Lecture 13

Graph Neural Networks

cscis-89 Deep Learning, Fall 2024

Zoran B. Djordjević & Rahul Joglekar

References

- “Graph Representation Learning” by William L. Hamilton, Morgan & Claypool 2020, book available at: https://www.cs.mcgill.ca/~wlh/grl_book/files/GRL_Book.pdf
- “Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges” by Michael M. Bronstein, Joan Bruna, Taco Cohen, Petar Veličković, https://arxiv.org/abs/2104.13478
- Bronstein, Michael M. et al. "Geometric deep learning: going beyond Euclidean data." IEEE Signal Processing Magazine 34.4 (2017): 18-42.” https://arxiv.org/abs/1611.08097
- Liu, Zhiyuan; Zhou, Jie, Introduction to Graph Neural Networks, Morgan & Claypool, 2020.

- The best resource for learning GNNs is the course:

  CS224W: Machine Learning with Graphs, Jure Leskovec, Stanford University

  http://cs224w.stanford.edu

  These notes borrow heavily from cs224W materials.
- For Python network aware software look for Networkx package https://networkx.org/documentation/stable/tutorial.html
- An excellent book on networks: “Network Science” by Alberto-Laszlo Barabas. Available in online version: http://networksciencebook.com/

Networks and/or Graphs

Graphs

- Graphs are data structures and a universal language for describing complex systems. In the most general terms, a graph is a collection of objects (i.e., nodes), along with a set of interactions (i.e., edges) between pairs of those objects.

- **Graphs are a general language for describing and analyzing entities with relations/interactions.**

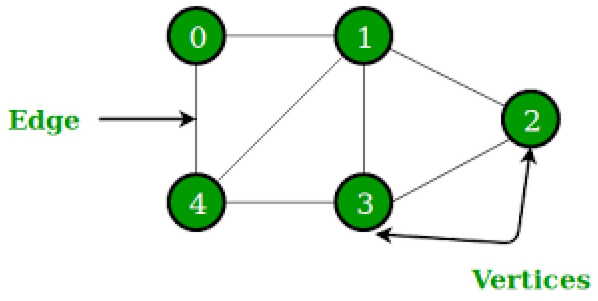
- We can find graphs (network structures) in nature (molecules), society (social networks), technology (the internet), and everyday settings (roadmaps).

- For example, to encode a social network as a graph we use nodes to represent individuals and use edges to represent the friendship or some other relationship between the individuals.
- In the biological domain, we could use the nodes to represent proteins, and use the edges to represent various biological interactions between proteins.

- To perform machine learning on graphs, we need a specialized form of the neural networks, the graph neural network or GNN.

Graphs vs Grid-like Structures

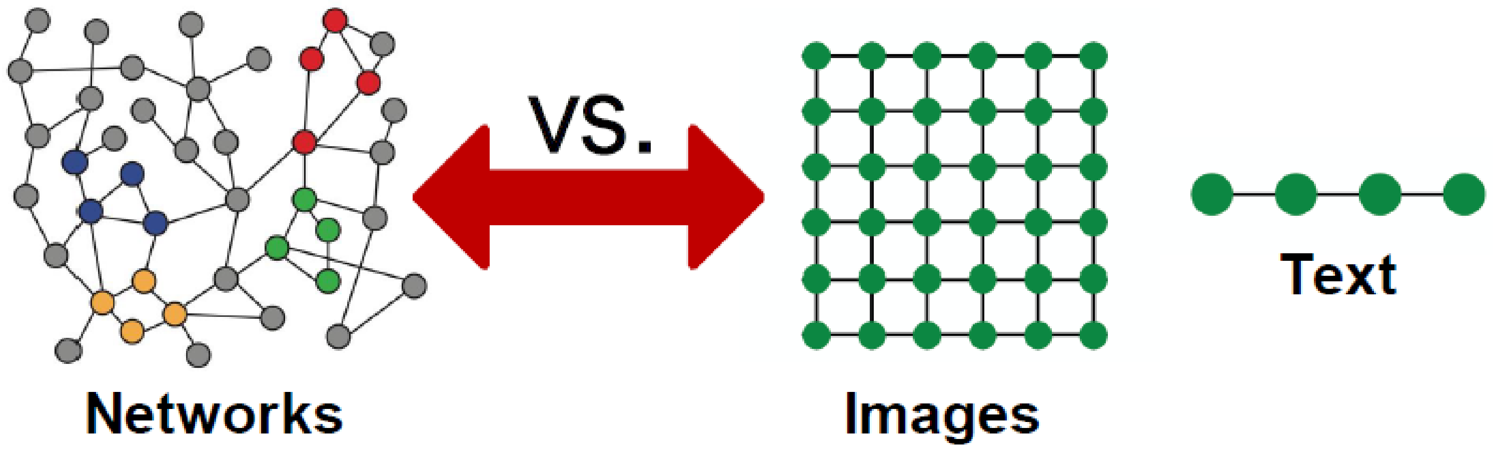
- Graphs or networks are data structures whose key elements are relationships. Graphs are often represented as images or diagrams, with edges representing relationships and nodes, or vertices representing entities. The diagram below is one representation of a graph. On subsequent slides we will see that graphs could be represented by matrices and tensors.
- In graphs or networks, the relationships are the first-class citizens.
- In more traditional learning and analytics approaches, we use grid-like structure such as the relational database tables, Pandas `DataFrames`, or Excel spreadsheets.
- In non-network approach, relationships play much less pronounced role than in the networks or graphs



- In above diagram, circles or nodes (vertices) represent entities. Lines or edges represent relationships. The numbers could be used to indicate node IDs. Nodes could be named and could have many attributes. Edges could be similarly labeled by edge IDs or names. Edges (relationships) could have different attributes as well.

What is different in Graphs

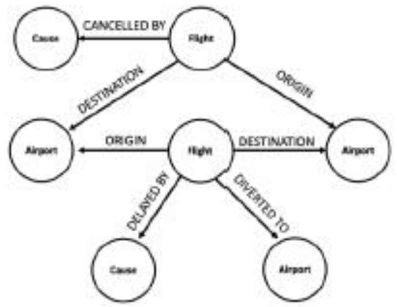
- Networks and graphs could be complex.
- Graphs could have arbitrary size and topological structure without locale spatial uniformity like grids.



- Network or graphs have no fixed node ordering or reference point
- Graphs are often dynamic and have multimodal features
- One can safely argue that images and text are also graphs, just with some additional structural rules or symmetries. Typically, when dealing with such “simplified or ordered” graphs, we insist on relaying on those extra rules and symmetries.

Application of Graphs

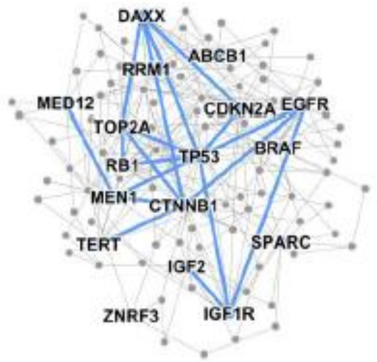
Applications of Graphs



Event Graphs

Image credit: SalientNetworks

Computer Networks



Disease Pathways

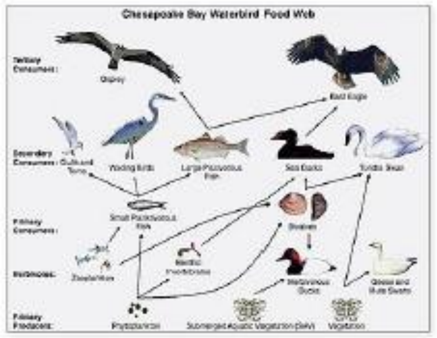


Image credit: Wikipedia

Food Webs

Image credit: Pinterest

Particle Networks

Image credit: visitlondon.com

Underground Networks

Applications of Graphs

Image credit: Medium

Social Networks

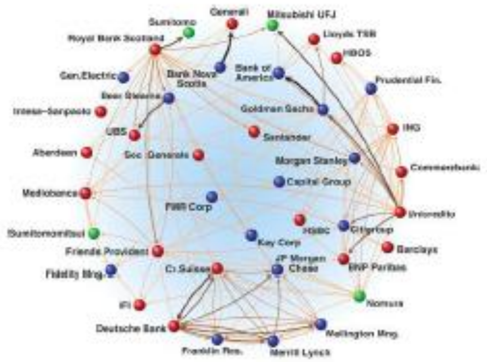
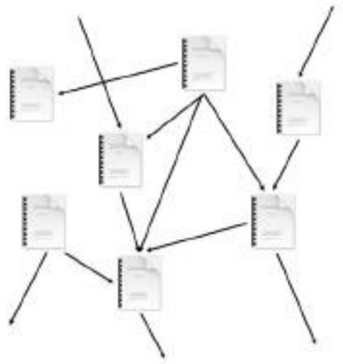


Image credit: Science

Economic Networks

Image credit: Lumen Learning

Communication Networks



Citation Networks

Image credit: Missoula Current News

Internet

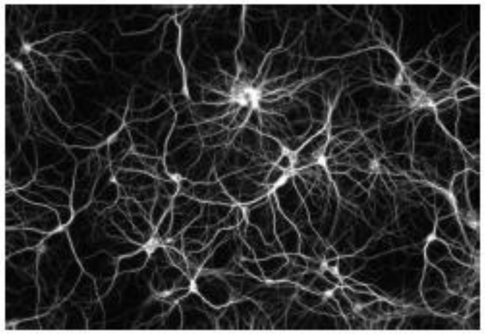


Image credit: The Conversation

Networks of Neurons

Applications of Graphs

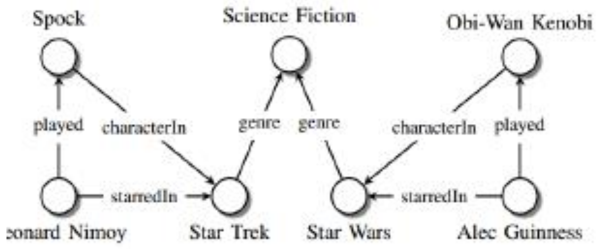


Image credit: Maximilian Nickel et al

Knowledge Graphs

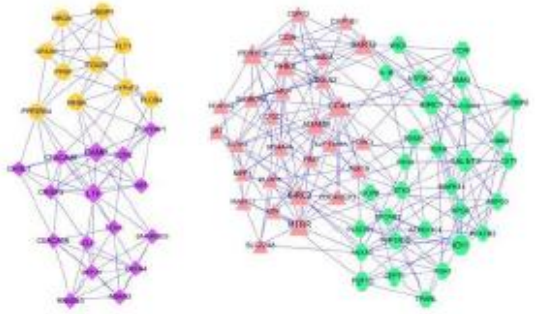


Image credit: ese.wustl.edu

Regulatory Networks

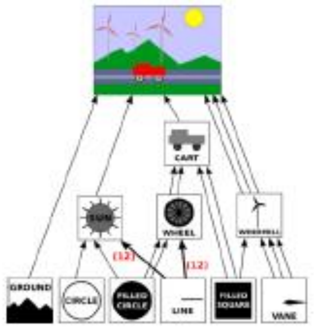


Image credit: math.hws.edu

Scene Graphs

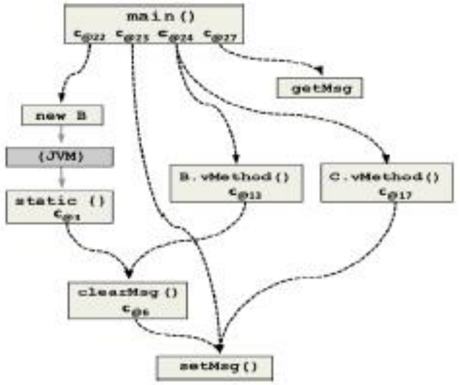


Image credit: ResearchGate

Code Graphs

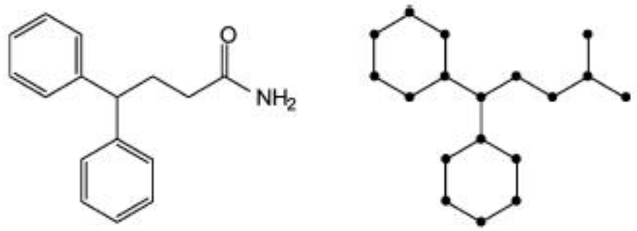


Image credit: MDPI

Molecules

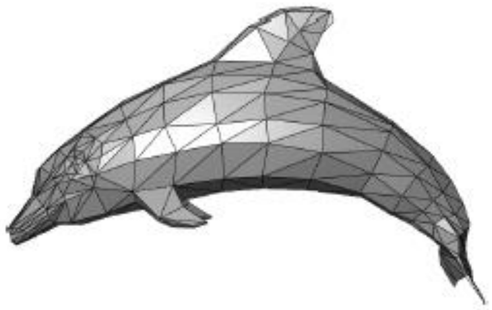
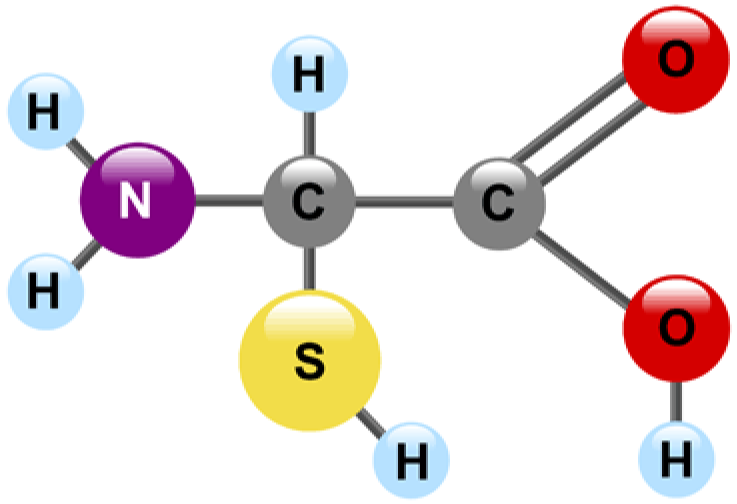


Image credit: Wikipedia

3D Shapes

Molecular Fingerprints and Interfaces

- In chemistry and molecular sciences, a prominent problem is how to represent molecules in a general application-agnostic way, and inferring possible interfaces between organic molecules, such as proteins. Some molecule representation bear resemblance to a graph structure, consisting of nodes (atoms) and edges (atomic bonds).



- Applying Graph Convolutional Networks to structure of molecules is a nascent field that promises to outperform traditional ‘fingerprint’ methods, which involve the creation of features by domain experts to capture molecular properties.

What could we do with graph data & GNNs

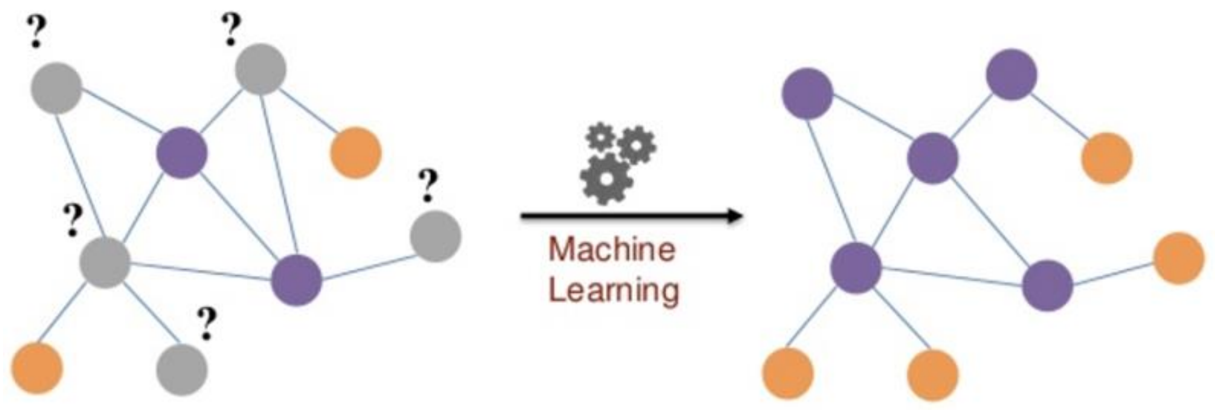
- Assuming one had access to the graph data and it is of sufficient quantity and quality, what could a GNN extract from such data?
- Historical records are not always complete. We could use GNNs to fill in the missing information.
- GNNs could perform **node classification** to predict node attributes. For example, on the list of Titanic passengers, we could predict missing passenger information. For example, if the citizenship information was missing from some of the passenger data, we could use node classification to uncover the citizenship labels.
- GNNs could perform **edge prediction** (or link prediction) to uncover hidden or missing links between nodes. In the Titanic case, we could use it to find non-obvious relationships between the passengers, like extended family relationships (cousins on the ship), and business relationships.
- **Graph-level tasks like classification and regression** involve predicting the label or a quantity associated with the entire graph.
- There are also unsupervised tasks that involve GNNs. These involve Graph Auto Encoders that embed graphs and do the opposite process of generating a graph from an embedding. There is also a class of models that use adversarial methods to generate graphs.

Prediction Tasks

- A slew of graph algorithms and analytical methods has existed for decades, for a variety of use cases. Many of these methods are quite powerful, are used at scale, and are very successful.
- **PageRank**, the first algorithm used by Google to order web search results, is probably the most famous example of a graph algorithm used to great success by an enterprise.
- Applications where graph neural networks shine are problems that require predictive models. We either use the GNN to train such a model, or we use it to produce a representation of the graph data that can be used in a downstream model.

Node-Level vs Edge-Level tasks

- Nodes in graphs often have attributes and labels. Node classification is the **node-level task** analog to the traditional machine learning classification: given a graph with several node classes, predict the class of an unlabeled node.



- For example, we want to find out what are the true colors of gray nodes?
- **Edge-level tasks** are very similar to node-level tasks. Link classification involves predicting a link's label in a supervised or semi-supervised way. Edge Prediction is inferring a link between nodes, where one may not be provided in the graph under study.

Types of Graphs and Terminology

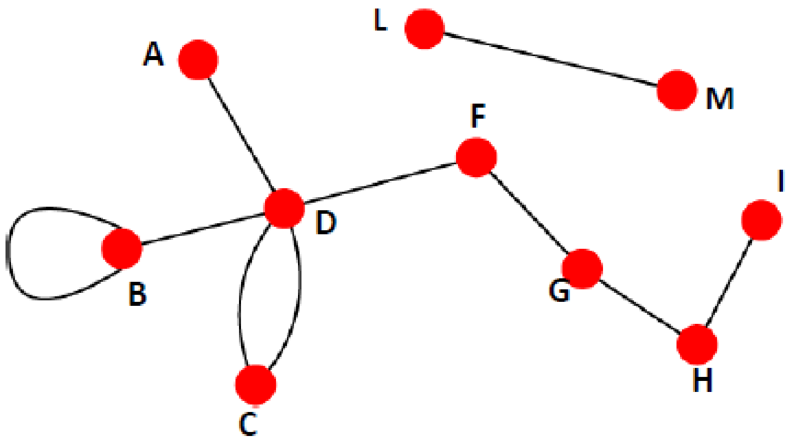
Key Terms

- **Graphs** are formally defined by a set of vertices and a set of edges between these vertices: G = G (V, E). Fundamentally, graphs are just a way to encode data. Every property of a graph represents some real element, or concept. Graphs can be used to represent complex concepts.
- **Vertices** represent items, entities, or objects, which have quantifiable attributes and relationships to other items, entities, or objects. We refer to a set $\{vi\}$ of vertices as V. In a graph, not all vertices have to be homogenous with the same set of properties.
- **Edges** represent and characterize relationships between items, entities, or objects. Formally, a single edge can be defined with respect to two vertices. We label the set $\{e_{ij}\}$ of edges as E. e_{ij} is a single edge between the i-th and j-th vertices. There could be more edges connecting the same pair of vertices.
- **Neighborhoods** are subgraphs within a graph and represent distinct groups of vertices and edges. Most commonly, the neighborhood N_{v_i} is centered around v_i , and contains v_i itself, its adjoining edges (e_{ij}), and the vertices that are directly connected to node (vertex) v_i. Neighborhoods can be iteratively grown from a single vertex by considering the vertices attached (via edges) to the current neighborhood. Note that a neighborhood can be defined subject to certain vertex and edge feature criteria (i.e., all vertices within 2 hops of the central vertex, rather than 1 hop).
- **Features** are quantifiable attributes which characterize a phenomenon that is under study. In the graph domain, features characterize vertices and edges. In social networks, we might have features for each person (vertex) specifying the person's age, popularity, and social media usage. Similarly, we might have a feature for each relationship (edge) which quantifies how well two people know each other, or the type of relationship they have (familial, collegial, etc.). In practice, there might be many different features for each vertex or edge, and they are usually represented by numeric feature vectors referred to as F_{v_i} and $F_{e_{ij}}$, respectively.

Undirected vs. Directed Graphs

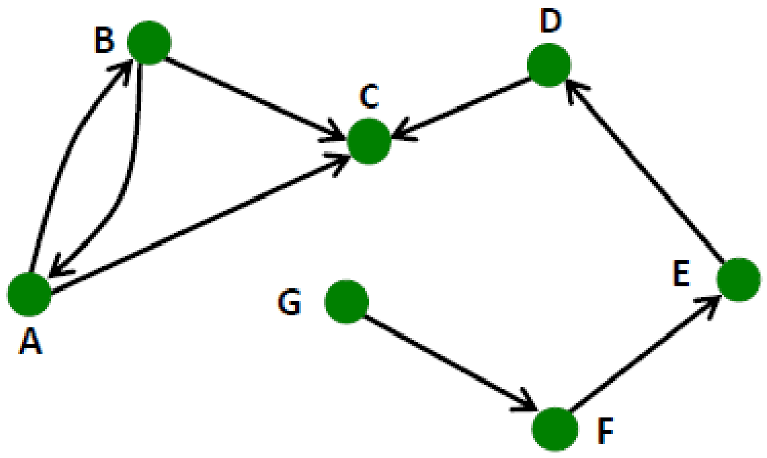
Undirected graphs

- **Links:** undirected (symmetrical, reciprocal)
- **Examples:**
  - Collaborations
  - Friendship on Facebook
- **Node degree, $\boldsymbol{k_i}$:** the number of edges adjacent to node $i.\ e.\ g.,\ k_i$ ***=4***
- Avg. degree: $\bar{k} = \frac{1}{N}\sum_{i=1}^{N} k_i$



Directed graphs:

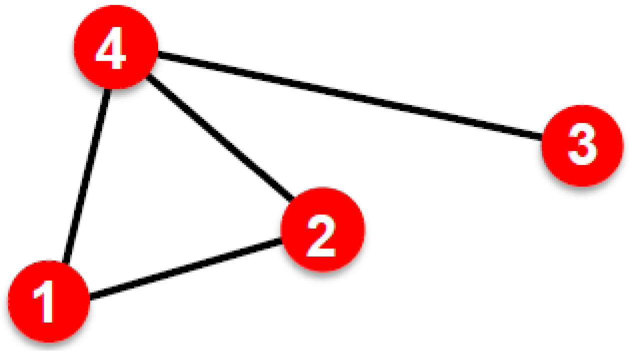
- **Links:** directed (arcs)
- **Examples:**
  - Phone calls
  - Followings on Twitter
- In directed networks we define an in-degree and out-degree.
- The (total) degree of a node is the sum of in- and out-degrees.

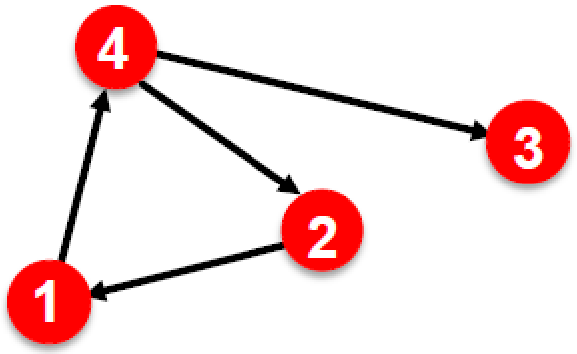


Representing Graphs: Adjacency Matrix

- Undirected Graph

Directed graph





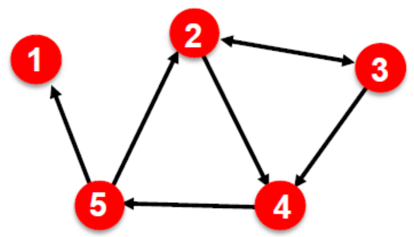
- $\boldsymbol{A_{ij}} = \mathbf{1}$ if there is a link from node $\boldsymbol{i}$ to node $\boldsymbol{j}$
- $\boldsymbol{A_{ij}} = \mathbf{0}$ otherwise

- $A_{undir} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$ $\qquad A_{dir} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$

- The adjacency matrix for an undirected graph A_{undir} is a symmetric matrix.
- The adjacency matrix for a directed graph A_{dir} is not symmetric.
- Coefficients in the adjacency matrix do not have to be 0-s and 1-s. They could represent weights and be any rational numbers.
- Adjacency matrixes are sparse. Usually, they have very few 1-s and many 0-s.

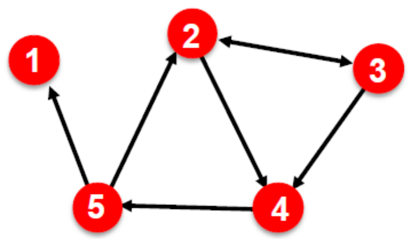
Representing Graphs: Lists of Edges and Adjacency Lists

- Represent graph as a **list of edges:**



- (2, 3)
- (2, 4)
- (3, 2)
- (3, 4)
- (4, 5)
- (5, 2)
- (5, 1)

- Represent graph as an **adjacency list**:
- Easier to work with if network is Large and Sparse
- Adjacency list allows us to quickly retrieve all neighbors of a given node.



- 1:
- 2: 3, 4
- 3: 2, 4
- 4: 5
- 5: 1, 2

Node and Edge Attributes, Types of Graphs

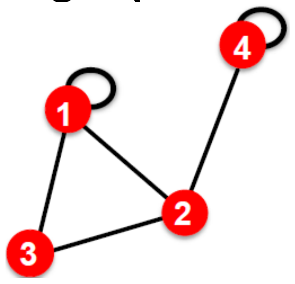
Possible options for node and edge attributes:

- Weight (e.g. , frequency of communication)
- Ranking (best friend, second best friend…)
- Type (friend, relative, co worker)
- Sign: Friend vs. Foe, Trust vs. Distrust
- Properties depending on the structure of the rest of the graph: Number of common friends.

Graphs could be **connected** : Any two vertices can be joined by a path

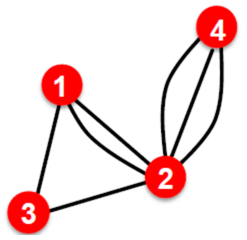
A **disconnected graph** is made up by two or more connected components

Self-edges (with self-loops, undirected)



$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \; A_{ii} \neq 0, A_{ij} = A_{ji}$$

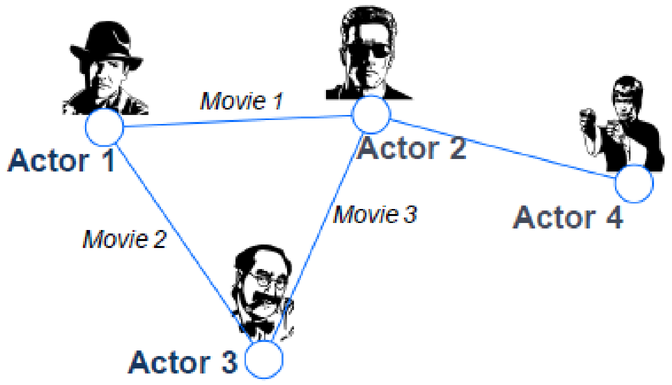
Multigraph (undirected)

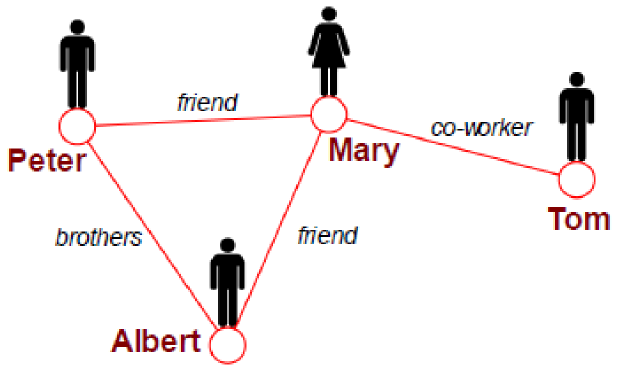


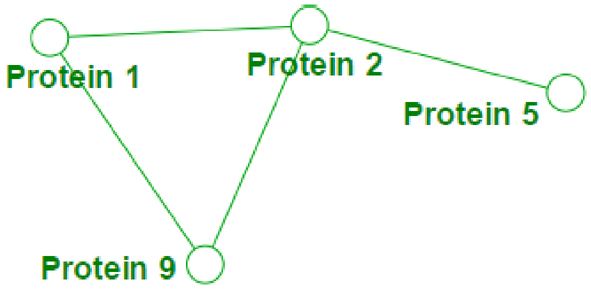
$$A = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 3 \\ 1 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}, \; A_{ii} = 0, A_{ij} = A_{ji}$$

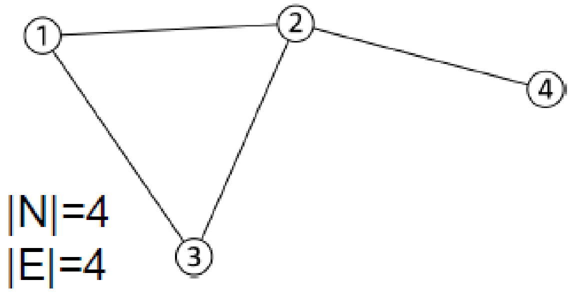
Component of a Graph, Graphs are Abstractions

- **Objects**: nodes, vertices N or V
- **Interactions**: links, edges E
- **System:** network, graph G(N,E) or G(V,E)
- Graphs are abstraction. The same graph could be used for a variety of applications
- Graphs are a common language for many domains. All problems depicted below can be represented by the same graph with 4 vertices and 4 edges.



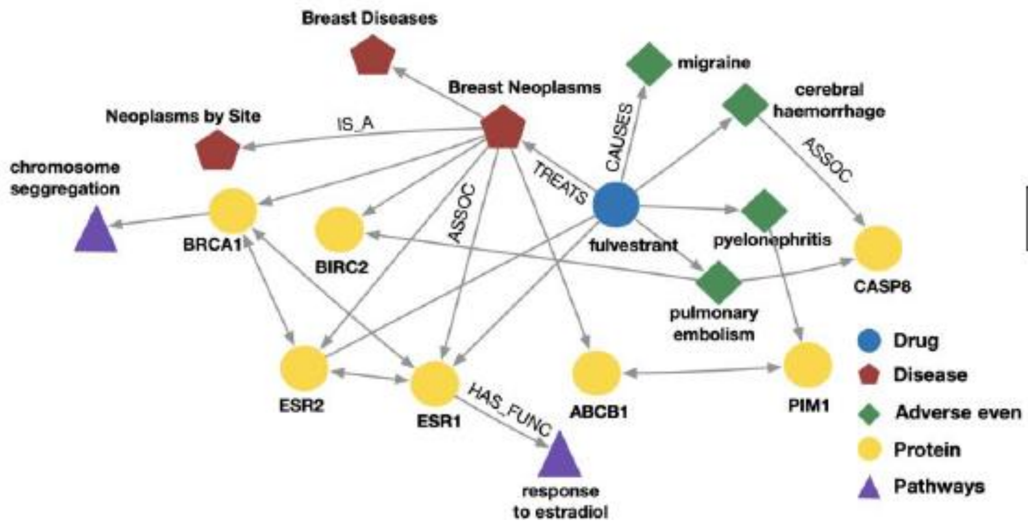


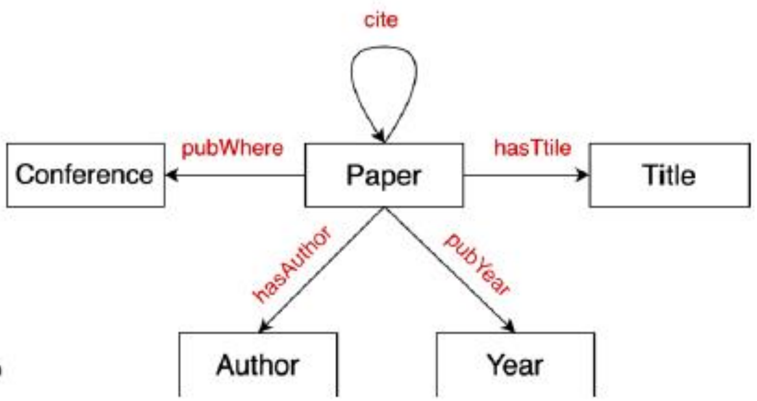




Many Graphs are Heterogeneous Graphs

- A heterogeneous graph is defined as: $G = (V, E, R, T)$
  - Vertices (nodes) with node types $v_i \in V$
  - Edges with relation types (v_i, r, v_j)
  - Node type $T(v_i)$
  - Relation type $r \in R$





Biomedical Knowledge Graphs
Example node: Migraine
Example edge: (fulvestrant, Treats, Breast Neoplasms)
Example node type: Protein
Example edge type (relation): Causes

Academic publications graph
Example node: ICML
Example edge: GraphCN, NeurIPS
Example node: Author
Example edge type (relation) pubYear

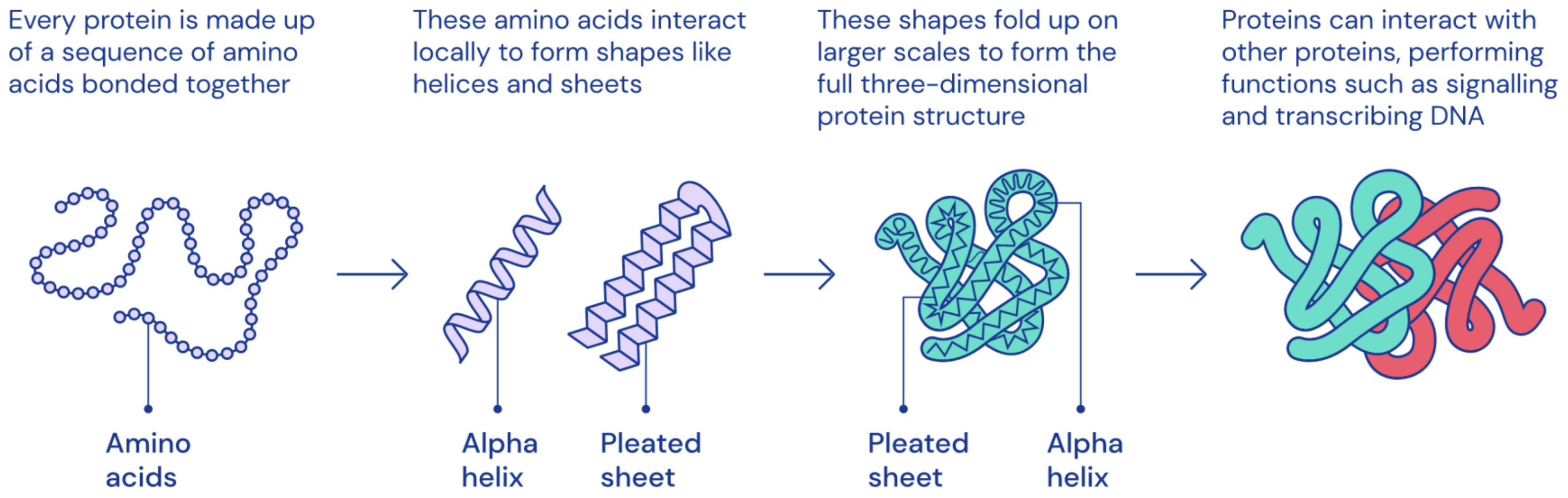
Examples of Uses of GNNs

Graph Neural Networks

- A graph neural network (GNN) is an algorithm that allows us to learn from graphs, including predicting or classifying properties of nodes, edges and graph features.
- Graph neural networks are similar to conventional neural networks, except the learning takes place on graph data.
- Specialized neural network architecture for graphs was first introduced in the late 1990s, gained traction in the early 2000s, and greatly accelerated after 2015.
- Today, there are many examples of enterprise systems (Google, Alibaba, others) that employ GNNs at scale.
- GNNs are profitably used in speeding up drug discovery.
- GNNs is still a rapidly developing field, with many remaining challenges

Example: Protein Folding

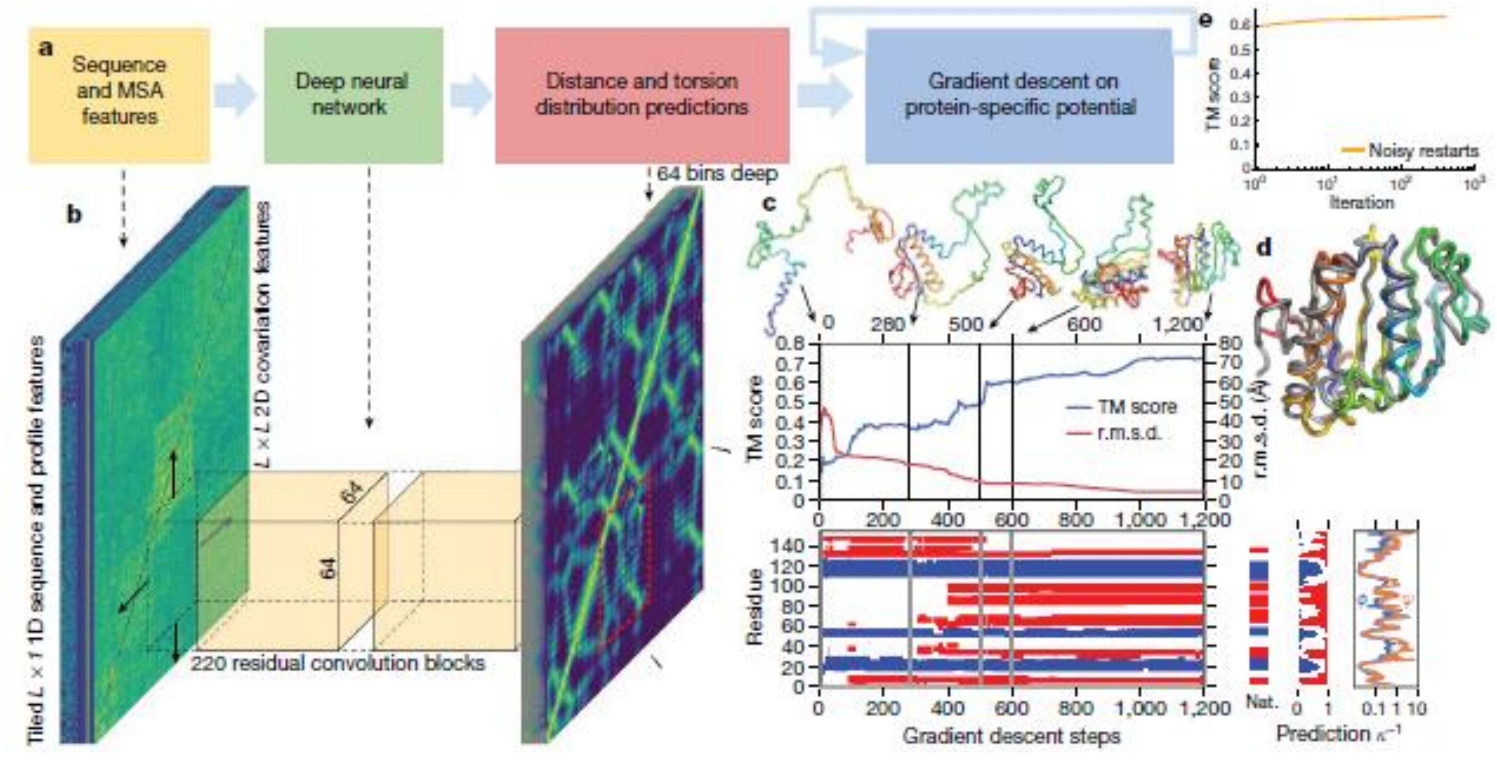
- A protein chain in solution folds into a 3D structure. Knowledge of that structure is essential for understanding how a protein interacts with other proteins.
- One successful application of GNNs is predicting such 3D structures.


Every protein is made up of a sequence of amino acids bonded together
These amino acids interact locally to form shapes like helices and sheets
These shapes fold up on larger scales to form the full three-dimensional protein structure
Proteins can interact with other proteins, performing functions such as signalling and transcribing DNA
Amino acids
Alpha helix
Pleated sheet
Pleated sheet
Alpha helix

- [AlphaFold: Using AI for scientific discovery | DeepMind]

AlphaFold

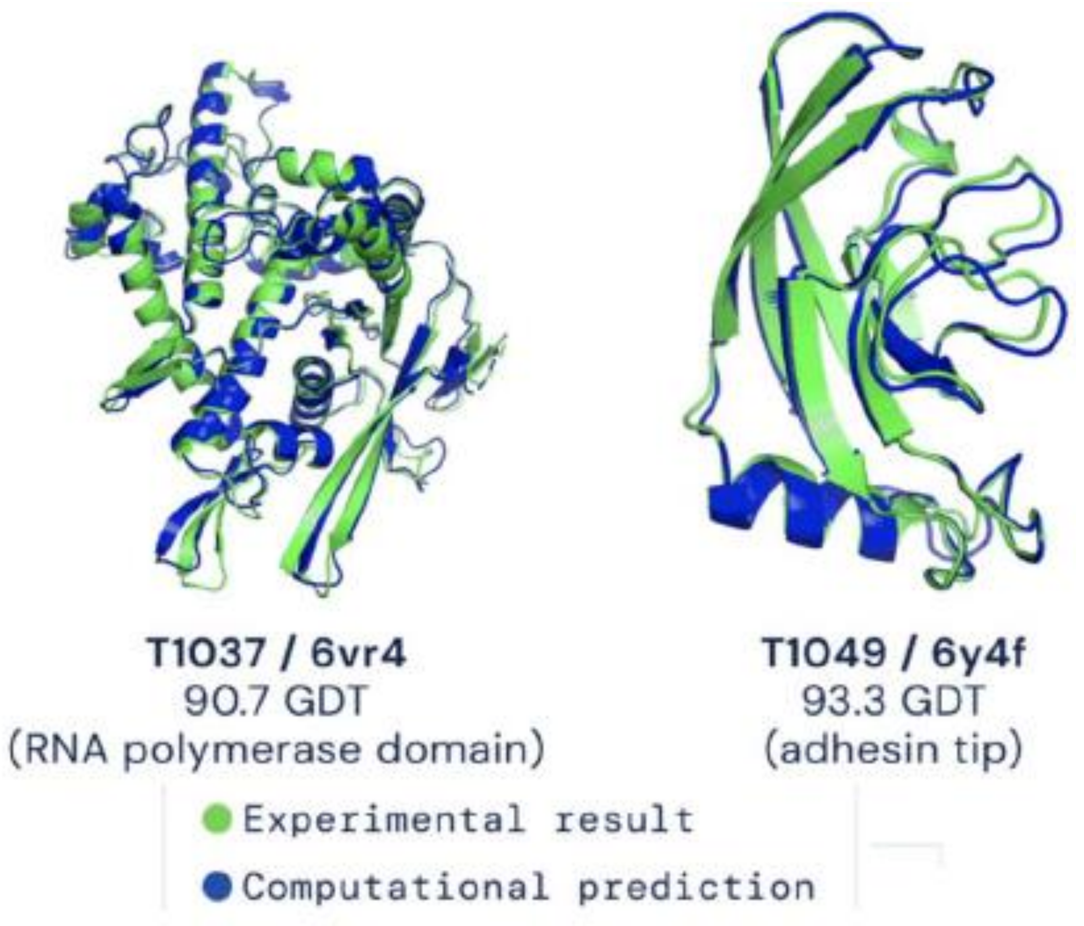
- AlphaFold is a project of DeepMind a Google company. DeepMind developed GNNs which predict the structure (folding) of proteins. https://www.nature.com/articles/s41586-019-1923-7.epdf


a
Sequence and MSA features
Deep neural network
Distance and torsion distribution predictions
Gradient descent on protein-specific potential
b
Tiled L × 1 1D sequence and profile features
L × L 2D covariation features
64
64
220 residual convolution blocks
64 bins deep
i
j
c
0
280
500
600
1,200
TM score
r.m.s.d.
r.m.s.d. (Å)
Residue
Gradient descent steps
Nat.
Prediction κ⁻¹
d
e
TM score
Noisy restarts
Iteration

Protein Folding

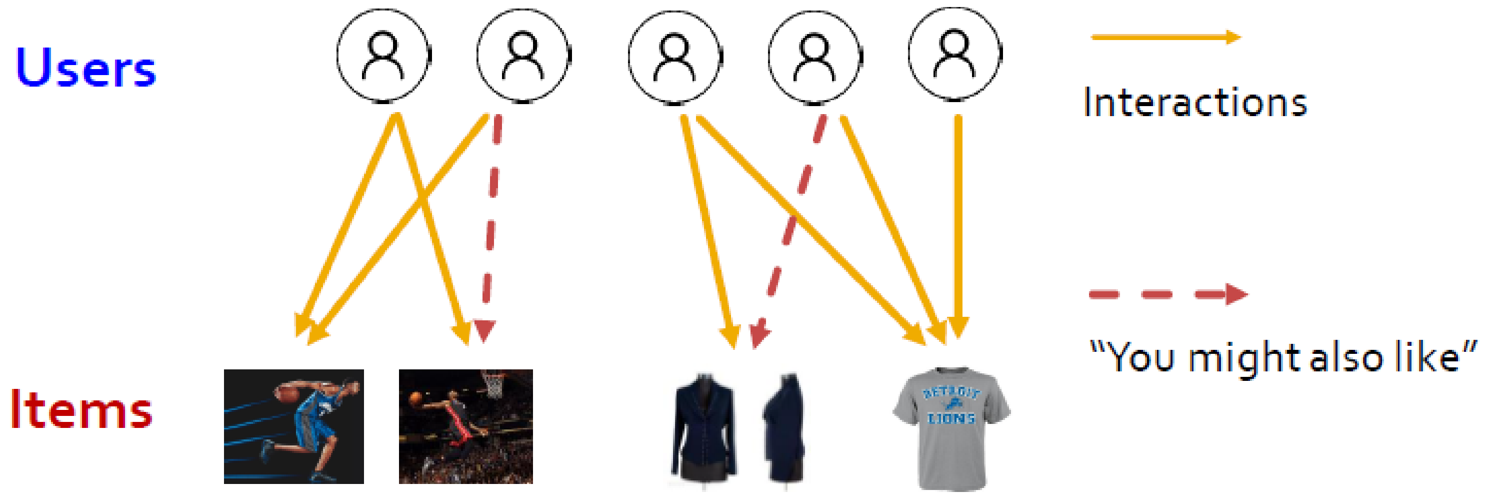
- GNNs could computationally predict proteins' 3D structure based solely on protein's amino acid sequence.

- **Key idea:** "Spatial graph"
- **Nodes:** Amino acids in a protein sequence
- **Edges:** Proximity between amino acids (residues)



Recommender Systems

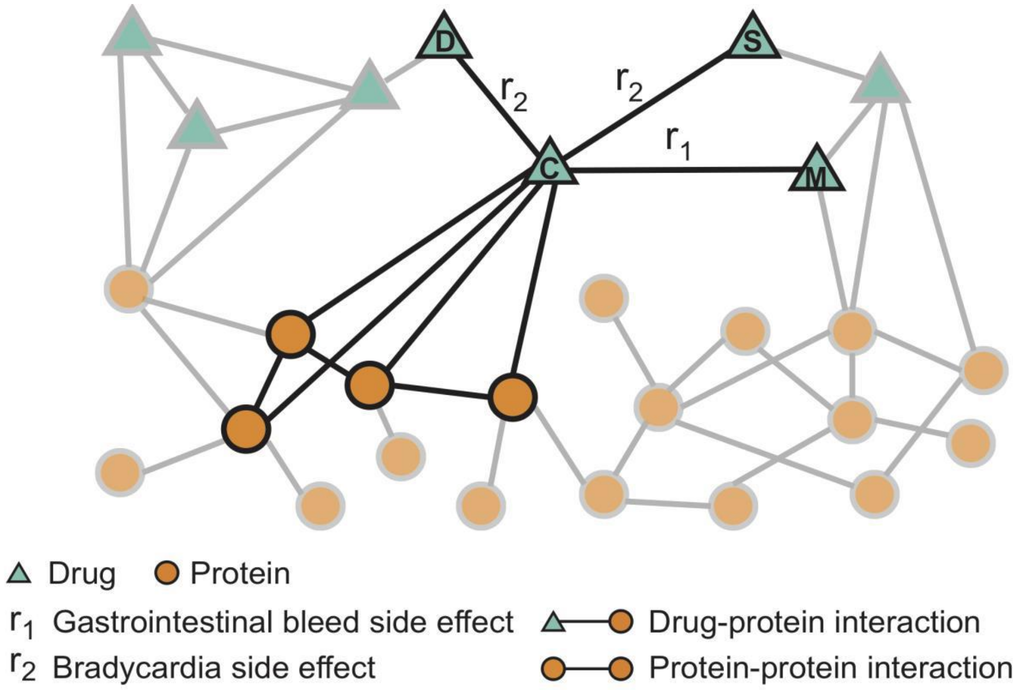
- We can build recommender systems using GNNs.
- Users interacts with items
  - Watch movies, buy merchandise, listen to music
  - **Nodes**: Users and items
  - **Edges:** User item interactions
- Goal: Recommend items users might like



Drug Side Effects

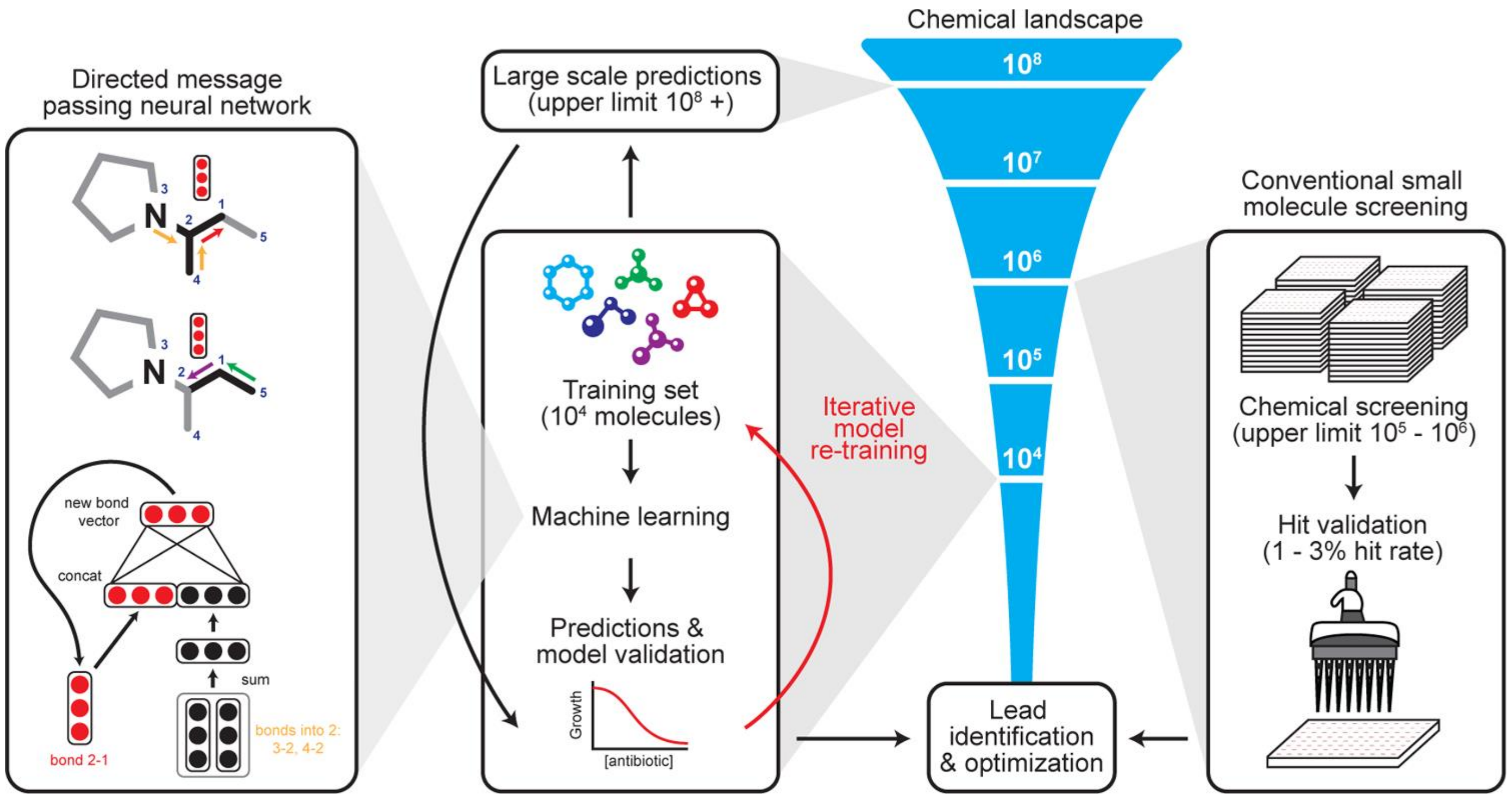
- Many patients take multiple drugs to treat complex or co-existing diseases:
  - 46% of people ages 70-79 take more than 5 drugs
  - Many patients take more than 20 drugs to treat heart disease, depression, insomnia, etc.
- Given a pair of drugs, GNNs can predict adverse side effects
- [Modeling Polypharmacy Side Effects with Graph Convolutional Networks (arxiv.org)]
- **Nodes**: Drugs & Proteins
- **Edges**: Interactions

Query: How likely will Simvastatin and Ciprofloxacin, when taken together, break down muscle tissue?


D
S
C
M
r_2
r_2
r_1
Drug
Protein
r_1 Gastrointestinal bleed side effect
r_2 Bradycardia side effect
Drug-protein interaction
Protein-protein interaction

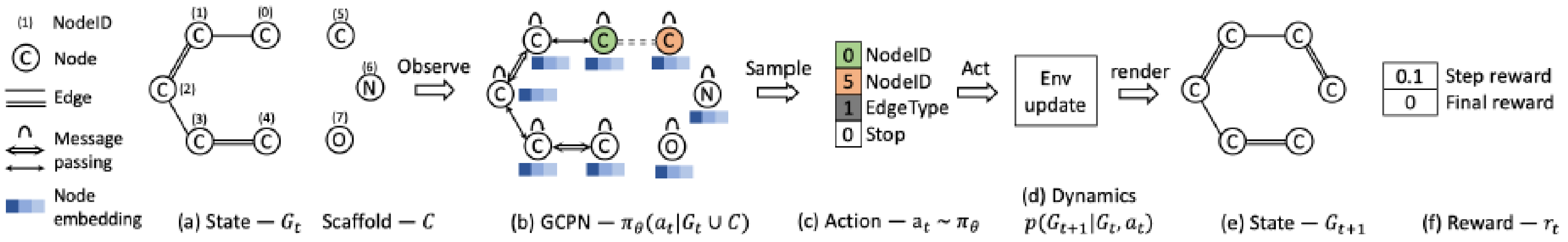
Drug Discovery

- Antibiotics are small molecular graphs
  - **Nodes**: Atoms
  - **Edges:** Chemical bonds
- A classification model based on Graph Neural Network **can** predict promising molecules from a large pool of candidates (https://pubmed.ncbi.nlm.nih.gov/32084340/)

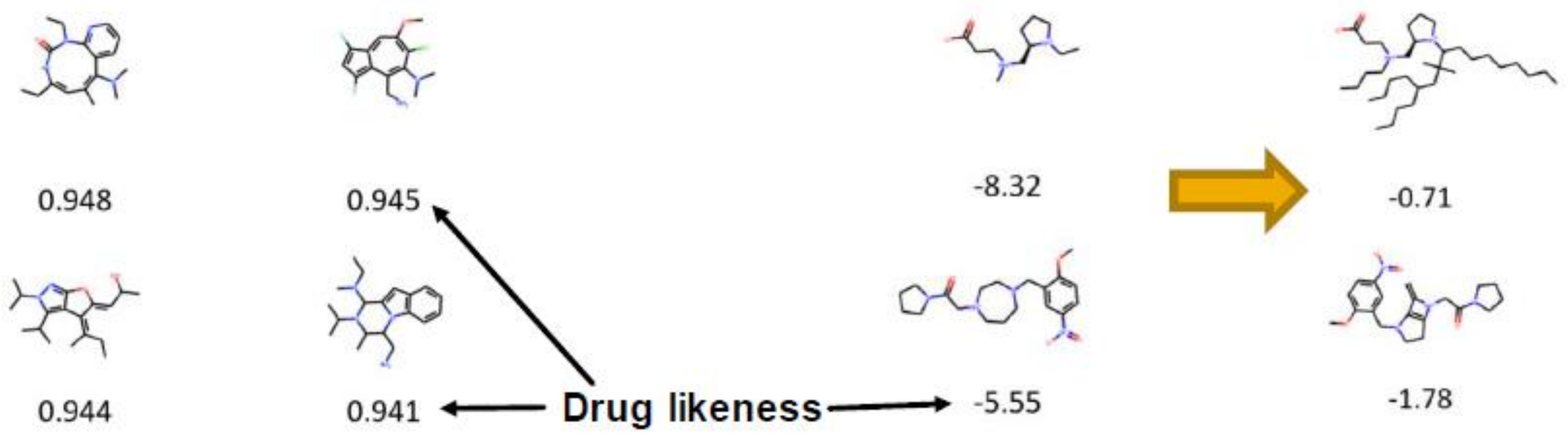

Directed message passing neural network
N
1
2
3
4
5
new bond vector
concat
sum
bond 2-1
bonds into 2: 3-2, 4-2
Large scale predictions (upper limit 10^8 +)
Training set (10^4 molecules)
Machine learning
Predictions & model validation
Growth
[antibiotic]
Iterative model re-training
Chemical landscape
10^8
10^7
10^6
10^5
10^4
Lead identification & optimization
Conventional small molecule screening
Chemical screening (upper limit 10^5 - 10^6)
Hit validation (1 - 3% hit rate)

Molecule Generation - Optimization

- **GNNs can generate novel molecules**

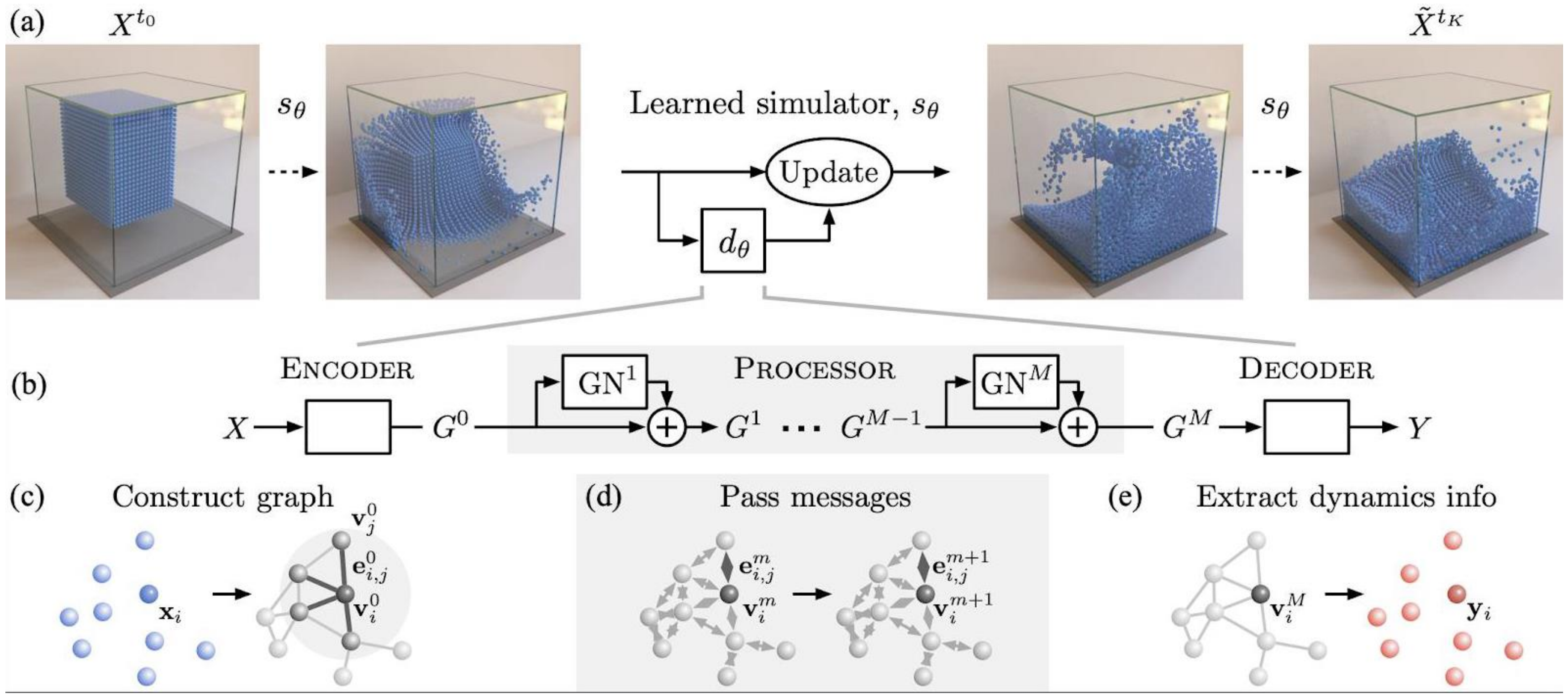


- **Use case 1**: Generate novel molecules with high drug likeness value
- **Use case 2:** Optimize existing molecules to have desirable properties



Physical Simulations

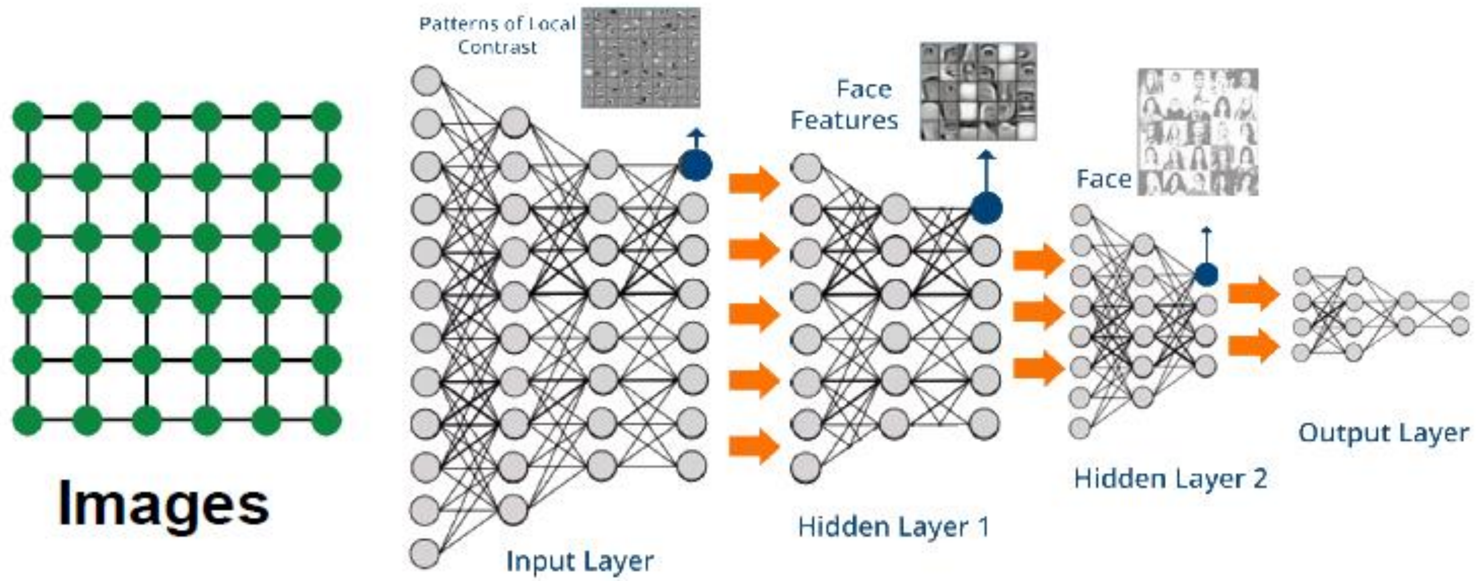
- We could treat a physical structure as a dynamic graph
  - **Nodes:** particles
  - **Edges:** interactions between particles
  - **Predict:** how graph evolves over time

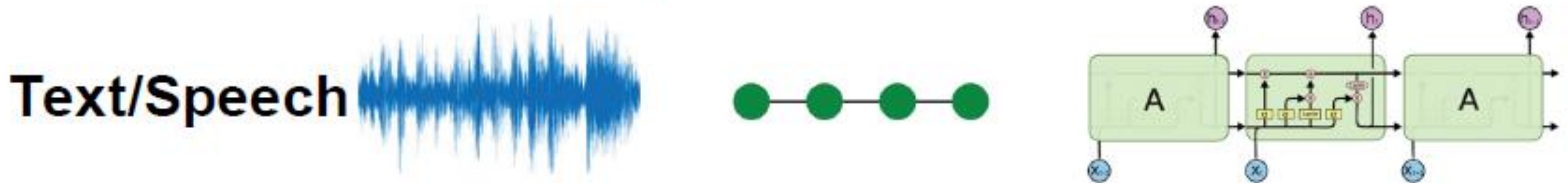


Graph Analysis & Embeddings

Legacy Neural Networks

- “Classical” deep learning toolbox is designed for simple sequences & grids





Graph Analysis

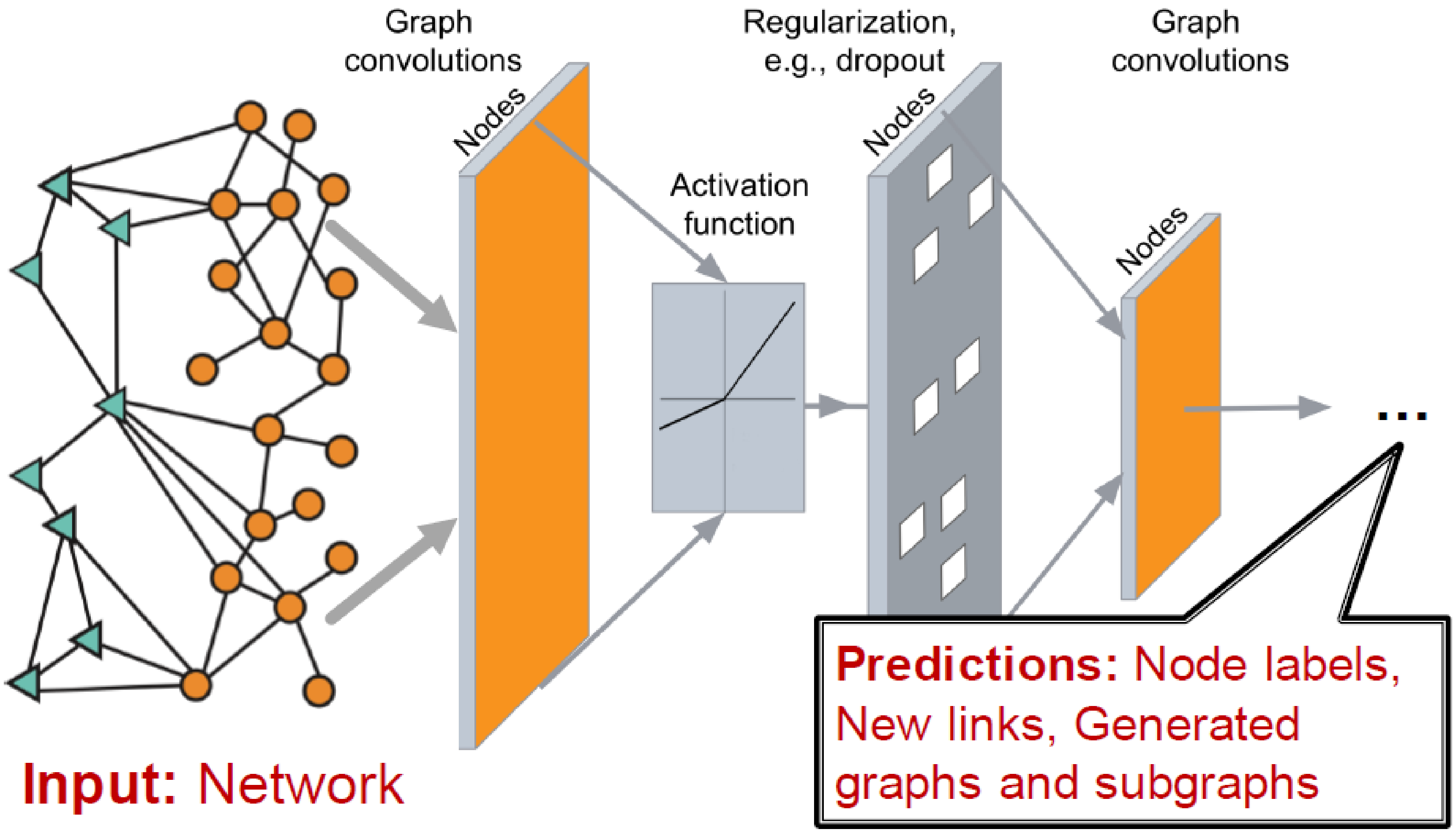
- A **Graph Analysis** (e.g., Graph Neural Network's forward pass) can be thought of as two processes:
  a) converting input graphs into useful embeddings, and/or
  b) performing some downstream task (e.g., classification) on the embeddings, which converts the embeddings into some useful output.
- Typically, in Graph Analysis we produce three output types:

  (1) **Vertex-level outputs,** when we require a prediction (e.g., a distinct class or predicted property value) for each vertex in a graph.

  (2) **Edge-level outputs,** when we make a prediction for one or many properties of edges in a graph.

  (3) **Graph-level outputs,** when we require a prediction for a property or feature of an entire graph. For example, solubility in a certain liquid is a property of the entire molecule (graph).

Forward Transformation of Graphs through GNNs

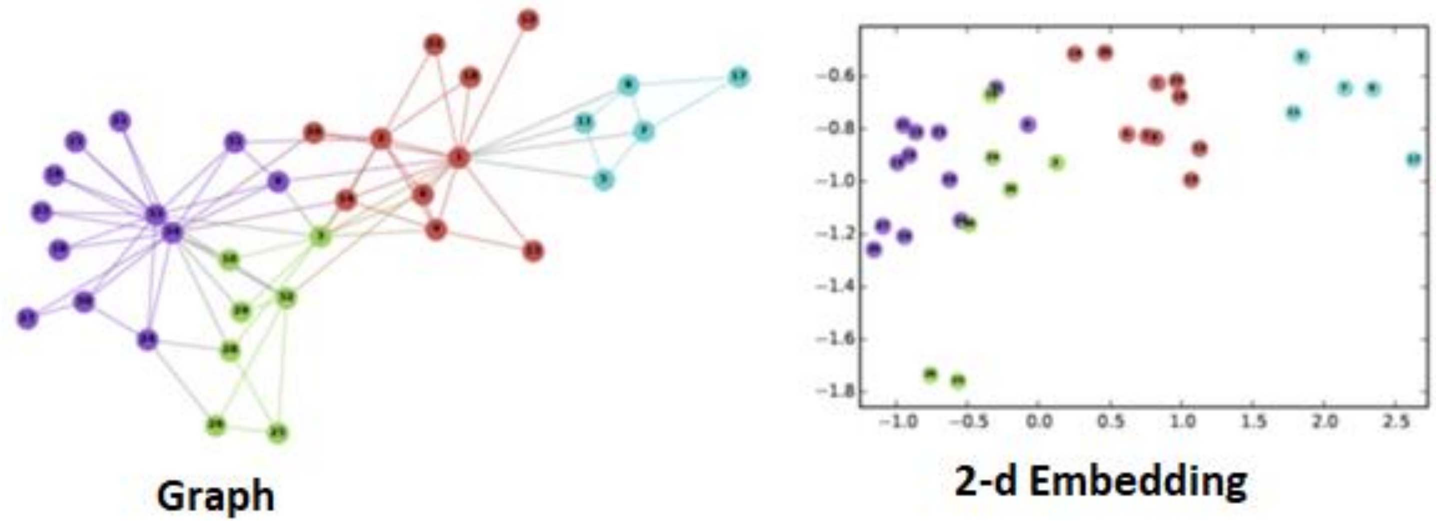

Graph convolutions
Regularization, e.g., dropout
Graph convolutions
Nodes
Activation function
Nodes
Nodes
...
Predictions: Node labels, New links, Generated graphs and subgraphs
Input: Network

Embeddings

- **Embeddings** are compressed feature representations. By reducing large feature vectors associated with vertices and edges into low dimensional embeddings, we make it possible to classify vertices and edges with lower-order models.
- A key measure of an embedding's quality is whether the (cosine or some other) similarity between nodes or edges in the original space (graph) is retained in the embedding space.
- Embeddings can be created (or learned) for vertices, edges, neighborhoods, or the entire graphs.
- Embeddings are also referred to as *representations*, encodings, *latent vectors*, or high-level *feature vectors* depending on the context.

Graph Representation

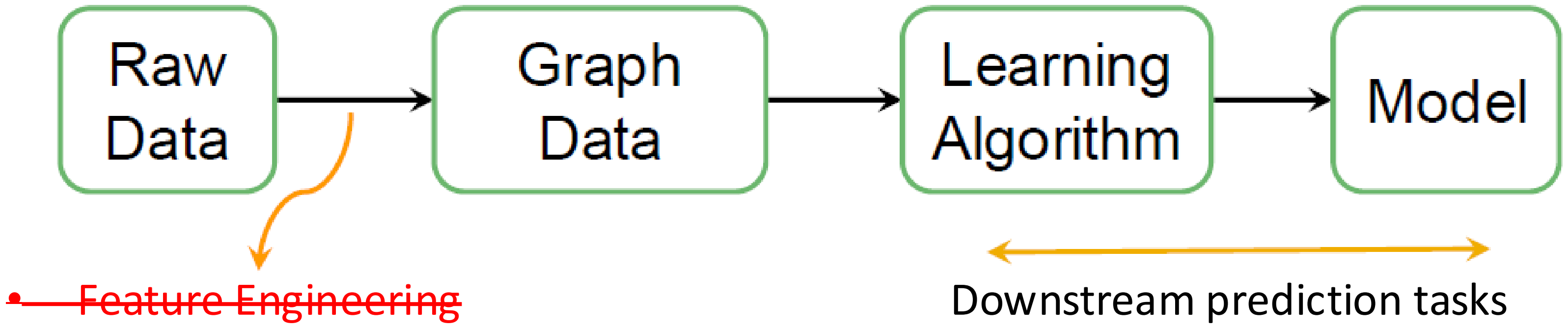
- The challenge of dealing with non-Euclidian graphs is to represent them accurately in a way that can be fed as an input into a "conventional" neural network layers.
- In GNNs, the starting layers produce **embeddings which** transform a graph structure into a vector or matrix representation.



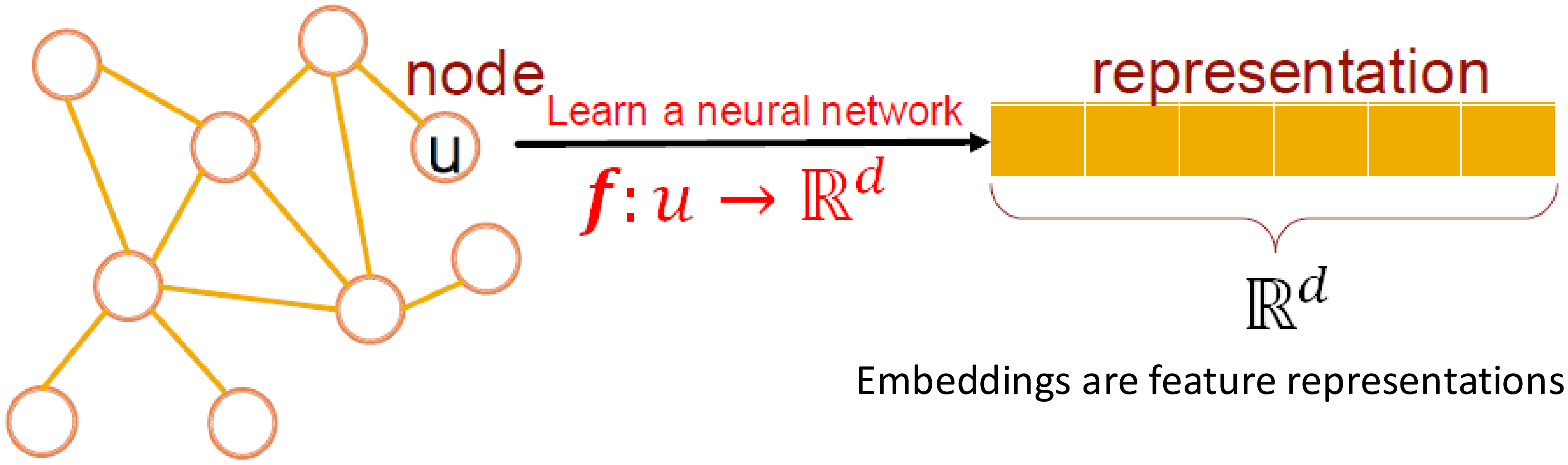
- Above is an illustration of graph embeddings. We start (left) with a non-Euclidean graph structure and end up with embedding vectors (right) that can be projected into a 2D Euclidean space.

Machine Learning Lifecycle: Representation Learning

- Representation Learning – Automatically learn the features (embeddings)



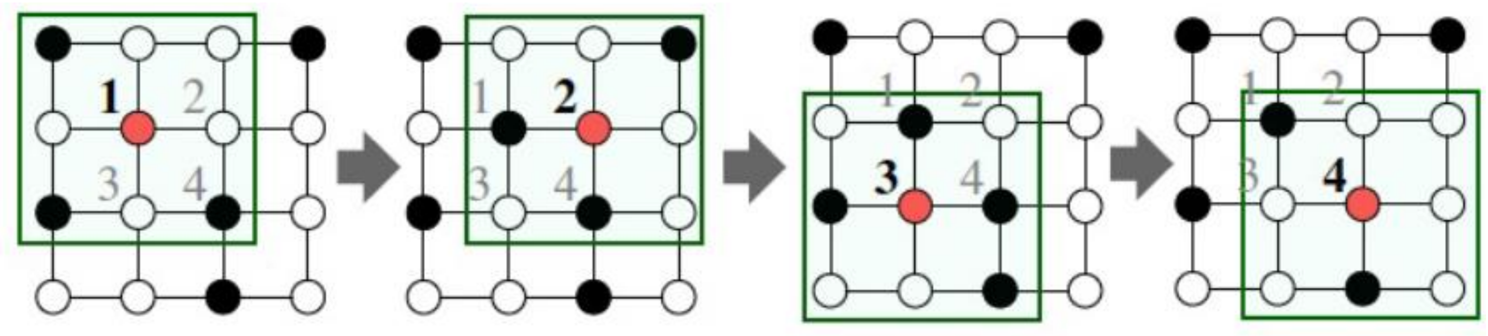
- Initial layers of GNNs map nodes to d-dimensional vectors (embeddings) such that similar nodes in the network are embedded close together.

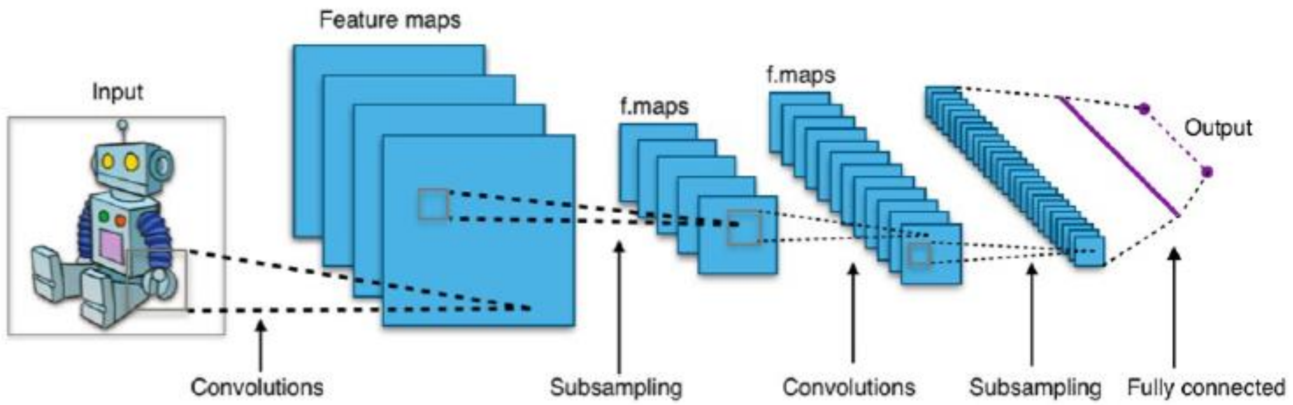


Embeddings are feature representations

Implementing a NN on a Graph

- We understand CNNs well. So, our goal is to generalize convolutions beyond simple lattices.
- We want to use, leverage node features/attributes (e.g., text, images).
- CNNs on images relied on the input organized as a regular grid and filters (flashlights, sliding windows) we move over those images.

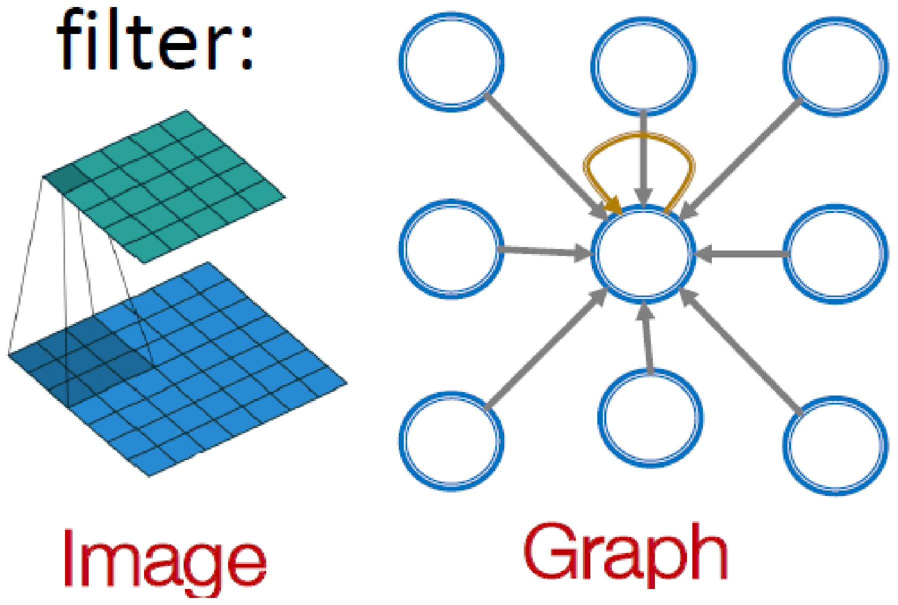




- The question is, could we do something similar on a graph?

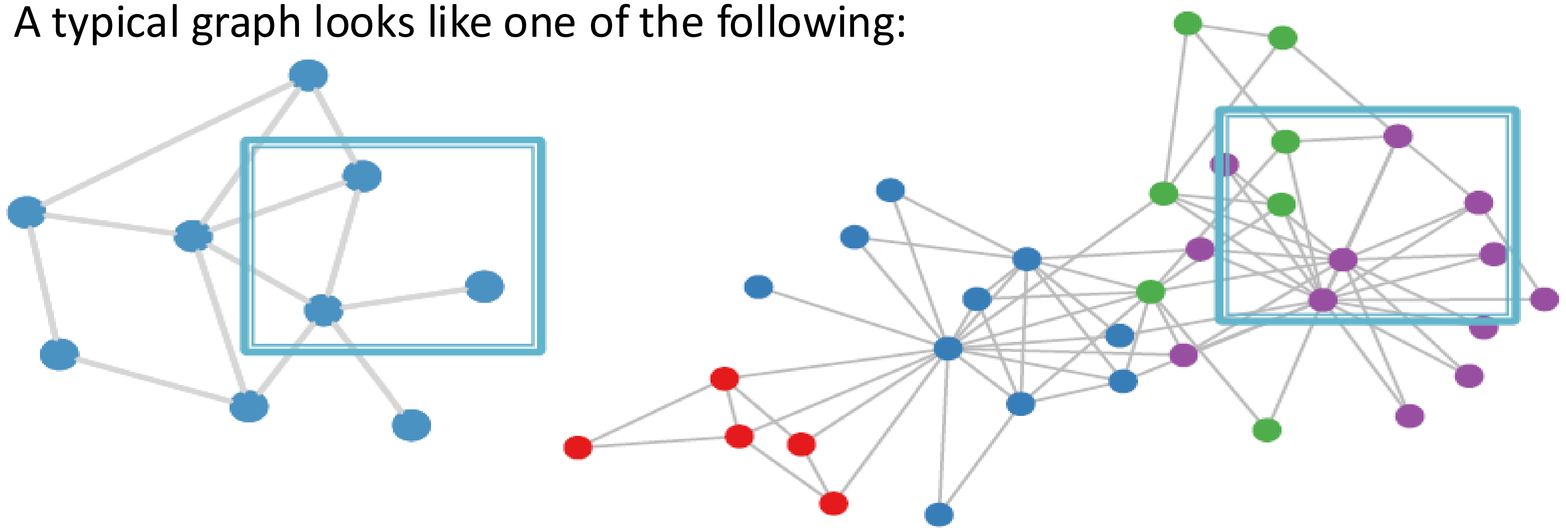
CNN vs. GNN

- CNN can be seen as a special GNN with fixed neighbor size and ordering:
  - In CNNs, the size of the filter is pre-defined.
  - GNN process arbitrary graphs with different neighborhoods for each node.



A typical graph is Permutation Invariant

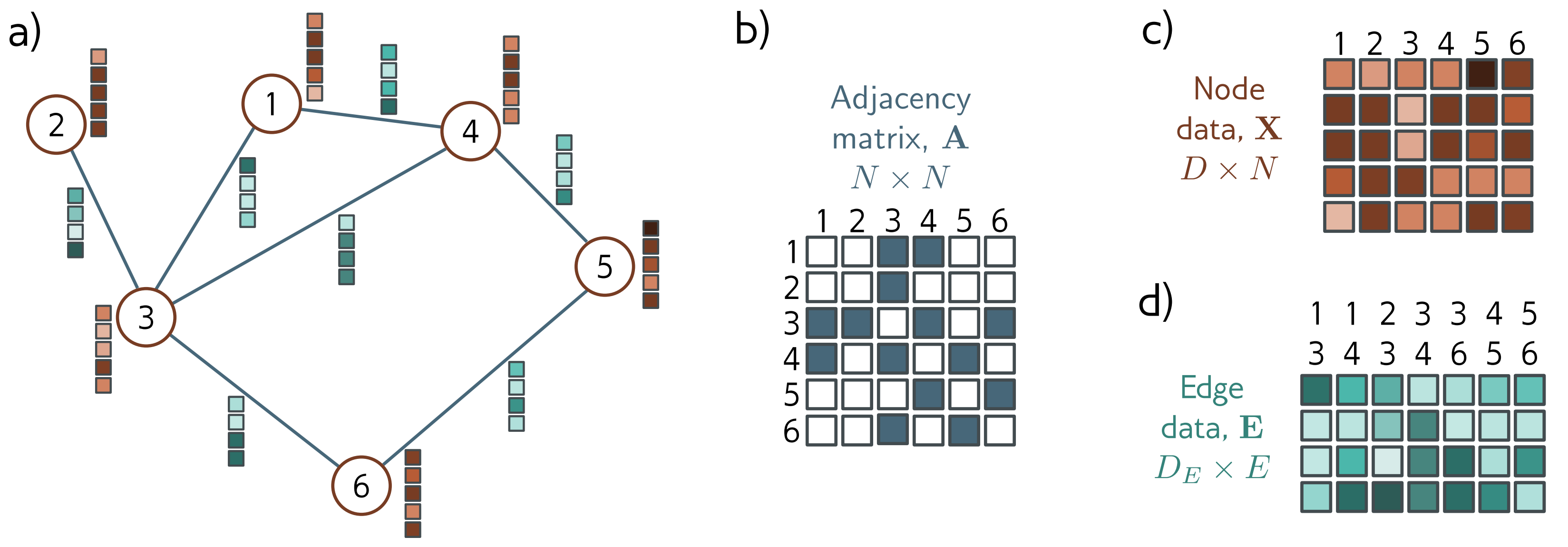
- A typical graph looks like one of the following:



- How do you systematically move a “window” over the graph.
- There is no fixed notion of locality or simple way to define a sliding window on the graph
- Graph is permutation invariant. Relabeling the nodes should not change anything. We want any schema we define to keep on working under arbitrary permutation of node labels.
- Permutation invariance means that a **graph does not have a canonical order of the nodes!**
- We can have many different order plans. Any labeling of nodes should be equivalent to any other. We cannot take the adjacency matrix too seriously. Relabeling will result in very different adjacency matrix. Graph and node representation should be the same for any node labeling.

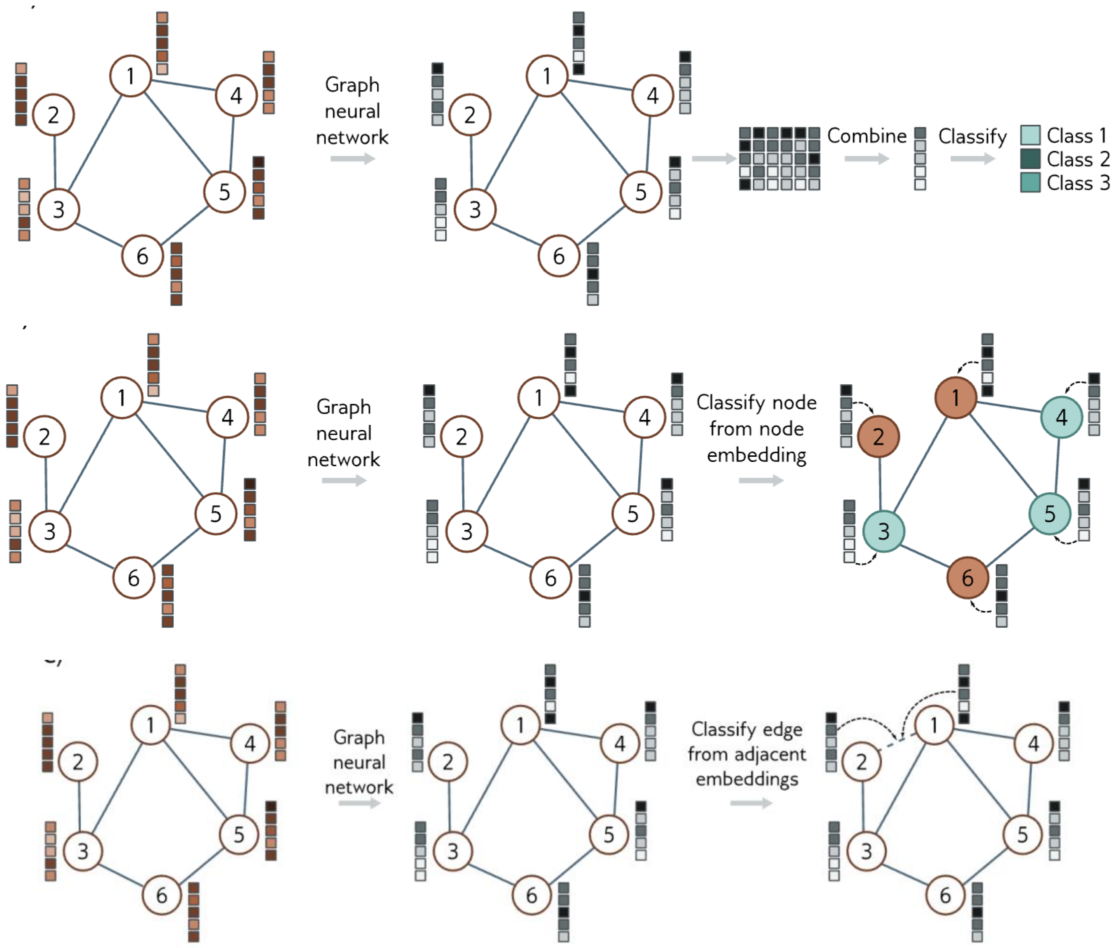
Implementing GNNs

Thinking in Graphs



a) A Graph with 6 Nodes and respective Node and Edge features
b) Adjacency Matrix
c) Node Data Matrix
d) Edge Data Matrix

Thinking in Graphs



Graph Level

Node Level

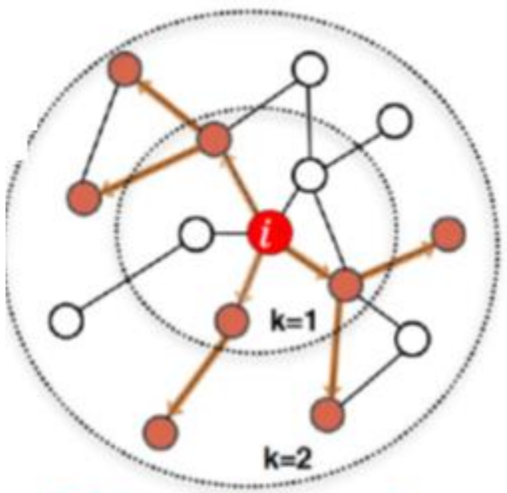
Edge Level

Central Idea of GNN

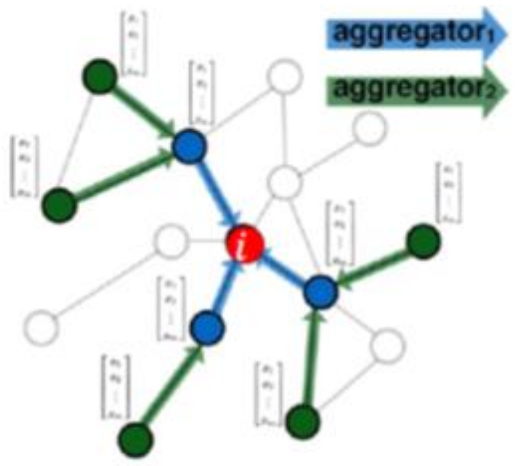
The central idea of GNN’s is the neighborhood aggregation scheme which generates node embeddings by combining information from neighborhoods !

Permutation Invariance, Graph Convolutional Networks

- We design graph neural networks that are permutation invariant by passing and aggregating information from every node's neighbors!
- **Key idea:** Node's neighborhood defines a local computation graph.



Determine node's computation graph

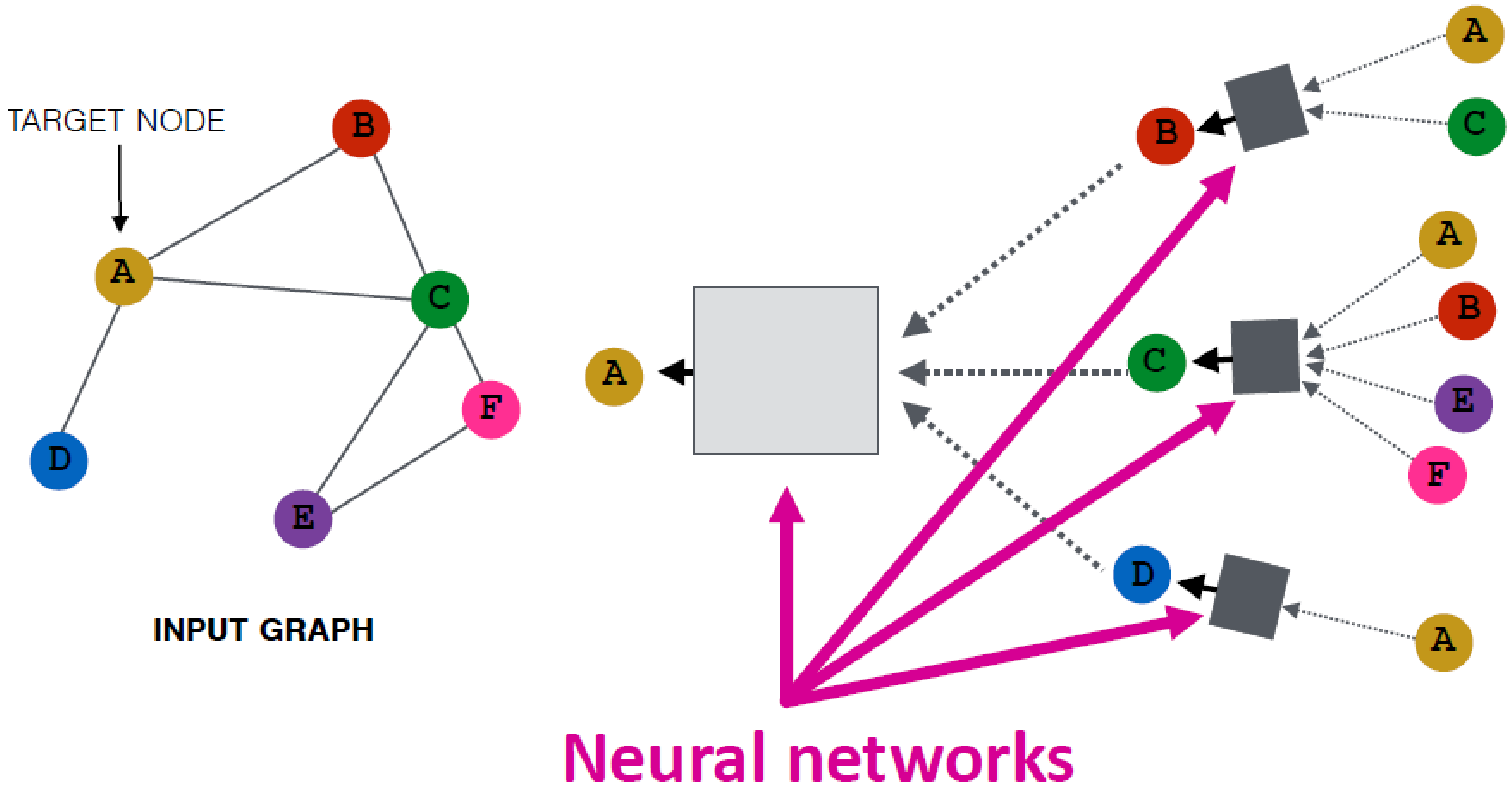


Propagate and transform information

- We propagate information across the graph and compute node features.
- We generate node representations (embeddings) based on local network neighborhoods.
- Calculations are iterative and are repeated until the features are stabilized.

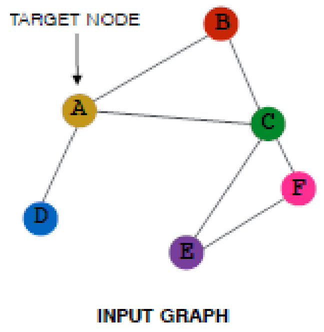
Aggregate Neighbor's Features

- **Key idea:** Generate node embeddings based on local network neighborhoods.
- **Intuition:** Nodes aggregate information from their neighbors using an iterative process we will call the neural network.



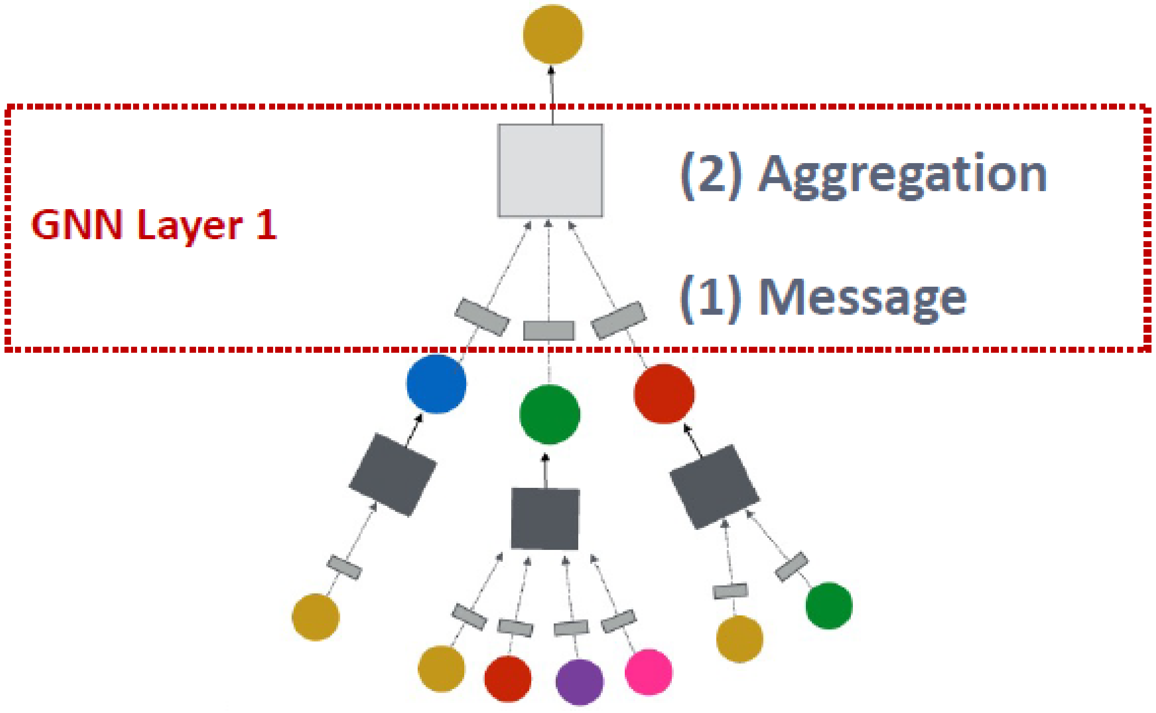
GNN Layer = Message + Aggregation

- There are two key notions in this process: messages and aggregations
- Different GNN architectures define messages and aggregations in different ways



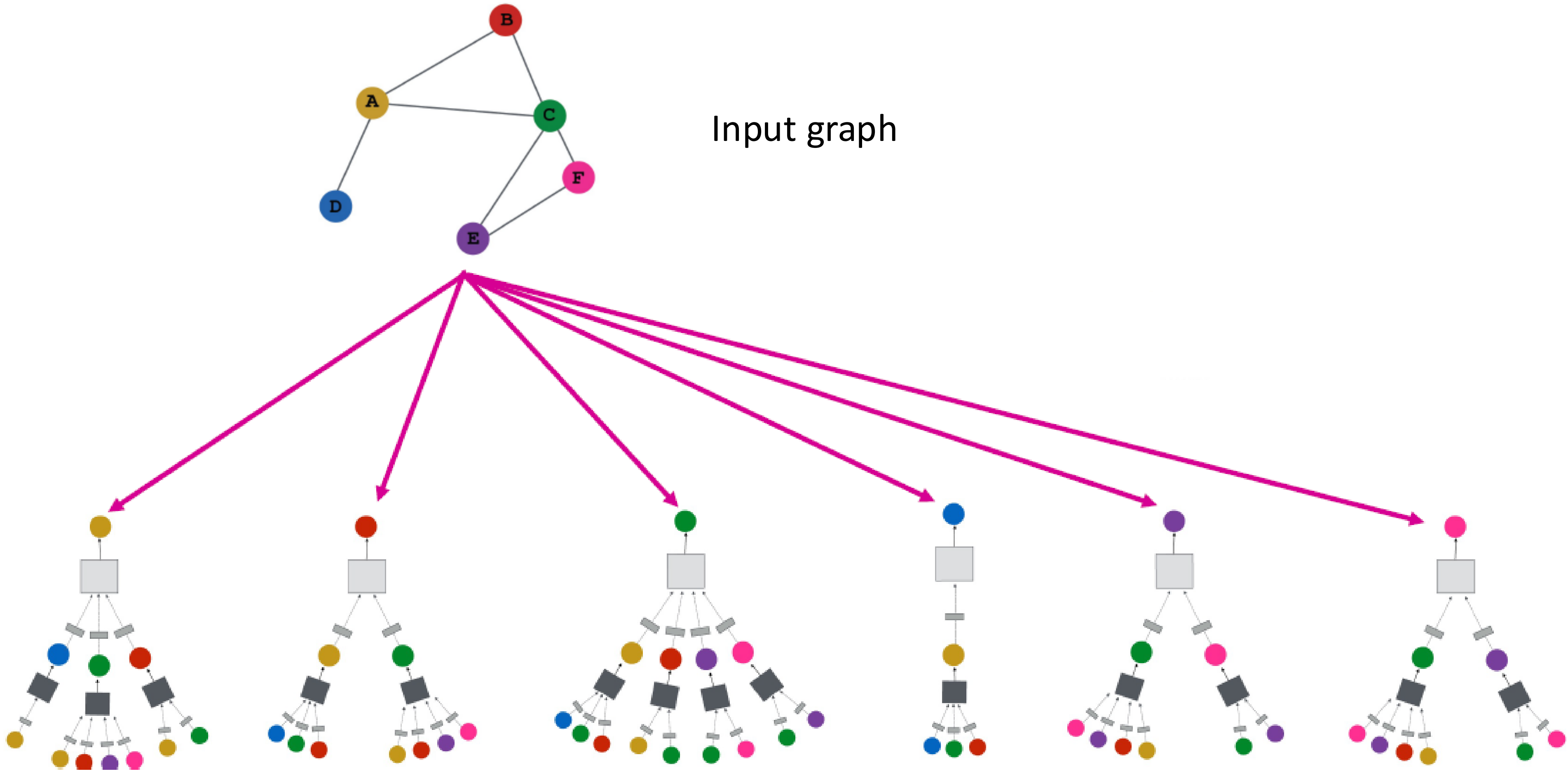
GNN Layer = Message + Aggregation

- **Different instantiations under this perspectiv**
- **GCN, GraphSAGE, GAT, …**



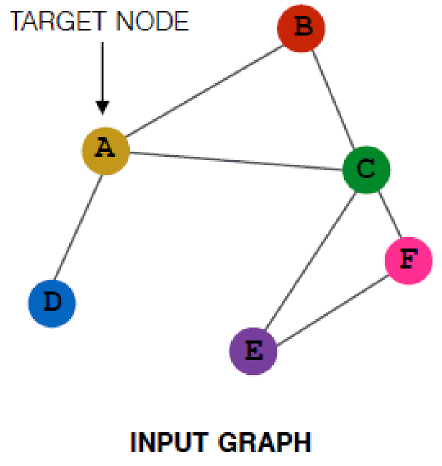
Aggregating Neighbors

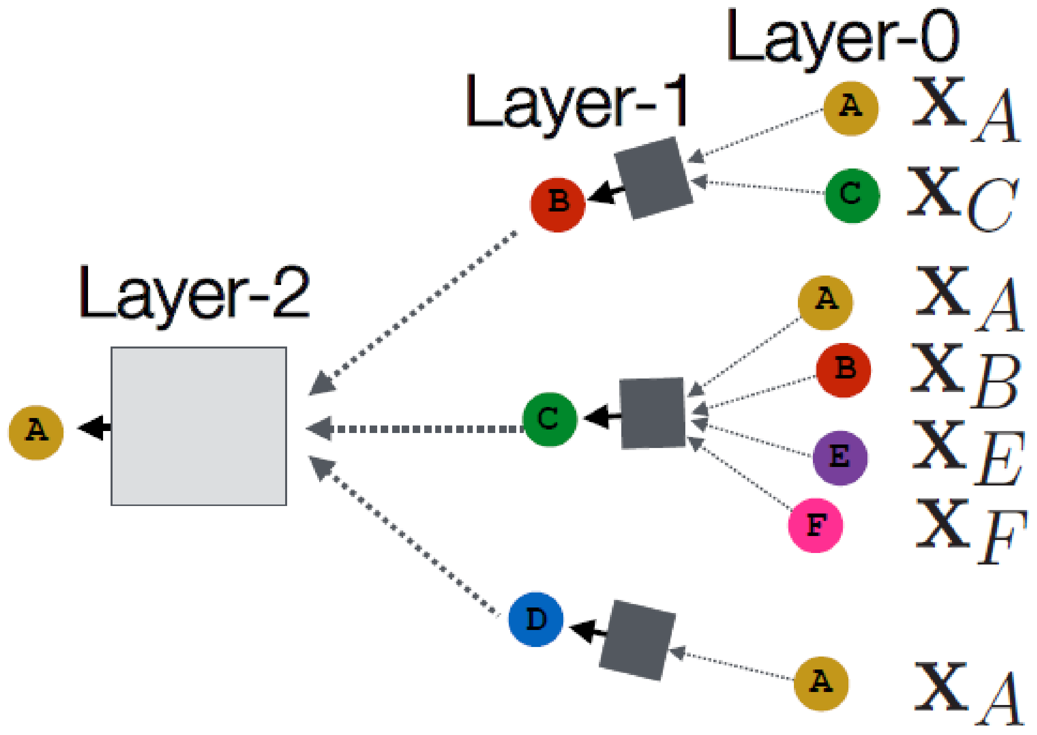
- **Intuition:** Network neighborhood defines a computation graph.
- Every node defines a (local) computational graph based on its neighborhood!
- Graph with 6 nodes will result in 6 computational graphs.



Depth of the Model: How many layers

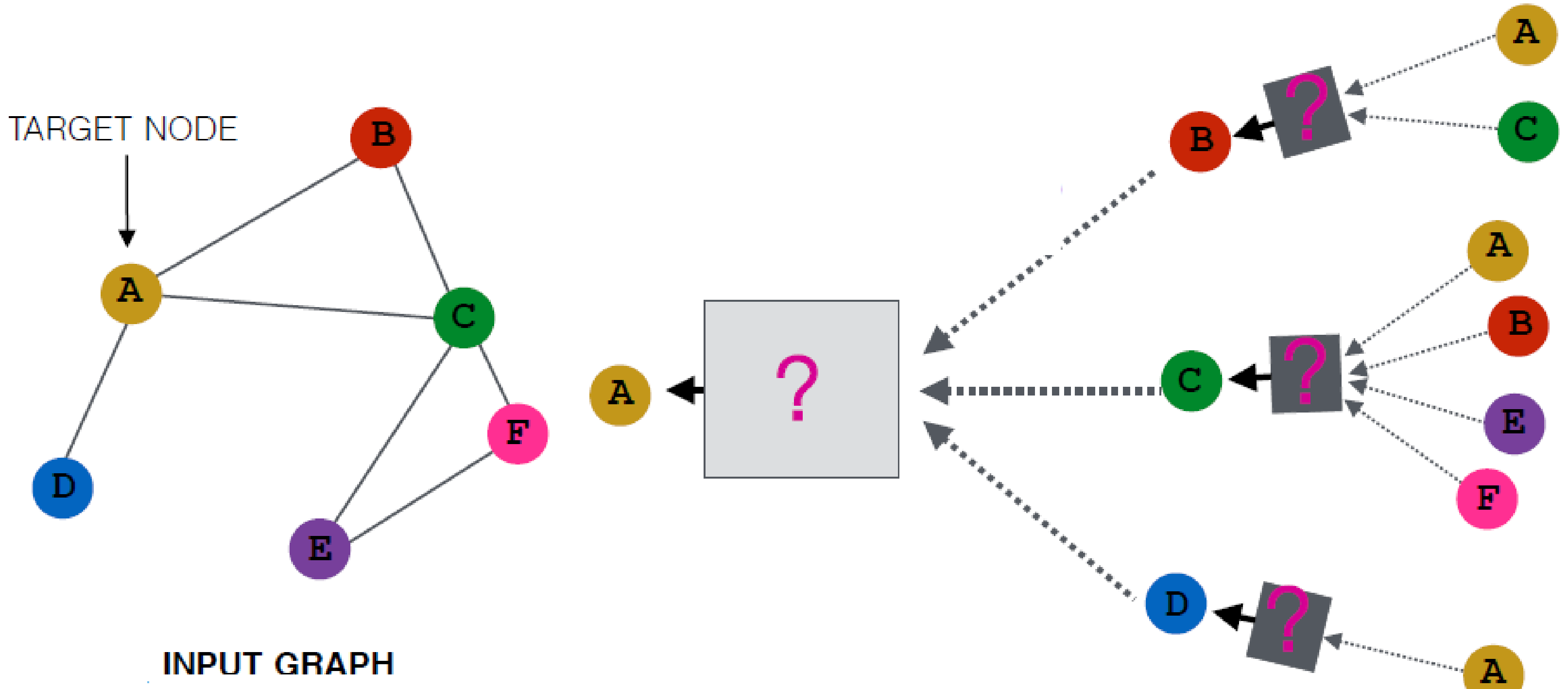
- What we proposed on the previous slide implies recursion. All nodes source in inputs from all their neighbors and so on.
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node v is its input feature x_v
  - Layer-k embedding gets information from nodes that are k hops away





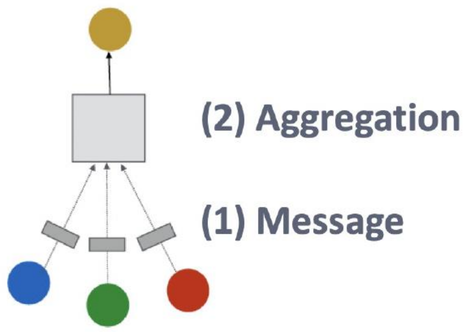
Neighborhood Aggregation

- How we perform aggregation of information (generate embeddings) from the neighborhood is somewhat arbitrary.
- Different types of GNNs perform aggregation in different way.
- **Different GNN types are define by the “content” of the box?**



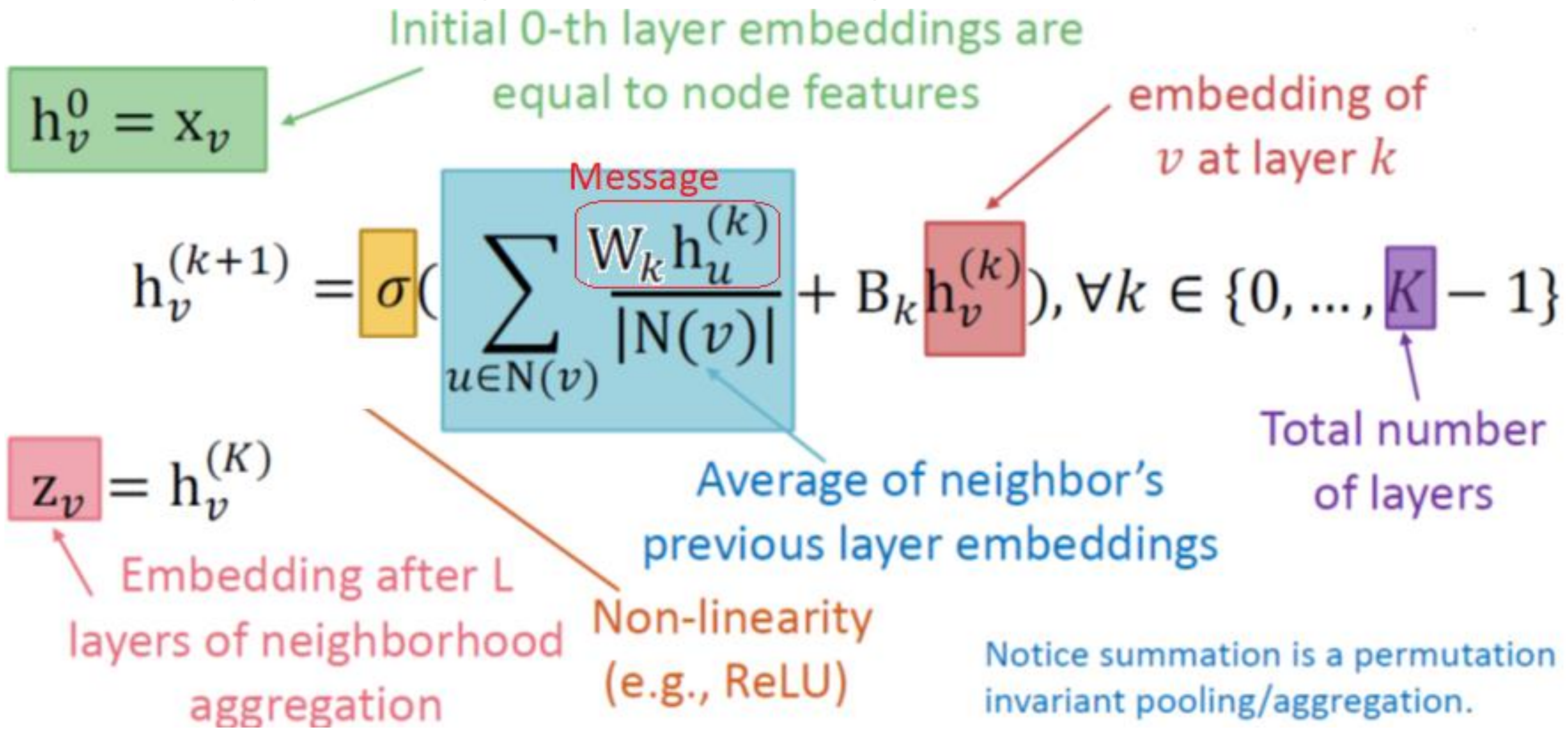
Basic Approach : Message Passing and Aggregation (Simplified)

$$\mathbf{h}_v^{(l)} = \sigma\left(\underbrace{\sum_{u \in N(v)}}_{\text{Aggregation}} \overbrace{\mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}^{\text{Message}} \right)$$



Basic approach

- Basic approach: Average information from neighbors



$$h_v^0 = x_v$$

$$h_v^{(k+1)} = \sigma\Big(\sum_{u \in N(v)} \frac{W_k h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\Big), \forall k \in \{0, \dots, K-1\}$$

$$z_v = h_v^{(K)}$$

Initial 0-th layer embeddings are equal to node features
Message
embedding of v at layer k
Total number of layers
Average of neighbor's previous layer embeddings
Embedding after L layers of neighborhood aggregation
Non-linearity (e.g., ReLU)
Notice summation is a permutation invariant pooling/aggregation.

- Message passing and neighbor aggregation in graph convolution networks is permutation equivariant. Every target node v will have the same computational graph irrespective of different labels (permutations) attached to nodes.
- W_k is trainable weight matrix for neighborhood aggregation. B_k weight matrix for transforming hidden vector of self. h_v^k the hidden representation of node v at layer k.

Training the model, Matrix Formulation

- How do we train the GCN to generate embeddings?
- Need to define a loss function on the embeddings.
- We can feed embeddings h_v^k into any loss function and run SGD to train the weight parameters.
- Many aggregations can be performed efficiently by (sparse) matrix operations.
- We define the matrix of hidden embeddings as

  $H^{(k)} = \left[\, h_1^{(k)} \ \dots \ h_V^{(k)} \right]^T$

- Let D is the "degree" matrix. It is a diagonal matrix listing the number of neighbors $N\,(v)$ of any node v

  $D_{v,v}$ = Deg (v) = $|N\,(v)\,|$

- The inverse of D: D^{-1} is also diagonal:

  $D^{-1}_{v,v}$ = $1/\,|N(v)|$

- $\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|}$ can be written as $H^{(k+1)} = D^{-1} A H^{(k)}$

Matrix formulation

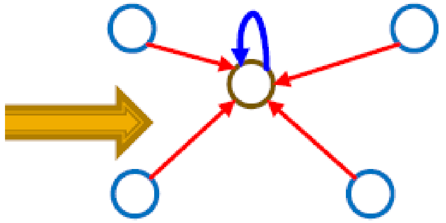
- One can show that the update function could be written in the matrix form as:

$$H^{(k+1)} = \sigma(D^{-1}AH^{(k)}W_k^T + H^{(k)}B_k^T)$$

- Red: neighborhood aggregation
- Blue: self transformation

$$H^{(k)} = \left[\, h_1^{(k)} \ \dots \ h_V^{(k)} \right]^T$$

is hidden state (embeddings) of all nodes in layer k



- In practice, this implies that efficient sparse matrix multiplication can be used (matrix $D^{-1}A$ is sparse)
- **Note**: not all GNNs can be expressed in matrix form. Sometimes, aggregation function is too complex to be written as a matrix expression.

- Number of layers (k) in GNNs is not arbitrarily large. Typically, we go through a small number recursions. Practice has shown that too many recursions would result in smoothing out of node and edge features and all states of all nodes will start approaching the same value.

Node Similarity

- When calculating Cross Entropy for the Loss Function we will use node similarity. The simplest **node similarity** can be defined as “nodes u, v are similar if they are connected by an edge.”
- This means: $z_v^T z_u$= $A_{u,v}$ which is the $(u$,$v)$ entry of the graph adjacency matrix A
- Therefore, $Z^T Z = A$
- Node could be similar even if they are not adjacent.
- There are several stochastic techniques for assessing that similarity using techniques like the deep (random) walk over node vicinity.
- Most importantly, two nodes are similar if they have high value of the cosine similarity between their embedded (latent) vectors.

Loss Function

- *Similar nodes have similar embeddings*. The loss function is defined as sum over all node pairs:

$$L = \sum_{(z_u, z_v)} CE(y_{u,v}, (z_u, z_v))$$

- Where $y_{u,v}$ is node similarity and can be 1 when node u and v are **similar** or could be less than 1
- CE stands for the cross entropy.
- **Node similarity** can be calculated in different ways, using
  - **Random walks** (node2vec, DeepWalk, struc2vec)
  - **Matrix factorization**
  - **Node proximity in the graph**

Loss Function in Supervised Training

- **We could directly train** the model for a supervised task (e.g., node classification) such as discovering whether a drug is safe or toxic.
- Safety of a drug is in this case a binary feature of the drug.
- Drug-drug and drug-protein interactions are edges of the graph.
- In these problems, we usually use simpler Loss Function:

$$L = \sum_{v \epsilon V} c_v \log(\sigma(z_v^T \theta)) + (1 - c_v) \log(1 - \sigma(z_v^T \theta))$$

- Here z_v is node embedding, θ is classifcation weights and c_v is node class label.

Inductive Capability

- Inductive node embedding -> generalize to entirely unseen graphs
  - We could train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B.
- Many application settings constantly encounter previously unseen nodes:
  - We need to generate new embeddings “on the fly”

Graph Attention Networks

- In GANs, recursive equation for embedding vector of node v in layer k is defined as:

$$h_v^{(k)} = \sigma\left(\sum_{u \epsilon N(v)} W^{(k-1)} \alpha_{vu} h_u^{(k-1)}\right)$$

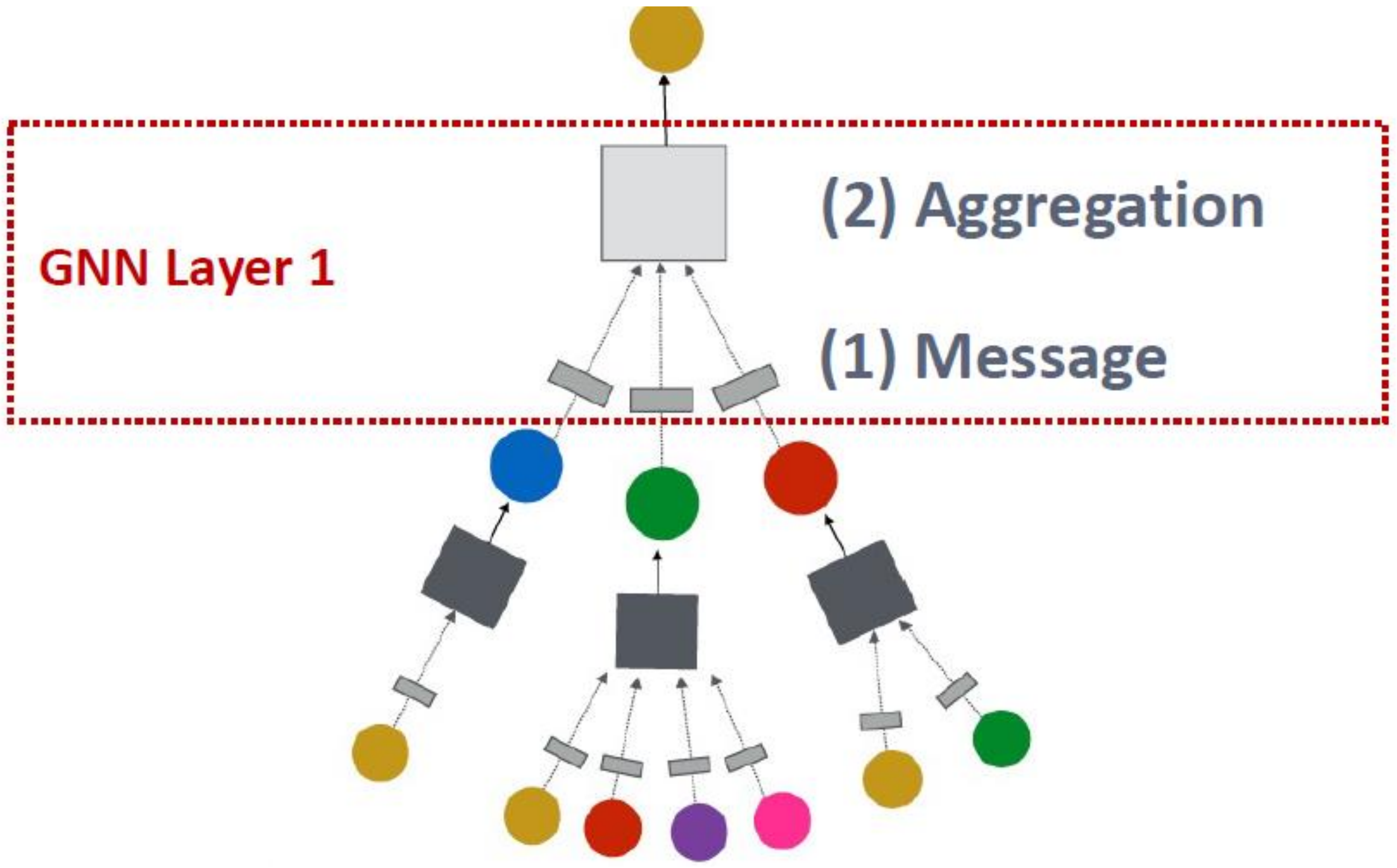
- In Graph Convolutional Networks(GCNs) $\alpha_{vu} = {}^1\!/_{N(v)}$ is the weighting factor (importance) of the message coming from note u to node v
- α_{vu} is defined **explicitly** based on the structural properties of the graph (node degree) and the assumption that all neighbors $u \in N(v)$ are equally important to node v.
- Attention model starts with assumption that **not all node's neighbors are equally important**
- **Attention** is inspired by cognitive attention and the attention mechanism we will study in NLP
- The **attention** α_{vu} focuses on the important parts of the input data and fades out the rest.
- **Idea:** the NN should devote more computing power on that small but important part of the data.
- Through training, depending on the context, network will learn attention coefficients and find which neighbors are more important.

Message Passing GNNs

- The basic intuition behind the GNN message-passing framework is straightforward: at each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph. To be precise: after the first iteration ($k = 1$), every node embedding contains information from its 1-hop neighborhood, i.e., every node embedding contains information about the features of its immediate graph neighbors, which can be reached by a path of length 1 in the graph; after the second iteration ($k = 2$) every node embedding contains information from its 2-hop neighborhood; and in general, after k iterations every node embedding contains information about its k-hop neighborhood.
- But what kind of "information" do these node embeddings actually encode? Generally, this information comes in two forms. On the one hand there is structural information about the graph. For example, after k iterations of GNN message passing, the embedding $h_u^{(k)}$ of node u might encode information about the degrees of all the nodes in u's k-hop neighborhood. This structural information can be useful for many tasks. For instance, when analyzing molecular graphs, we can use degree information to infer atom types and different structural motifs, such as benzene rings.
- In addition to structural information, the other key kind of information captured by GNN node embedding is feature-based. After k iterations of GNN message passing, the embeddings for each node also encode information about all the features in their k-hop neighborhood. This local feature-aggregation behavior of GNNs is analogous to the behavior of the convolutional kernels in convolutional neural networks (CNNs). However, whereas CNNs aggregate feature information from spatially-defined patches in an image, GNNs aggregate information based on local graph neighborhoods

GNN Layer = Message + Aggregation

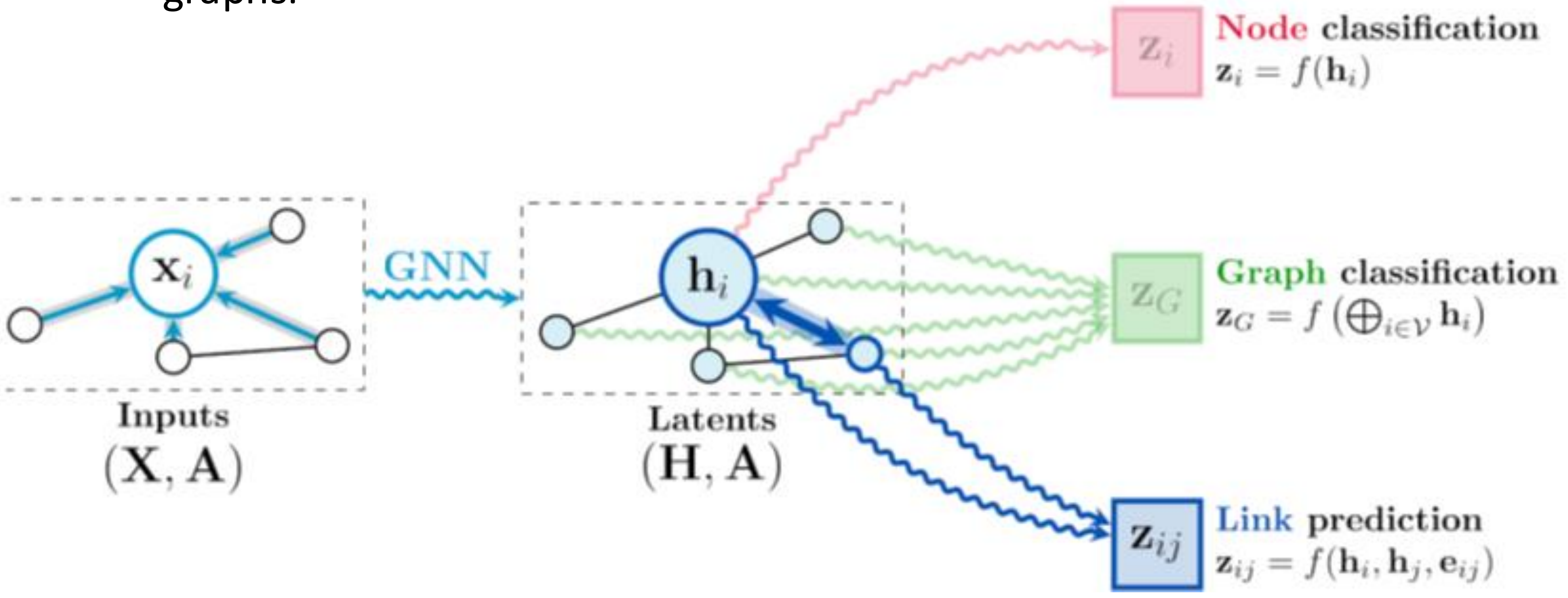
- We can generalize operations in all GNN as different instantiations under this perspective:



Types of GNNs

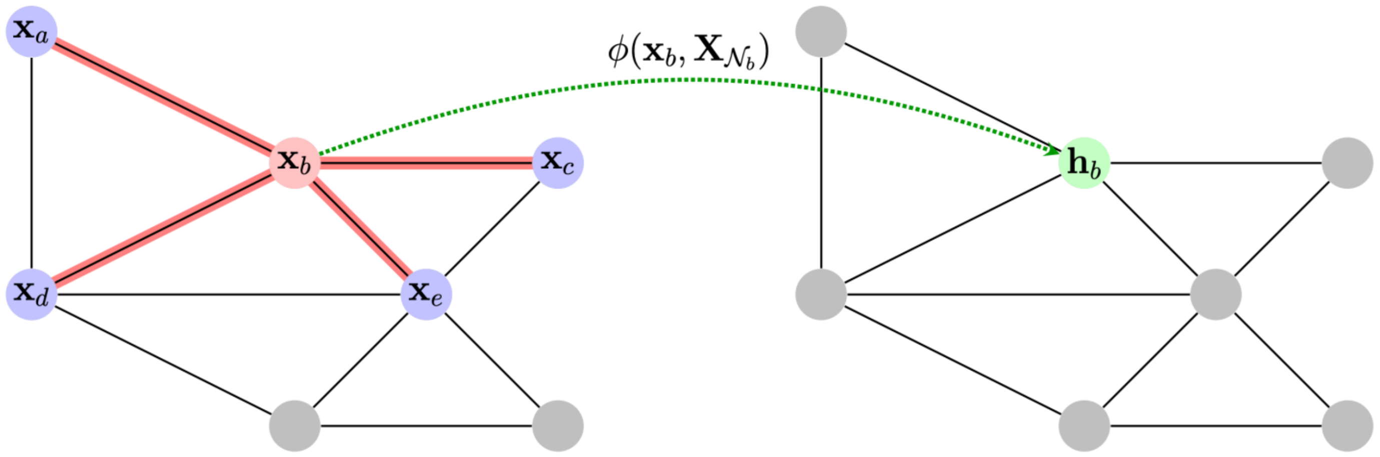
General blueprint for learning on graphs

- Given some functions X on the domain made of nodes and/or edges of a graph with topology described by adjacency matrix A, we calculate embeddings or latent values H.
- Starting with latent values H and adjacency matrix A, we could make:
  - predictions on properties of nodes or perform node classifications,
  - predictions on the properties of edges or classification of edges, and make
  - predictions on the properties of the entire graphs or classification of graphs.



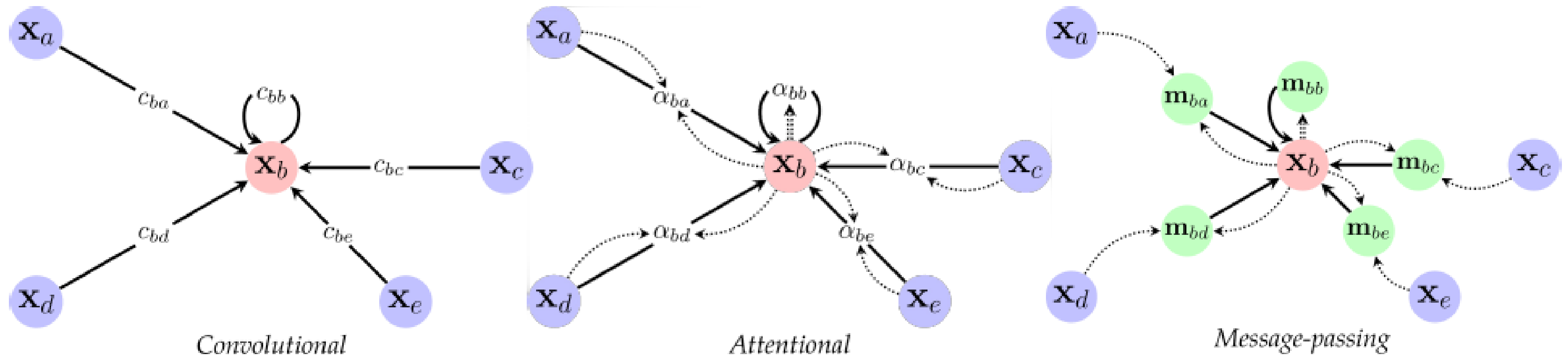
$\mathbf{x}_i$
GNN
$\mathbf{h}_i$
Inputs
$(\mathbf{X}, \mathbf{A})$
Latents
$(\mathbf{H}, \mathbf{A})$
$\mathbf{z}_i$
Node classification
$\mathbf{z}_i = f(\mathbf{h}_i)$
$\mathbf{z}_G$
Graph classification
$\mathbf{z}_G = f\left(\bigoplus_{i\in\mathcal{V}} \mathbf{h}_i\right)$
$\mathbf{z}_{ij}$
Link prediction
$\mathbf{z}_{ij} = f(\mathbf{h}_i, \mathbf{h}_j, \mathbf{e}_{ij})$

Recipe for graph neural networks, visualized



- Results of any calculations described on the previous slide in any layer of any GNN are vector ***embeddings*** or the ***latent vectors*** h_b
- We build permutation equivariant functions $\mathbf{F}(\mathbf{X}, \mathbf{A})$ on graphs by shared application of a *local* permutation-invariant function $\phi(x_i, X_{N_i})$
- Common terminology used in GNNs is:
  - $\mathbf{F}$ is a *"GNN layer"*
  - ϕ is *"diffusion" , "propagation" or "message passing"*
- **How** do we implement ϕ is a **very intense** area of research!
- Fortunately, *almost all* of layers can be classified across three *"flavors"*

The three "flavors" of GNN layers



$$\mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} c_{ij}\psi(\mathbf{x}_j)\right) \qquad \mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} a(\mathbf{x}_i, \mathbf{x}_j)\psi(\mathbf{x}_j)\right) \qquad \mathbf{h}_i = \phi\left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}_i} \psi(\mathbf{x}_i, \mathbf{x}_j)\right)$$

- Most of GNNs can be classified as **Convolutional, Attention** or **Message-passing**.
- Symbol $\bigoplus$ stands for some generalized aggregation. It could be summation, multiplication, concatenation, whatever.
- **Convolutional GNNs** are most frequently encountered in social networks, where edges encode similarity between nodes. Highly scalable. Most industry apps are Conv GNNs
- **Attention GNNs** aggregate features of neighbors with implicit weights (attention) $\alpha_{ij} = a(x_i, x_j)$. Useful as "middle ground" w.r.t. capacity, scale, interpretability. Edges need not encode similarity between nodes. These networks are still computing only a **scalar** per edge.
- **Message-passing GNNs** send arbitrary vectors (messages) across edges. Messages computed as $m_{ij} = \psi(x_i, x_j)$. Most generic GNN layer. Edges give "recipe" for passing data. Have scalability or learnability issues. Ideal for computational chemistry, reasoning and simulation tasks

Node Classification with GNNs

Node Classification with Graph Neural Networks

- This example demonstrate an implementation of a Graph Neural Network (GNN) model.
- The model is used for a node prediction task on the Cora dataset of scientific papers. The model will predict the subject of a paper given its words and citations network.
- **We implement a Graph Convolution Layer from scratch** to provide better understanding of how they work. However, there are specialized TensorFlow-based libraries that provide rich GNN APIs, such as Spectral, StellarGraph, and GraphNets.

Setup: We need `networkx`, a Python library for network manipulation and analysis

```
!pip install network
```

```
import os
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Fetch the Dataset

- The Cora dataset consists of 2,708 scientific papers classified into one of seven classes. The citation network consists of 5,429 links. Each paper has a binary word vector of size 1,433, indicating the presence of a word with a given index.
- The dataset has two tab-separated files: `cora.cites` and `cora.content`.
- The `cora.cites` includes the citation records with two columns: `cited_paper_id` (target) and `citing_paper_id` (source).
- The `cora.content` includes the paper content records with 1,435 columns: `paper_id`, `subject`, and `1,433 binary features.`

Download the dataset

```
zip_file = keras.utils.get_file(
    fname="cora.tgz",
    origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",
    extract=True,
)
data_dir = os.path.join(os.path.dirname(zip_file), "cora")
```

- The dataset ends up in you home directory, subdirectory `.keras/datasets/cora.` On my Windws machine that is: `C:\Users\Zoran\.keras\datasets\cora`

Process and visualize the dataset, `citations`

- Load the citations data, file `cora.cites` into a Pandas DataFrame.

```
citations = pd.read_csv(
    os.path.join(data_dir, "cora.cites"),
    sep="\t",
    header=None,
    names=["target", "source"],
)
print("Citations shape:", citations.shape)
```

```
Citations shape: (5429, 2)
```

- Next we display a sample of the citations DataFrame. The `target` column includes the paper `ids` cited by the paper `ids` in the `source` column.

```
citations.sample(frac=1).head()
```

	target	source
65	35	1956
5037	578646	1128291
1195	6184	1120731
3927	100197	447250
4686	250566	1113926

Visualize `papers`

- We will load the `papers` data into a Pandas DataFrame.

```
column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] + ["subject"]
papers = pd.read_csv(
    os.path.join(data_dir, "cora.content"), sep="\t", header=None,
names=column_names,
)
print("Papers shape:", papers.shape)
```

Papers shape: (2708, 1435)

- The DataFrame `papers` includes the `paper_id` and the `subject` columns, as well as 1,433 binary column representing whether a term exists in the paper or not. A sample of `papers` reads:

```
print(papers.sample(5).T)
              615      1863              2594           2079            1005
paper_id    101263   107252           1105603         101660          416964
term_0           0        0                 0              0               0
term_1           0        0                 0              0               0
term_2           0        0                 0              0               0
term_3           0        0                 0              0               0
...            ...      ...               ...            ...             ...
term_1429        0        0                 0              0               0
term_1430        0        0                 0              0               0
term_1431        0        0                 0              0               0
term_1432        0        0                 0              0               0
subject     Theory   Theory   Neural_Networks  Rule_Learning Neural_Networks

[1435 rows x 5 columns]
```

- Note that we are presenting a transposed matrix `sample(5)`. Every paper is a column.

The count of the papers in each subject

```
print(papers.subject.value_counts())
Neural_Networks           818
Probabilistic_Methods     426
Genetic_Algorithms        418
Theory                    351
Case_Based                298
Reinforcement_Learning    217
Rule_Learning             180
Name: subject, dtype: int64
```

- Next, we convert the paper ids and the subjects into zero-based indices.

```
class_values = sorted(papers["subject"].unique())
class_idx = {name: id for id, name in enumerate(class_values)}
paper_idx = {name: idx for idx, name in
enumerate(sorted(papers["paper_id"].unique()))}
```

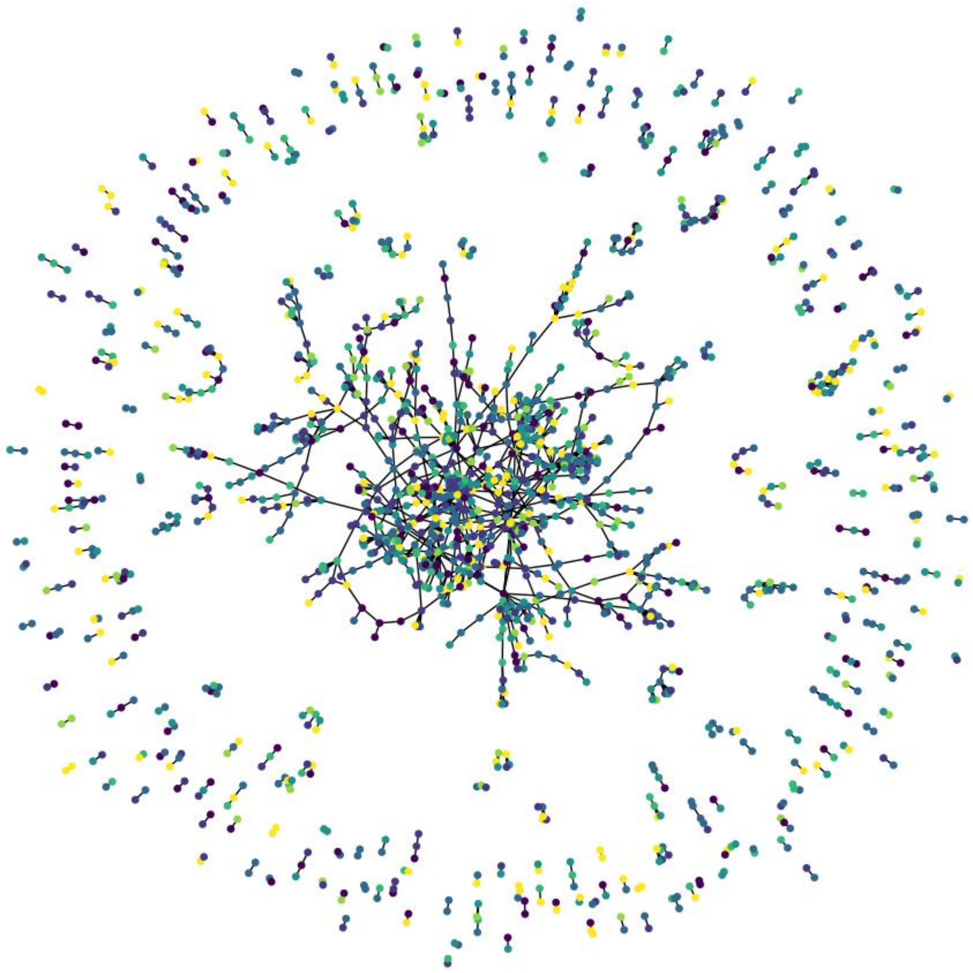
```
papers["paper_id"] = papers["paper_id"].apply(lambda name: paper_idx[name])
citations["source"] = citations["source"].apply(lambda name: paper_idx[name])
citations["target"] = citations["target"].apply(lambda name: paper_idx[name])
papers["subject"] = papers["subject"].apply(lambda value: class_idx[value])
```

The citation graph

- Now let's visualize the citation graph. Each node in the graph represents a paper, and the color of the node corresponds to its subject. We only show a sample of the papers in the dataset.

```
plt.figure(figsize=(10, 10))
colors = papers["subject"].tolist()
cora_graph = nx.from_pandas_edgelist(citations.sample(n=1500))
subjects =
list(papers[papers["paper_id"].isin(list(cora_graph.nodes))]["subject"])
nx.draw_spring(cora_graph, node_size=15, node_color=subjects)
```

The citation graph



Split the dataset into stratified train and test sets

```
train_data, test_data = [], []

for _, group_data in papers.groupby("subject"):
    # Select around 50% of the dataset for training.
    random_selection = np.random.rand(len(group_data.index)) <= 0.5
    train_data.append(group_data[random_selection])
    test_data.append(group_data[~random_selection])

train_data = pd.concat(train_data).sample(frac=1)
test_data = pd.concat(test_data).sample(frac=1)

print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
```

```
Train data shape: (1325, 1435)
Test data shape: (1383, 1435)
```

Train and Evaluate Experiment

```
hidden_units = [32, 32]
learning_rate = 0.01
dropout_rate = 0.5
num_epochs = 300
batch_size = 256
```

- This function compiles and trains an input model using the given training data.

```
def run_experiment(model, x_train, y_train):
    # Compile the model.
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],
    )
    # Create an early stopping callback.
    early_stopping = keras.callbacks.EarlyStopping(
        monitor="val_acc", patience=50, restore_best_weights=True
    )
    # Fit the model.
    history = model.fit(
        x=x_train,
        y=y_train,
        epochs=num_epochs,
        batch_size=batch_size,
        validation_split=0.15,
        callbacks=[early_stopping],
    )
    return history
```

The loss and accuracy curves

- This function displays the loss and accuracy curves of the model during training.

```
def display_learning_curves(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    ax1.plot(history.history["loss"])
    ax1.plot(history.history["val_loss"])
    ax1.legend(["train", "test"], loc="upper right")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")

    ax2.plot(history.history["acc"])
    ax2.plot(history.history["val_acc"])
    ax2.legend(["train", "test"], loc="upper right")
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Accuracy")
    plt.show()
```

Feedforward Network (FFN) Model

- We will use this module in the baseline and the GNN models.

```
def create_ffn(hidden_units, dropout_rate, name=None):
    fnn_layers = []

    for units in hidden_units:
        fnn_layers.append(layers.BatchNormalization())
        fnn_layers.append(layers.Dropout(dropout_rate))
        fnn_layers.append(layers.Dense(units, activation=tf.nn.gelu))

    return keras.Sequential(fnn_layers, name=name)
```

- Prepare the data for the baseline model

```
feature_names = set(papers.columns) - {"paper_id", "subject"}
num_features = len(feature_names)
num_classes = len(class_idx)

# Create train and test features as a numpy array.
x_train = train_data[list(feature_names)].to_numpy()
x_test = test_data[list(feature_names)].to_numpy()
# Create train and test targets as a numpy array.
y_train = train_data["subject"]
y_test = test_data["subject"]
```

Implement a baseline classifier

- We add five FFN blocks with skip connections, so that we generate a baseline model with roughly the same number of parameters as the GNN models to be built later.

```
def create_baseline_model(hidden_units, num_classes, dropout_rate=0.2):
    inputs = layers.Input(shape=(num_features,), name="input_features")
    x = create_ffn(hidden_units, dropout_rate, name=f"ffn_block1")(inputs)
    for block_idx in range(4):
        # Create an FFN block.
        x1 = create_ffn(hidden_units, dropout_rate, 
name=f"ffn_block{block_idx + 2}")(x)
        # Add skip connection.
        x = layers.Add(name=f"skip_connection{block_idx + 2}")([x, x1])
    # Compute logits.
    logits = layers.Dense(num_classes, name="logits")(x)
    # Create the model.
    return keras.Model(inputs=inputs, outputs=logits, name="baseline")


baseline_model = create_baseline_model(hidden_units, num_classes, 
dropout_rate)
baseline_model.summary()
```

baseline_model.summary()

```
Model: "baseline"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_features (InputLayer)    [(None, 1433)]       0           []

 ffn_block1 (Sequential)        (None, 32)           52804       ['input_features[0][0]']

 ffn_block2 (Sequential)        (None, 32)           2368        ['ffn_block1[0][0]']

 skip_connection2 (Add)         (None, 32)           0           ['ffn_block1[0][0]',
                                                                  'ffn_block2[0][0]']

 ffn_block3 (Sequential)        (None, 32)           2368        ['skip_connection2[0][0]']

 skip_connection3 (Add)         (None, 32)           0           ['skip_connection2[0][0]',
                                                                  'ffn_block3[0][0]']

 ffn_block4 (Sequential)        (None, 32)           2368        ['skip_connection3[0][0]']

 skip_connection4 (Add)         (None, 32)           0           ['skip_connection3[0][0]',
                                                                  'ffn_block4[0][0]']

 ffn_block5 (Sequential)        (None, 32)           2368        ['skip_connection4[0][0]']

 skip_connection5 (Add)         (None, 32)           0           ['skip_connection4[0][0]',
                                                                  'ffn_block5[0][0]']

 logits (Dense)                 (None, 7)            231         ['skip_connection5[0][0]']

==================================================================================================
Total params: 62,507
Trainable params: 59,065
Non-trainable params: 3,442
```

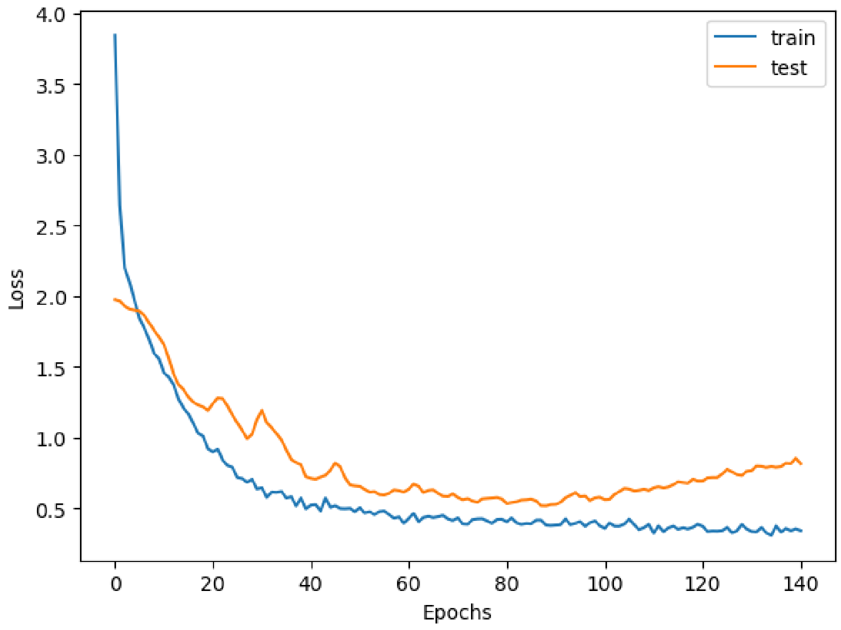
Train the baseline classifier

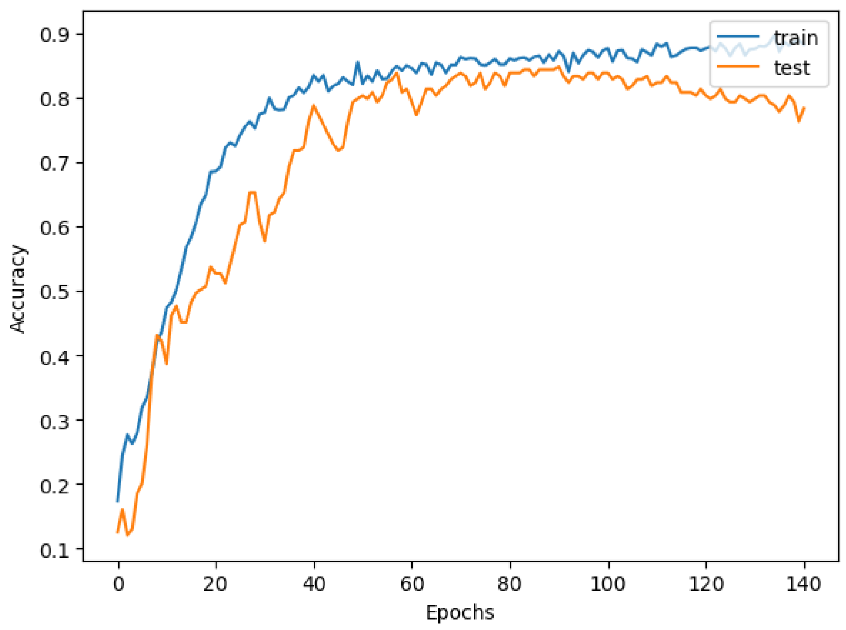
```
history = run_experiment(baseline_model, x_train, y_train)

Epoch 1/300
5/5 [==============================] - 8s 147ms/step - loss: 3.8450 - acc: 
0.1741 - val_loss: 1.9753 - val_acc: 0.1256
Epoch 2/300
5/5 [==============================] - 0s 21ms/step - loss: 2.6381 - acc: 
0.2451 - val_loss: 1.9667 - val_acc: 0.1608
Epoch 3/300
. . . .
Epoch 140/300
5/5 [==============================] - 0s 18ms/step - loss: 0.3560 - acc: 
0.8899 - val_loss: 0.8551 - val_acc: 0.7638
Epoch 141/300
5/5 [==============================] - 0s 24ms/step - loss: 0.3432 - acc: 
0.8837 - val_loss: 0.8169 - val_acc: 0.7839
```

The learning curves

- Let's plot the learning curves.

```
display_learning_curves(history)
```





- Evaluate the baseline model on the test data split.

```
_, test_accuracy = baseline_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

Examine the baseline model predictions

- We will create new data instances by randomly generating binary word vectors with respect to the word presence probabilities.

```
def generate_random_instances(num_instances):
    token_probability = x_train.mean(axis=0)
    instances = []
    for _ in range(num_instances):
        probabilities = np.random.uniform(size=len(token_probability))
        instance = (probabilities <= token_probability).astype(int)
        instances.append(instance)

    return np.array(instances)


def display_class_probabilities(probabilities):
    for instance_idx, probs in enumerate(probabilities):
        print(f"Instance {instance_idx + 1}:")
        for class_idx, prob in enumerate(probs):
            print(f"- {class_values[class_idx]}: {round(prob * 100, 2)}%")
```

```
new_instances = generate_random_instances(num_classes)
logits = baseline_model.predict(new_instances)
probabilities =
keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
1/1 [==============================] - 2s 2s/step
Instance 1:
- Case_Based: 5.96%
- Genetic_Algorithms: 1.55%
- Neural_Networks: 13.87%
- Probabilistic_Methods: 1.23%
- Reinforcement_Learning: 2.96%
- Rule_Learning: 28.57%
- Theory: 45.86%
Instance 2:
- Case_Based: 14.55%
- Genetic_Algorithms: 11.0%
- Neural_Networks: 34.77%
- Probabilistic_Methods: 3.09%
- Reinforcement_Learning: 0.92%
- Rule_Learning: 0.85%
- Theory: 34.83%
```

The baseline model predictions

```
Instance 3:
- Case_Based: 0.47%
- Genetic_Algorithms: 97.14%
- Neural_Networks: 1.07%
- Probabilistic_Methods: 0.22%
- Reinforcement_Learning: 0.6%
- Rule_Learning: 0.09%
- Theory: 0.4%
Instance 4:
- Case_Based: 1.16%
- Genetic_Algorithms: 0.71%
- Neural_Networks: 91.4%
- Probabilistic_Methods: 0.19%
- Reinforcement_Learning: 1.05%
- Rule_Learning: 0.49%
- Theory: 4.99%
Instance 5:
- Case_Based: 0.36%
- Genetic_Algorithms: 81.57%
- Neural_Networks: 14.92%
- Probabilistic_Methods: 2.47%
- Reinforcement_Learning: 0.52%
- Rule_Learning: 0.05%
- Theory: 0.1%
Instance 6:
- Case_Based: 0.17%
- Genetic_Algorithms: 0.94%
- Neural_Networks: 18.5%
- Probabilistic_Methods: 0.3%
- Reinforcement_Learning: 79.23%
- Rule_Learning: 0.35%
- Theory: 0.51%
Instance 7:
- Case_Based: 2.87%
- Genetic_Algorithms: 4.82%
- Neural_Networks: 32.97%
- Probabilistic_Methods: 5.3%
- Reinforcement_Learning: 4.51%
- Rule_Learning: 9.11%
- Theory: 40.43%
```

Graph Neural Network Model

- Preparing and loading the graphs data into the model for training is the most challenging part in GNN models, which is addressed in different ways by the specialized libraries. In this example, we show a simple approach for preparing and using graph data that is suitable if your dataset consists of a single graph that fits entirely in memory.
- The graph data is represented by the `graph_info` tuple, which consists of the following three elements:
  1. `node_features`: This is a [`num_nodes, num_features`] NumPy array that includes the `node` features. In this dataset, the `nodes` are the papers, and the `node_features` are the word-presence binary vectors of each paper.
  2. `edges`: This is [`num_edges, num_edges`] NumPy array representing a sparse adjacency matrix of the links between the nodes. In this example, the `links` are the `citations` between the papers.
  3. `edge_weights` (optional): This is a [`num_edges`] NumPy array that includes the `edge` weights, which quantify the relationships between nodes in the graph. In this example, there are no weights for the paper citations.

`graph_info` tuple

```
# Create an edges array (sparse adjacency matrix) of shape [2, num_edges].
edges = citations[["source", "target"]].to_numpy().T

# Create an edge weights array of ones.
edge_weights = tf.ones(shape=edges.shape[1])

# Create a node features array of shape [num_nodes, num_features].
node_features = tf.cast(
    papers.sort_values("paper_id")[list(feature_names)].to_numpy(),
dtype=tf.dtypes.float32
)

# Create graph info tuple with node_features, edges, and edge_weights.
graph_info = (node_features, edges, edge_weights)

print("Edges shape:", edges.shape)
print("Nodes shape:", node_features.shape)
```

```
Edges shape: (2, 5429)
Nodes shape: (2708, 1433)
```

Graph convolution layer

- We implement a graph convolution module as a Keras Layer. Our GraphConvLayer performs the following steps:

1. **Prepare**: The input node representations are processed using a FFN to produce a *message*. You can simplify the processing by only applying linear transformation to the representations.
2. **Aggregate**: The messages of the neighbors of each node are aggregated with respect to the `edge_weights` using a permutation invariant pooling operation, such as sum, mean, and max, to prepare a single aggregated message for each node. See, for example, `tf.math.unsorted_segment_sum` APIs used to aggregate neighbor messages.
3. **Update**: The `node_repesentations` and `aggregated_messages`—both of shape [`num_nodes, representation_dim`]— are combined and processed to produce the new state of the node representations (node embeddings). If `combination_type` is gru, the `node_repesentations` and `aggregated_messages` are stacked to create a sequence, then processed by a GRU layer. Otherwise, the node_repesentations and `aggregated_messages` are added or concatenated, then processed using a FFN.

- The technique implemented use ideas from Graph Convolutional Networks, GraphSage, Graph Isomorphism Network, Simple Graph Networks, and Gated Graph Sequence Neural Networks. Two other key techniques that are not covered are Graph Attention Networks and Message Passing Neural Networks.

Class GraphConvLayer

```
class GraphConvLayer(layers.Layer):
    def __init__(
        self,
        hidden_units,
        dropout_rate=0.2,
        aggregation_type="mean",
        combination_type="concat",
        normalize=False,
        *args,
        **kwargs,):
        super(GraphConvLayer, self).__init__(*args, **kwargs)
        self.aggregation_type = aggregation_type
        self.combination_type = combination_type
        self.normalize = normalize

        self.ffn_prepare = create_ffn(hidden_units, dropout_rate)
        if self.combination_type == "gated":
            self.update_fn = layers.GRU(
                units=hidden_units,
                activation="tanh",
                recurrent_activation="sigmoid",
                dropout=dropout_rate,
                return_state=True,
                recurrent_dropout=dropout_rate,
            )
        else:
            self.update_fn = create_ffn(hidden_units, dropout_rate)
```

Class GraphConvLayer

```
def prepare(self, node_repesentations, weights=None):
        # node_repesentations shape is [num_edges, embedding_dim].
        messages = self.ffn_prepare(node_repesentations)
        if weights is not None:
            messages = messages * tf.expand_dims(weights, -1)
        return messages

def aggregate(self, node_indices, neighbour_messages, node_repesentations):
        # node_indices shape is [num_edges].
        # neighbour_messages shape: [num_edges, representation_dim].
        # node_repesentations shape is [num_nodes, representation_dim].
        num_nodes = node_repesentations.shape[0]
        if self.aggregation_type == "sum":
            aggregated_message = tf.math.unsorted_segment_sum(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "mean":
            aggregated_message = tf.math.unsorted_segment_mean(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "max":
            aggregated_message = tf.math.unsorted_segment_max(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        else:
            raise ValueError(f"Invalid aggregation type: {self.aggregation_type}.")

        return aggregated_message
```

Class GraphConvLayer

```
def update(self, node_repesentations, aggregated_messages):
        # node_repesentations shape is [num_nodes, representation_dim].
        # aggregated_messages shape is [num_nodes, representation_dim].
        if self.combination_type == "gru":
            # Create a sequence of two elements for the GRU layer.
            h = tf.stack([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "concat":
            # Concatenate the node_repesentations and aggregated_messages.
            h = tf.concat([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "add":
            # Add node_repesentations and aggregated_messages.
            h = node_repesentations + aggregated_messages
        else:
            raise ValueError(f"Invalid combination type: {self.combination_type}.")

        # Apply the processing function.
        node_embeddings = self.update_fn(h)
        if self.combination_type == "gru":
            node_embeddings = tf.unstack(node_embeddings, axis=1)[-1]

        if self.normalize:
            node_embeddings = tf.nn.l2_normalize(node_embeddings, axis=-1)
        return node_embeddings
```

Class `GraphConvLayer`

```
def call(self, inputs):
        """Process the inputs to produce the node_embeddings.

        inputs: a tuple of three elements: node_repesentations, edges, edge_weights.
        Returns: node_embeddings of shape [num_nodes, representation_dim].
        """

        node_repesentations, edges, edge_weights = inputs
        # Get node_indices (source) and neighbour_indices (target) from edges.
        node_indices, neighbour_indices = edges[0], edges[1]
        # neighbour_repesentations shape is [num_edges, representation_dim].
        neighbour_repesentations = tf.gather(node_repesentations, neighbour_indices)

        # Prepare the messages of the neighbours.
        neighbour_messages = self.prepare(neighbour_repesentations, edge_weights)
        # Aggregate the neighbour messages.
        aggregated_messages = self.aggregate(
            node_indices, neighbour_messages, node_repesentations
        )
        # Update the node embedding with the neighbour messages.
        return self.update(node_repesentations, aggregated_messages)
```

Graph neural network node classifier

- The GNN classification model proceeds, as follows:
  1. Apply preprocessing using FFN to the node features to generate initial node representations.
  2. Apply one or more graph convolutional layer, with skip connections, to the node representation to produce node embeddings.
  3. Apply post-processing using FFN to the node embeddings to generat the final node embeddings.
  4. Feed the node embeddings in a Softmax layer to predict the node class.
- Each graph convolutional layer added captures information from a further level of neighbours. However, adding many graph convolutional layer can cause oversmoothing, where the model produces similar embeddings for all the nodes.
- Note that the graph_info passed to the constructor of the Keras model, and used as a property of the Keras model object, rather than input data for training or prediction. The model will accept a batch of node_indices, which are used to lookup the node features and neighbours from the graph_info.

Class GNNNodeClassifier

```
class GNNNodeClassifier(tf.keras.Model):
    def __init__(
        self,
        graph_info,
        num_classes,
        hidden_units,
        aggregation_type="sum",
        combination_type="concat",
        dropout_rate=0.2,
        normalize=True,
        *args,
        **kwargs,
    ):
        super(GNNNodeClassifier, self).__init__(*args, **kwargs)

        # Unpack graph_info to three elements: node_features, edges, and edge_weight.
        node_features, edges, edge_weights = graph_info
        self.node_features = node_features
        self.edges = edges
        self.edge_weights = edge_weights
        # Set edge_weights to ones if not provided.
        if self.edge_weights is None:
            self.edge_weights = tf.ones(shape=edges.shape[1])
        # Scale edge_weights to sum to 1.
        self.edge_weights = self.edge_weights / tf.math.reduce_sum(self.edge_weights)
```

Class `GNNNodeClassifier`

```
# Create a process layer.
        self.preprocess = create_ffn(hidden_units, dropout_rate, name="preprocess")
        # Create the first GraphConv layer.
        self.conv1 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv1",
        )
        # Create the second GraphConv layer.
        self.conv2 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv2",
        )
        # Create a postprocess layer.
        self.postprocess = create_ffn(hidden_units, dropout_rate, name="postprocess")
        # Create a compute logits layer.
        self.compute_logits = layers.Dense(units=num_classes, name="logits")
```

Class `GNNNodeClassifier`

```
def call(self, input_node_indices):
        # Preprocess the node_features to produce node representations.
        x = self.preprocess(self.node_features)
        # Apply the first graph conv layer.
        x1 = self.conv1((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x1 + x
        # Apply the second graph conv layer.
        x2 = self.conv2((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x2 + x
        # Postprocess node embedding.
        x = self.postprocess(x)
        # Fetch node embeddings for the input node_indices.
        node_embeddings = tf.gather(x, input_node_indices)
        # Compute logits
        return self.compute_logits(node_embeddings)
```

Instantiate `GNNNodeClassifier`

- Let's test instantiating and calling the GNN model. Notice that if you provide N node indices, the output will be a tensor of shape [N, num_classes], regardless of the size of the graph.

```
gnn_model = GNNNodeClassifier(
    graph_info=graph_info,
    num_classes=num_classes,
    hidden_units=hidden_units,
    dropout_rate=dropout_rate,
    name="gnn_model",
)
print("GNN output shape:", gnn_model([1, 10, 100]))
gnn_model.summary()
```

gnn_model.summary()

```
GNN output shape: tf.Tensor(
[[-0.08258238 -0.23063104  0.05941907  0.15656857 -0.07656706 -0.15783894
  -0.10735619]
 [-0.1807271  -0.26390904  0.00740898  0.15194565 -0.11337246 -0.15723836
  -0.16575512]
 [-0.05183263  0.10546789 -0.09391725 -0.15574552 -0.01967955  0.06629369
  -0.14769788]], shape=(3, 7), dtype=float32)
Model: "gnn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 preprocess (Sequential)     (2708, 32)                52804

 graph_conv1 (GraphConvLayer  multiple                 5888
 )

 graph_conv2 (GraphConvLayer  multiple                 5888
 )

 postprocess (Sequential)    (2708, 32)                2368

 logits (Dense)              multiple                  231

=================================================================
Total params: 67,179
Trainable params: 63,481
Non-trainable params: 3,698

_________________________________
```

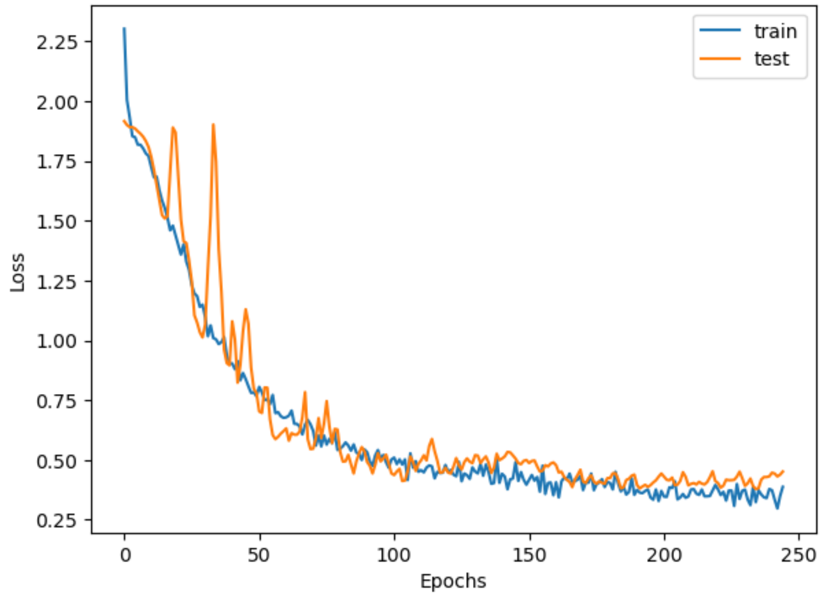
Train the GNN model

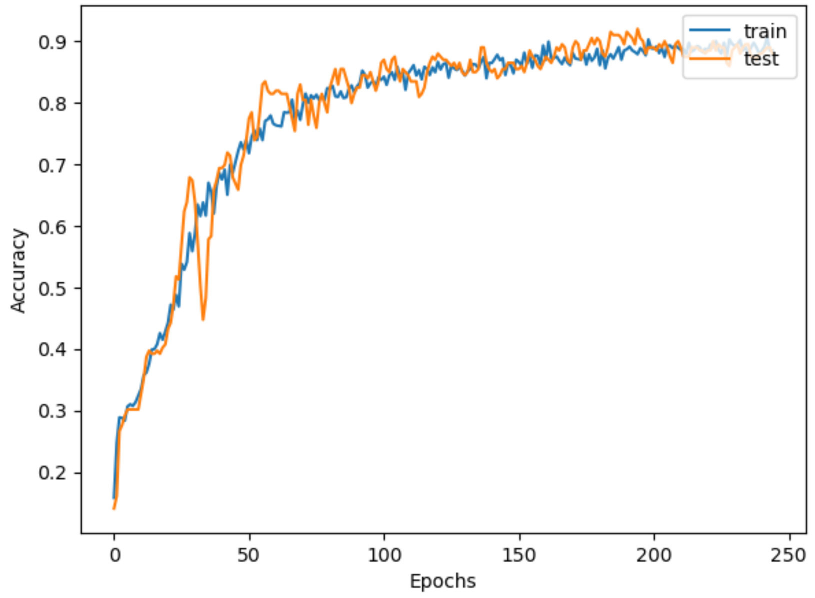
- Note that we use the standard supervised cross-entropy loss to train the model. However, we can add another self-supervised loss term for the generated node embeddings that makes sure that neighbouring nodes in graph have similar representations, while faraway nodes have dissimilar representations.

```
x_train = train_data.paper_id.to_numpy()
history = run_experiment(gnn_model, x_train, y_train)

Epoch 1/300
5/5 [==============================] - 4s 157ms/step - loss: 2.3026 - acc:
0.1581 - val_loss: 1.9164 - val_acc: 0.1407
Epoch 2/300
5/5 [==============================] - 0s 23ms/step - loss: 2.0054 - acc:
0.2469 - val_loss: 1.9004 - val_acc: 0.1608
Epoch 3/300
5/5 [==============================] - 0s 24ms/step - loss: 1.9330 - acc:
0.2886 - val_loss: 1.8916 - val_acc: 0.2663
. . . .
Epoch 244/300
5/5 [==============================] - 0s 21ms/step - loss: 0.3491 - acc:
0.8845 - val_loss: 0.4381 - val_acc: 0.8894
Epoch 245/300
5/5 [==============================] - 0s 30ms/step - loss: 0.3872 - acc:
0.8845 - val_loss: 0.4511 - val_acc: 0.8794
```

The learning curves

```
display_learning_curves(history)
```





- Now we evaluate the GNN model on the test data split. The results may vary depending on the training sample, however the GNN model always outperforms the baseline model in terms of the test accuracy.

```
x_test = test_data.paper_id.to_numpy()
_, test_accuracy = gnn_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

Test accuracy: 76.36%

GNN model predictions

- Let's add the new instances as nodes to the node_features, and generate links (citations) to existing nodes.

```
# First we add the N new_instances as nodes to the graph
# by appending the new_instance to node_features.
num_nodes = node_features.shape[0]
new_node_features = np.concatenate([node_features, new_instances])
# Second we add the M edges (citations) from each new node to a set
# of existing nodes in a particular subject
new_node_indices = [i + num_nodes for i in range(num_classes)]
new_citations = []
for subject_idx, group in papers.groupby("subject"):
    subject_papers = list(group.paper_id)
    # Select random x papers specific subject.
    selected_paper_indices1 = np.random.choice(subject_papers, 5)
    # Select random y papers from any subject (where y < x).
    selected_paper_indices2 = np.random.choice(list(papers.paper_id), 2)
    # Merge the selected paper indices.
    selected_paper_indices = np.concatenate(
        [selected_paper_indices1, selected_paper_indices2], axis=0
    )
    # Create edges between a citing paper idx and the selected cited papers.
    citing_paper_indx = new_node_indices[subject_idx]
    for cited_paper_idx in selected_paper_indices:
        new_citations.append([citing_paper_indx, cited_paper_idx])

new_citations = np.array(new_citations).T
new_edges = np.concatenate([edges, new_citations], axis=1)
```

Update the `node_features` and the `edges`

- Now let's update the `node_features` and the `edges` in the GNN model.

```
print("Original node_features shape:", gnn_model.node_features.shape)
print("Original edges shape:", gnn_model.edges.shape)
gnn_model.node_features = new_node_features
gnn_model.edges = new_edges
gnn_model.edge_weights = tf.ones(shape=new_edges.shape[1])
print("New node_features shape:", gnn_model.node_features.shape)
print("New edges shape:", gnn_model.edges.shape)

logits = gnn_model.predict(tf.convert_to_tensor(new_node_indices))
probabilities = keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
Original node_features shape: (2708, 1433)
Original edges shape: (2, 5429)
New node_features shape: (2715, 1433)
New edges shape: (2, 5478)
1/1 [==============================] - 1s 1s/step
Instance 1:
- Case_Based: 58.17%
- Genetic_Algorithms: 19.66%
- Neural_Networks: 3.94%
- Probabilistic_Methods: 0.85%
- Reinforcement_Learning: 3.0%
- Rule_Learning: 3.86%
- Theory: 10.52%
```

Display class probabilities

```
Instance 2:
- Case_Based: 0.49%
- Genetic_Algorithms: 97.4%
- Neural_Networks: 0.74%
- Probabilistic_Methods:
0.06%
- Reinforcement_Learning:
0.83%
- Rule_Learning: 0.01%
- Theory: 0.47%
Instance 3:
- Case_Based: 0.17%
- Genetic_Algorithms: 1.37%
- Neural_Networks: 51.64%
- Probabilistic_Methods:
45.65%
- Reinforcement_Learning:
0.15%
- Rule_Learning: 0.04%
- Theory: 0.97%
Instance 4:
- Case_Based: 0.04%
- Genetic_Algorithms: 0.87%
- Neural_Networks: 87.44%
- Probabilistic_Methods:
10.25%
- Reinforcement_Learning:
0.24%
- Rule_Learning: 0.03%
- Theory: 1.13%
```

```
Instance 5:
- Case_Based: 0.05%
- Genetic_Algorithms:
99.08%
- Neural_Networks: 0.13%
- Probabilistic_Methods:
0.01%
- Reinforcement_Learning:
0.71%
- Rule_Learning: 0.0%
- Theory: 0.02%
Instance 6:
- Case_Based: 1.88%
- Genetic_Algorithms: 2.49%
- Neural_Networks: 42.68%
- Probabilistic_Methods:
0.59%
- Reinforcement_Learning:
33.32%
- Rule_Learning: 10.92%
- Theory: 8.12%
Instance 7:
- Case_Based: 0.39%
- Genetic_Algorithms: 54.4%
- Neural_Networks: 5.32%
- Probabilistic_Methods:
17.31%
- Reinforcement_Learning:
0.39%
- Rule_Learning: 0.09%
- Theory: 22.11%
```

- Notice that the probabilities of the expected subjects (to which several citations are added) are higher compared to the baseline model implying that Graph modeling of citation problem exposes more information about the data.