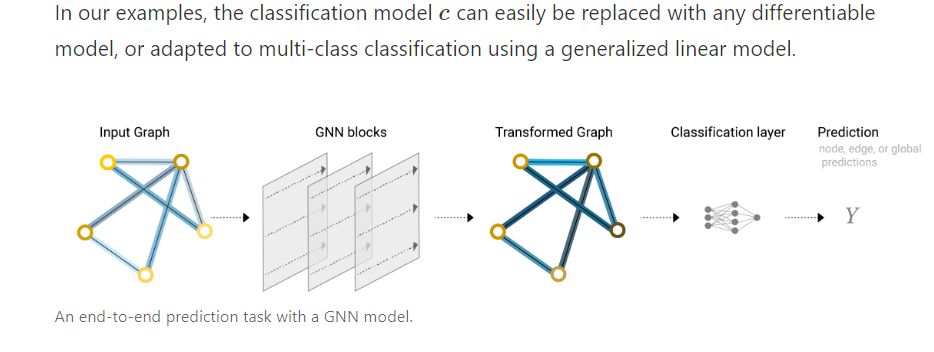
Edge prediction：



If 只有node 的vector 但沒有global 的vector：



GNN block每一層有三個MLP，對應的是三種屬性 node edge global



GNN block 的部分沒有考慮到哪些點跟哪些邊相連，或者哪些頂點是連在一起的信息，導致最後的圖並不能夠充分leverage整個圖的信息。

所以要用到massage passing，aggregation

The architecture for GNN:

Here is an overview of the architecture for multilayer GNNs:

Initialization: Initially, each node in the graph is assigned an initial feature representation.

Message Passing and Update (Layer 1): In the first layer, nodes perform message passing, aggregating information from their neighboring nodes. The aggregated information is then used to update the node features within the same layer.

Iteration (Layer 1): The message passing and update steps are performed iteratively for Layer 1, allowing nodes to refine their features by considering information from a local neighborhood.

Output Generation (Layer 1): After the desired number of iterations or convergence in Layer 1, the updated node features are used to generate intermediate outputs or representations.

Message Passing and Update (Subsequent Layers): The output from Layer 1 serves as the input for the next layer. In each subsequent layer, nodes perform message passing and update their features using the aggregated information from the previous layer.不管有多少個node，只會有一個fully connected layer，所有的node 共享一個fully connected layer 的參數

Iteration (Subsequent Layers): The message passing and update steps are performed iteratively for each subsequent layer, allowing nodes to incorporate information from a broader neighborhood and refine their features.

Output Generation (Final Layer): After the desired number of iterations or convergence in the final layer, the updated node features are used to generate the final output, which could be node-level predictions, graph-level predictions, or other relevant tasks.

A different perspective:

GNN Layer: Each GNN layer in the architecture operates on the graph's components (nodes, edges, and global context) independently using separate MLPs.

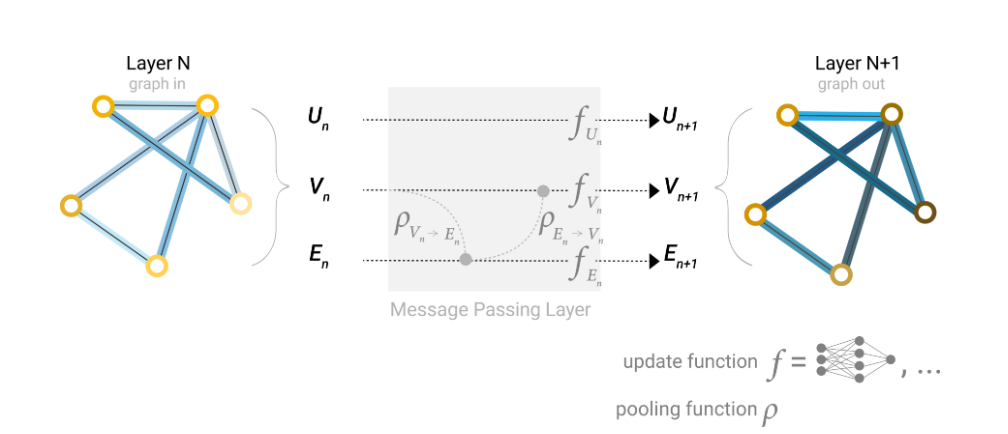
Node MLP: The node MLP takes the node vectors as input and applies a set of learnable transformations to obtain updated node embeddings. This process allows each node to refine its representation based on its local information.

Edge MLP: The edge MLP takes the edge embeddings as input and applies learnable transformations to obtain updated edge embeddings. This step enables the model to capture edge-specific information or relationships between nodes.

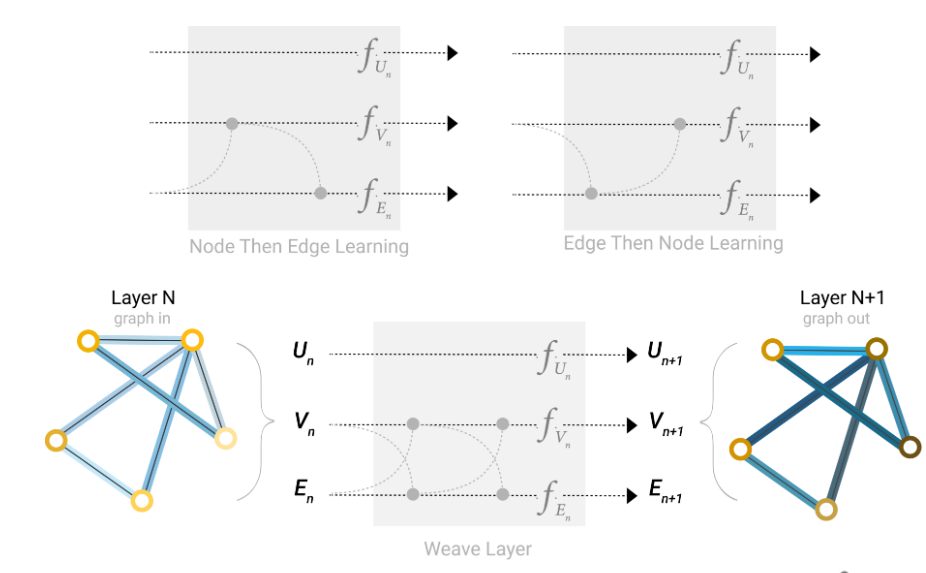
Global-Context MLP: The global-context MLP takes the global-context vector (representing the entire graph) as input and applies learnable transformations to obtain an updated global-context embedding. This step allows the model to capture overall graph-level information or patterns.

Stack of GNN Layers: The architecture involves stacking multiple GNN layers on top of each other. Each layer independently applies the MLPs to the corresponding components of the graph, allowing information to flow and be refined through multiple layers.

MLP時因把鄰點的影響也考慮進去(aggregation)，所以layers到最後會包含鄰點的鄰點...等多重關係都考量在內。

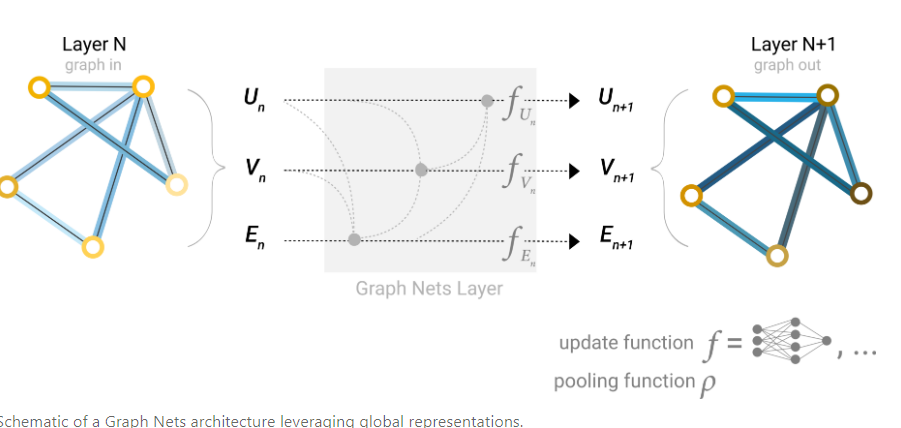
MLP時也可以把頂點的向量傳給邊，然後再傳回給下一個頂點，例如把每個邊鏈接的兩端頂點的信息加到邊自己的vector裡面（massage passing in same layer）。每個頂點也可以把他連的邊的那些信息都加進自己的vector裡面，然後再進入各自的MLP做更新。

Massage passing的先後順序會影響結果，可以交替更新（同時做）互相aggregate 舊的作update



How about global information?

為何要有global（U跟Vnode 和E edge相連），因為假設圖很大，一個點的信息傳遞到另一個點很慢，所以加入一個master node 或context vector的東西。這個點是虛擬的，假設可以跟所有的點和邊相連。



Update完E和V後再aggregate到global裡面，再放進去MLP做下一個global的更新

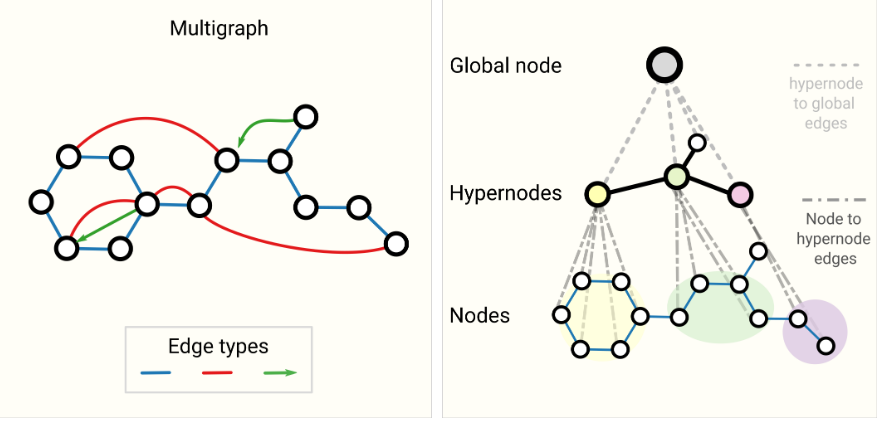
Aggregation不只是sum，還有max pooling、average pooling！！！三種沒太大不一樣，但如果上下層value一樣，max pooling和average pooling無法detect不同

通常是taking a weighted sum or concatenation of the neighboring node features

Model evaluation metric is AUC, the greater is best

### Other type of graph:**(multigraphs, hypergraphs, hypernodes, hierarchical graphs)**

頂點可以同時包含有向邊和無向邊,node也可以是分層的



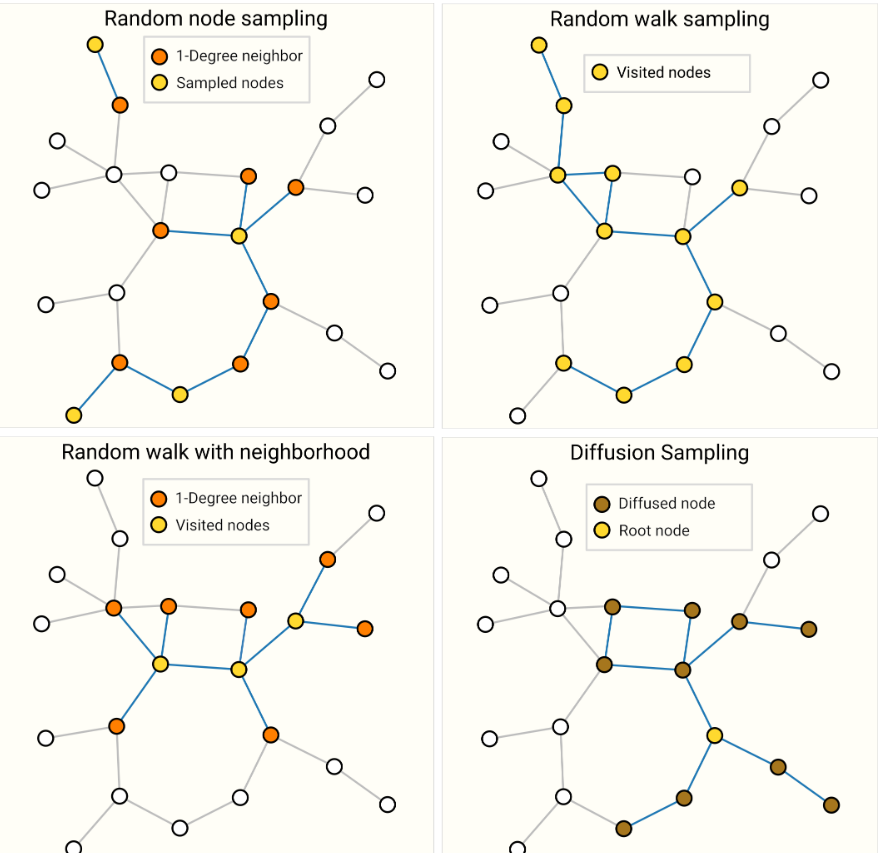
怎麼對圖做sampling and batching

A common practice for training neural networks is to update network parameters with gradients calculated on randomized constant size (batch size) subsets of the training data (mini-batches). 假設有很多層，最後那一層的頂點因為囊括了很多信息，所以也是一個全局的信息。計算gradient的時候會把中間的變量存下來，如果最後一個頂點要看全圖的話，意味著要把整個圖的gradient都存下來，所以可能無法承受這種運算。最後就是sample一個小sample出來做信息的匯聚，再做gradient

Method 1：隨機採樣一些點，再把他們的鄰居找出來，在計算的時候只在這個子圖上做計算

Method2：做一些隨機遊走，在這個子圖上計算

Method3：隨機走幾步，再把鄰點找出來。

Method4：取一個點，把他的鄰居都找出來，然後往前走k步

Batching

不想對每一個點做單獨運算，而是想把頂點像sampling一樣做成一個一個小的batch，但問題是，頂點的鄰居數量是不一樣的，怎麼把頂點和鄰居合並成一個規則的vector是有挑戰性的問題。

Inductive Biases

graph symmetries (permutation invariance)，假設頂點順序不管怎麼變，都不會改變GNN對他的作用

匯聚跟卷積的差別，匯聚是把自己與相鄰的node加在一起，但卷積則是用filter做一個權重的transformation，filter的權重跟位置在圖神經網絡中沒有關係。所以我們需要讓權重對位置不敏感，所以可以用注意力機制

Graph attention network

你的權重取決於兩個node的向量之間的關係。

Coding：

import torchfrom torch.nn import Linearfrom torch\_geometric.nn import GCNConv

class **GCN**(torch.nn.Module):

def \_\_init\_\_(self):

super(GCN, self).\_\_init\_\_()

torch.manual\_seed(12345)

self.conv1 = GCNConv(dataset.num\_features, 4)

self.conv2 = GCNConv(4, 4)

self.conv3 = GCNConv(4, 2)

self.classifier = Linear(2, dataset.num\_classes)

def forward(self, x, edge\_index):

h = self.conv1(x, edge\_index)

h = h.tanh()

h = self.conv2(h, edge\_index)

h = h.tanh()

h = self.conv3(h, edge\_index)

h = h.tanh() *# Final GNN embedding space.*

*# Apply a final (linear) classifier.*

out = self.classifier(h)

return out, h

model = GCN()print(model)

import torch

edge\_index = torch.tensor([[0, 1, 2, 2, 3],

[1, 0, 1, 3, 2]])

Edge 0: Node 0 -> Node 1

Edge 1: Node 1 -> Node 0

Edge 2: Node 2 -> Node 1

Edge 3: Node 2 -> Node 3

Edge 4: Node 3 -> Node 2

model = GCN()criterion = torch.nn.CrossEntropyLoss() *#Initialize the CrossEntropyLoss function.*optimizer = torch.optim.Adam(model.parameters(), lr=0.01) *# Initialize the Adam optimizer.*

def train(data):

optimizer.zero\_grad() *# Clear gradients.*

out, h = model(data.x, data.edge\_index) *# Perform a single forward pass.*

loss = criterion(out[data.train\_mask], data.y[data.train\_mask]) *# Compute the loss solely based on the training nodes.*

loss.backward() *# Derive gradients.*

optimizer.step() *# Update parameters based on gradients.*

return loss, h

for epoch **in** range(401):

loss, h = train(data)

print(f'Epoch: **{**epoch**}**, Loss: **{**loss**}**')