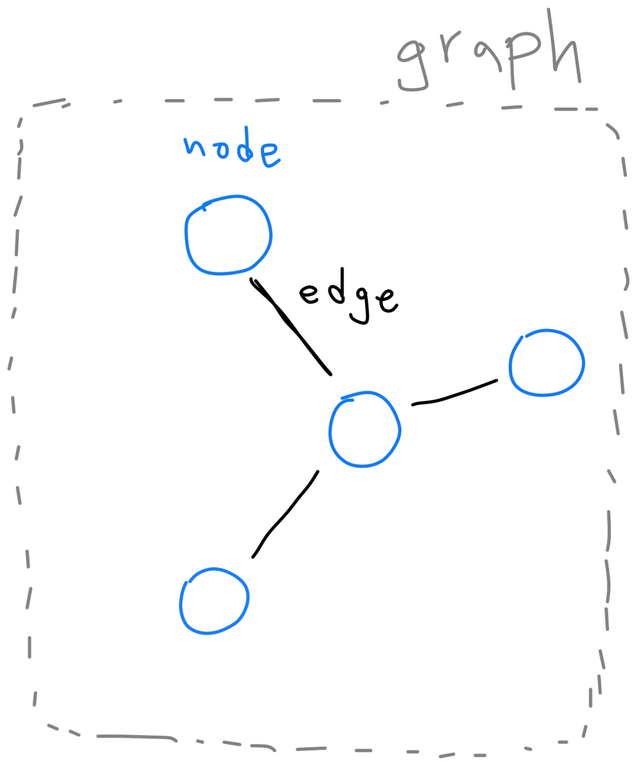
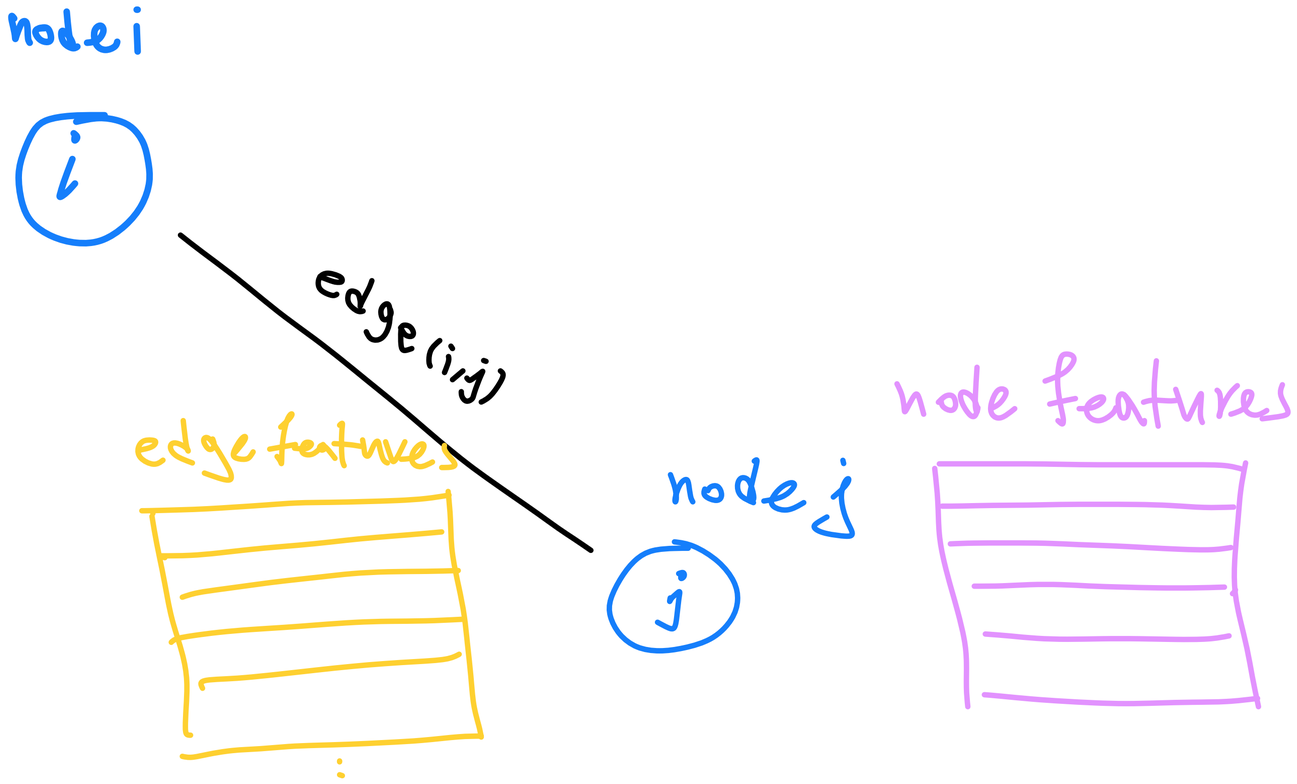
B. Data Structures, Symmetries, and Neural Network Architectures

B.1 Tabular data: MLPs, residual connections, and autoencoders
B.2 Grid-structured data: CNNs
B.3 Sequential data: RNNs, LSTMs, and GRUs
B.4 Graph-structured data: GNNs and message passing
B.5 Set-structured data: Deep Sets

Tiam Heydari, PhD
Postdoctoral Fellow, UBC

Graphs: Introduction to the concept

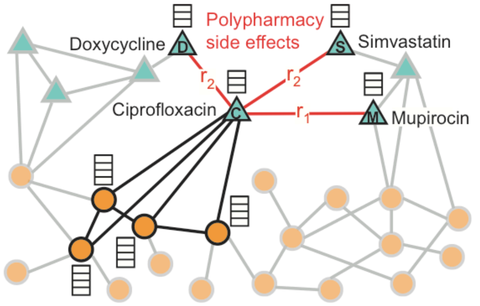


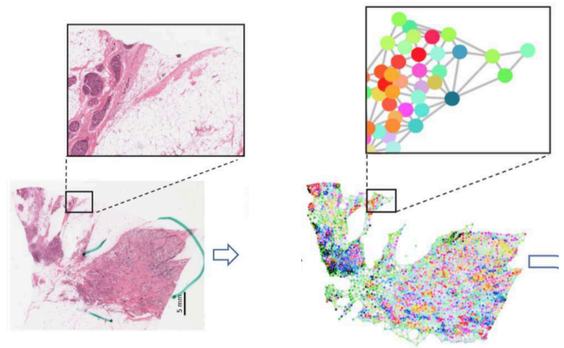


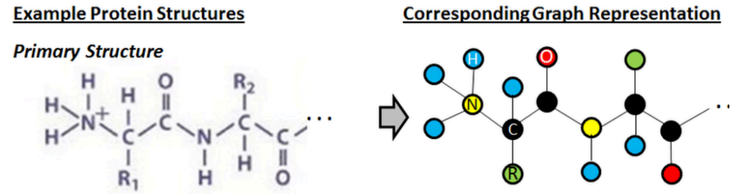
Graphs: Introduction to the concept and use cases

Some data are best naturally represented as graphs

Context	Node	Edges	Node features	Edge Features
Social Networks	Users	Friendship	Age, location, replies, interests, gender	—
Drug Repurposing	Drugs	Drug Interactions	Chemical fingerprints, molecular weight	binding affinity, side-effect profile of interaction
Pathology	Individual cells	Spatial proximity	Cell morphology, nuclear shape	Distance, spatial adjacency
Proteomics	individual atoms	bonds	Polarity, charge, hydrophobicity, atom type	Bond type, 3D distance, interaction strength



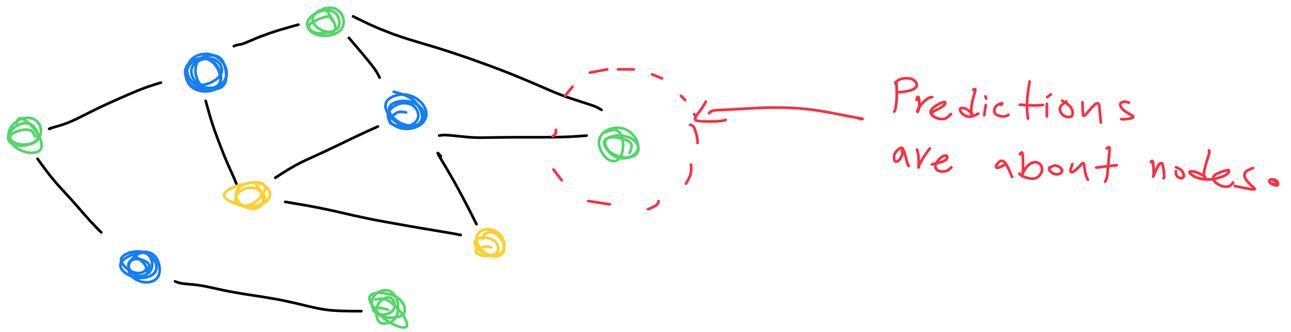




Graphs: Introduction to the concept and use cases

What are potential predictions that we are interested of graphs? We can divide these intro 3 categories based on the level of the task:

A- Node level tasks:

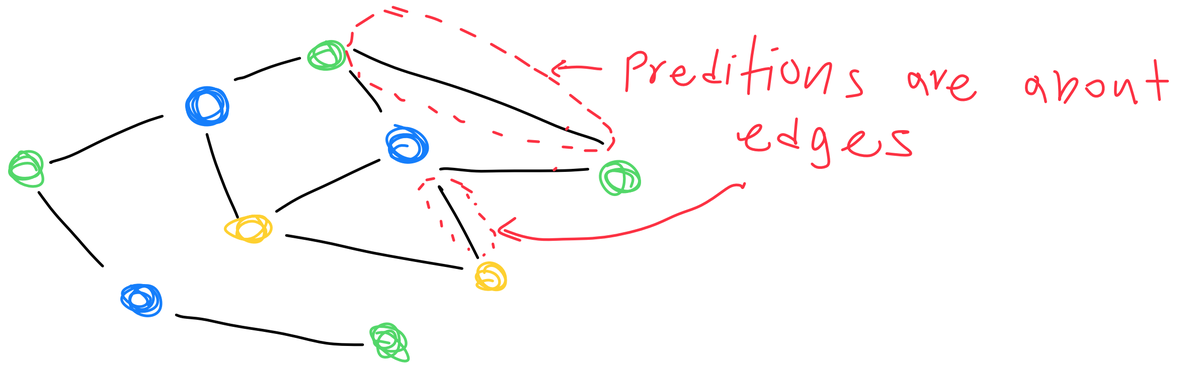


Task type	Output	Typical learning	Example(s)
Node classification	Label per node	Supervised	Cell type per cell (spatial graph); residue class in protein
Node regression	Value per node	Supervised	Risk score per IP; sensor-level rainfall forecast in station graph
Node ranking / importance	Score/rank per node	Unsupervised or Supervised	Influential users; essential genes/proteins
Node clustering / community detection	Cluster/community id	Unsupervised	Disease modules; social communities

Graphs: Introduction to the concept

What are potential predictions that we are interested of graphs? We can divide these intro 3 categories based on the level of the task:

B- Edge level tasks:

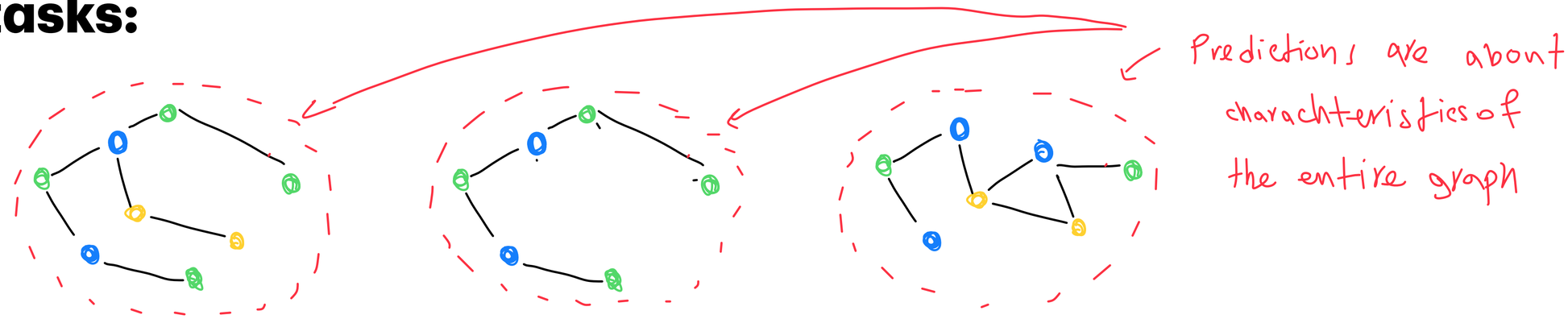


Task type	Output	Typical learning	Example(s)
Link prediction	edge exists for node pair	Supervised or Self-supervised	Predict missing friendships; predict drug–target links
Edge classification	Relation label per edge	Supervised	Synergistic vs antagonistic drug pairs; relation type in knowledge
Edge regression	Value per edge	Supervised	Interaction strength; binding affinity on drug–target edges
Temporal edge forecasting	Future edge/weight	Supervised	Future messages between users; evolving contacts in tissues

Graphs: Introduction to the concept and some use cases

What are potential predictions that we are interested of graphs? We can divide these intro 3 categories based on the level of the task:

C- Graph level tasks:

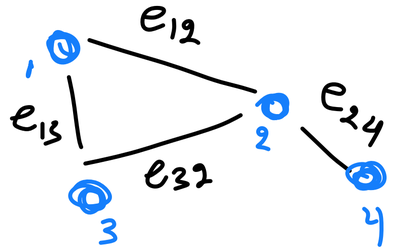


Task type	Output	Typical learning	Example(s)
Graph classification	Label per graph	Supervised	Molecule toxic/non-toxic; patient graph subtype
Graph regression	Value per graph	Supervised	Solubility; conductivity
Graph similarity / retrieval	Nearest graphs / similarity score	Self-supervised or Supervised	Find similar molecules; find similar patients
Graph generation	New graph structure	Self-supervised or RL	De novo molecular design

Graphs: Formal representation of graphs and some mathematical basis

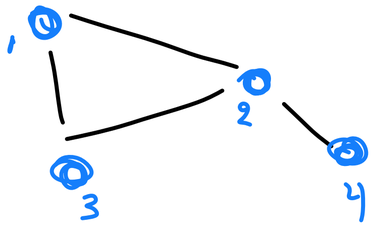
Formal representation of graphs

- Graph G, with a set of nodes V and edges E is presented as: **G(V,E)**
- There are two ways that we can represent the edges:
  - As a set of edges **{e(i,j)}**:



$$E = \{ e_{12}, e_{13}, e_{24}, e_{23} \}$$

  - Or via adjacency matrix **A**:

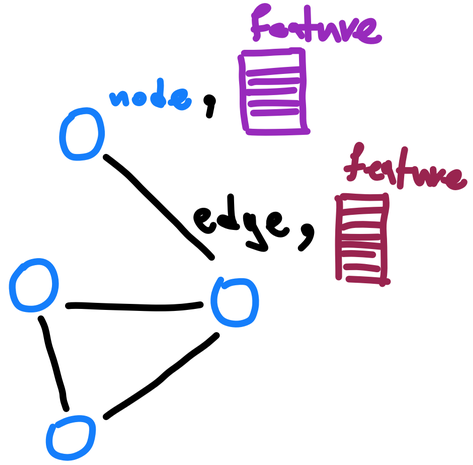


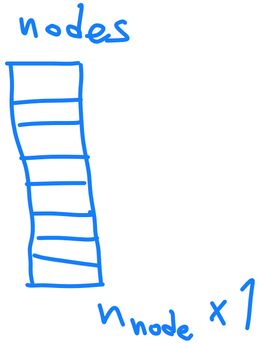
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

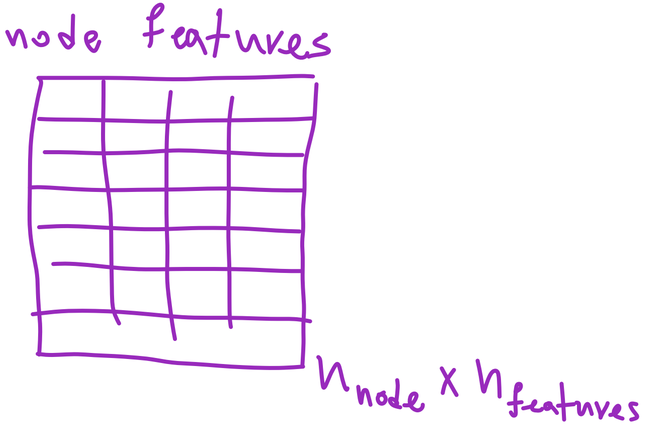
Graphs: about matrix representation of graphs

Formal representation of graphs:

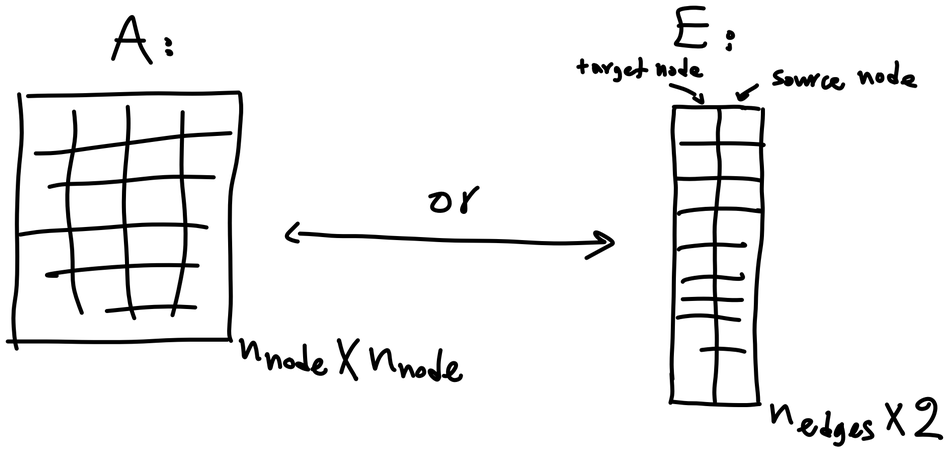
Dimension of tensors: Nodes, Edges, Node features, Edge features:

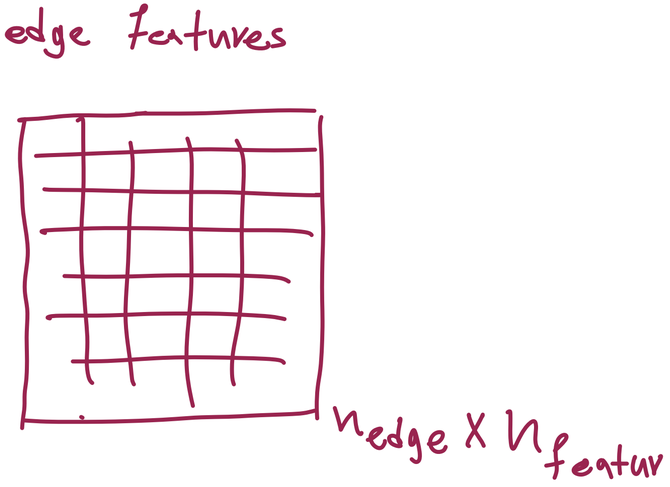






edges:

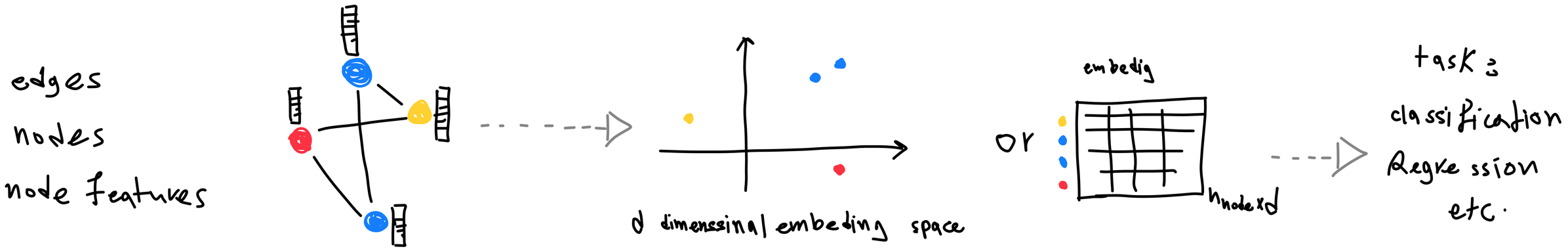




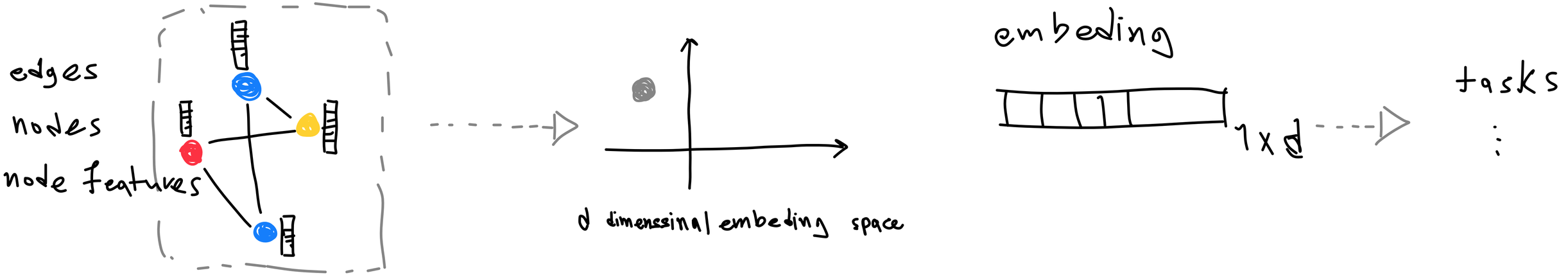
Graphs: The concept of embedding

Graph embedding at the node and graph level. Later these embedding can be used for downstream tastsk mentioned before.

1- **Node embedding**: Learn an embedded vector representation for each node.



2-**Graph embedding**: Learn an embedded vector representation for entire graph

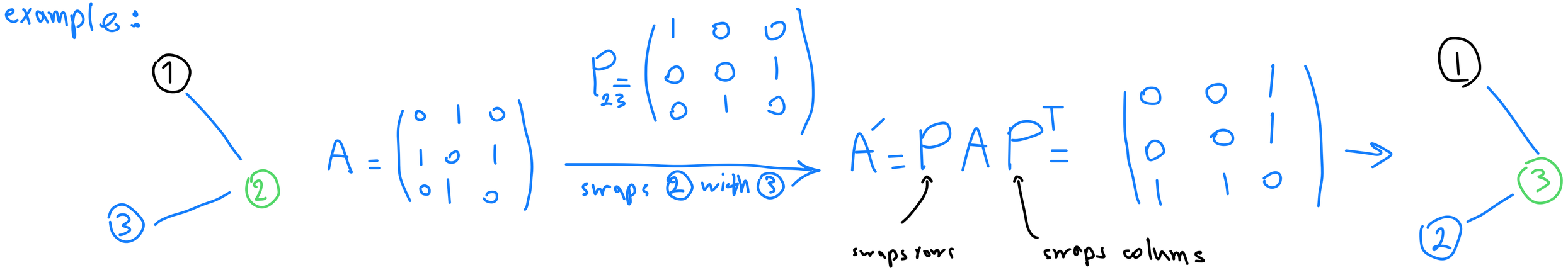


Graphs: symmetries in graphs: Permutation Invariance & Permutation equivariance

Permutation:

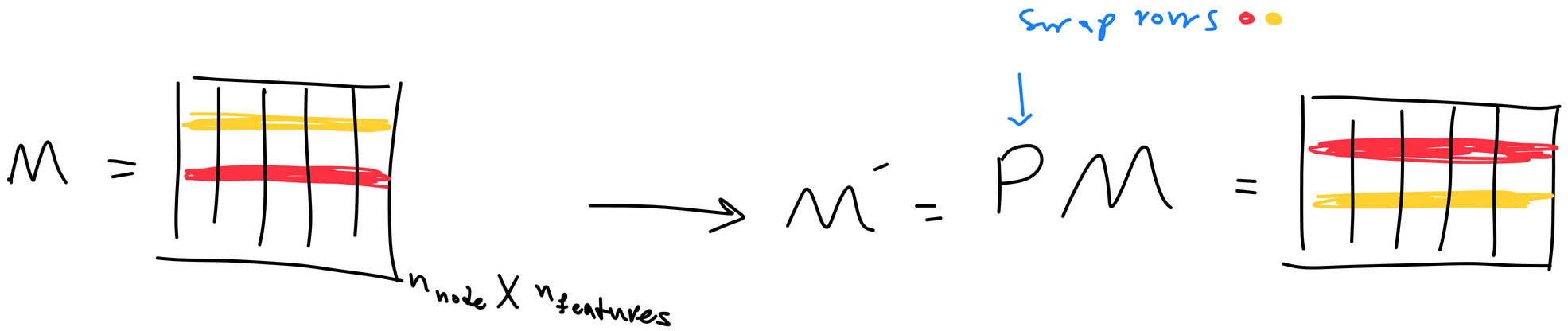
- The key concept to understand is Permutation (swapping the order) of nodes. A graph has no natural node order.
- A permutation just means relabeling/reordering nodes (and of course the associated features of nodes, etc).
- The *same* graph can be written with many different adjacency matrices depending on that order
- Mathematically Permutation operator (**P**) is doing this: $A_{new} = P A P^T$

example:



Graphs: symmetries in graphs: Permutation Invariance & Permutation equivariance

- Note that when we perform permutation, we should also apply it to node and edge features as well, since the order will be different.



Graphs: symmetries in graphs: Permutation Invariance & Permutation equivariance

Permutation Equivariance:

- A function g(G) is permutation-equivariant if reordering nodes reorders the output in the same way.
- Intuition: “if node labels move, the predictions move with them.”
- Example: Node classification: one label per node, if we reorder the nodes the prediction should also reorder.
- Mathematically:

$$g(PAP^T, PM) = P\, g(A, M)$$

Graphs: symmetries in graphs: Permutation Invariance & Permutation equivariance

Permutation Invariance:

- A function f(G) is permutation-inivariant if reordering nodes does not change the output.
- It is usually applied to the graph level predictions.
- Intuition: “if node labels move, the predictions does not change, as it is related to rthe whole graph.”
- Example: graph classification- is the graph representing a toxic or non-toxic molecule
- Mathematically:

$$f(PAP^T, PM) = f(A, M)$$

⏸ Pause Point

Test your knowledge:

- What is permeation invariance?
- Which level of tasks on graph should be permutation invariant?
- What is permutation equivariance?
- Which level of tasks on graph should be permutation equivaraint?

GNNs: Background

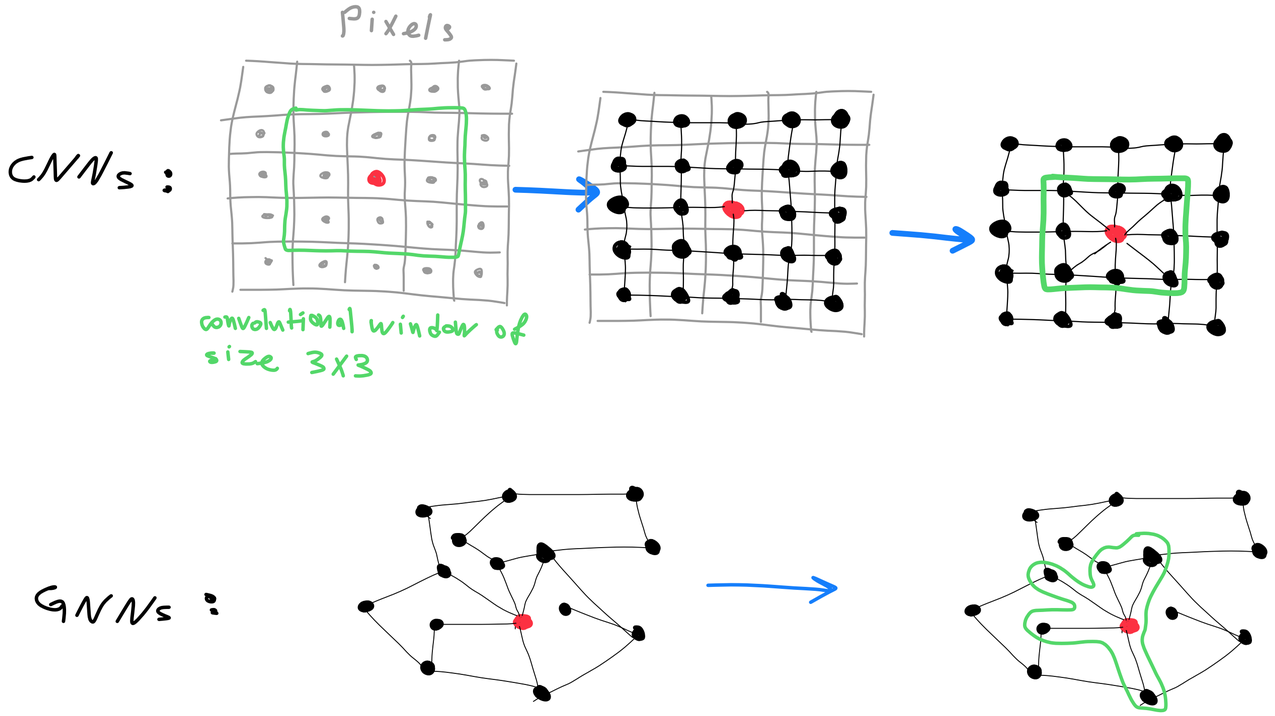
- Graph Neural Networks (GNNs) are specialized set of neural networks that are capable of learning functions (embedings) from graphs to perform specific tasks.
- The information inputs are **nodes**, **edges**, and **node features** and **edge features**.
- GNNs are Specialized to graph based data: GNNs are following a set of 'inductive biases' that makes them very efficient and powerful when dealing with graphs and tasks related to them

We have already seen the power of inductive biases, when dealing with a specific data structure and symmetries.

What are symmetries that should be present when dealing with graphs?

GNNs: Background, starting from CNNs

- We have learned about CNNs before, let's reframe image pixels as gaphs, and see how CNN is similar and different a GNN (we will define GNNs later in more details):



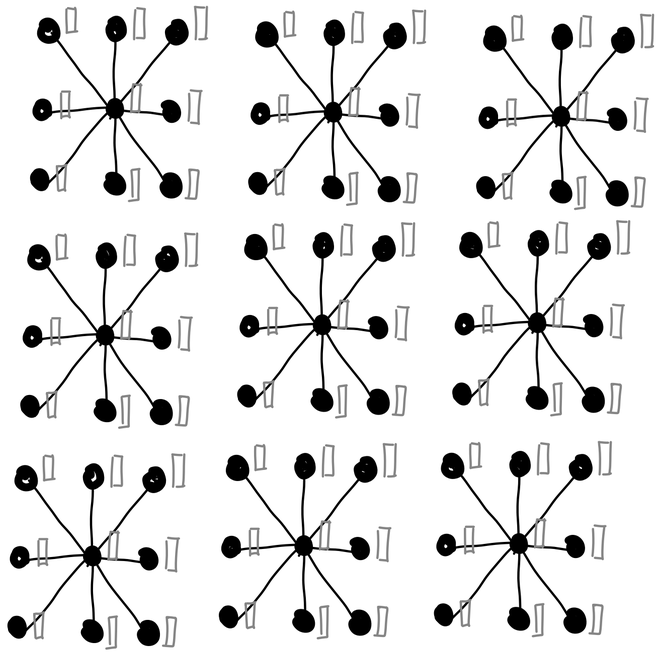
- Similarities: Local operation, globalized via depth, weight sharing
- Difference: Neighbourhood is not regular

- CNNs are basically GNNs on a grid! GNNs are more generalizable!

⏸ Pause Point

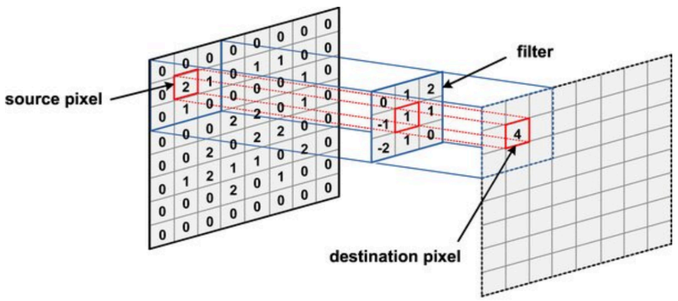
We have formed a GNN over the 9x9 image bellow based on the specific graph structure that we made.

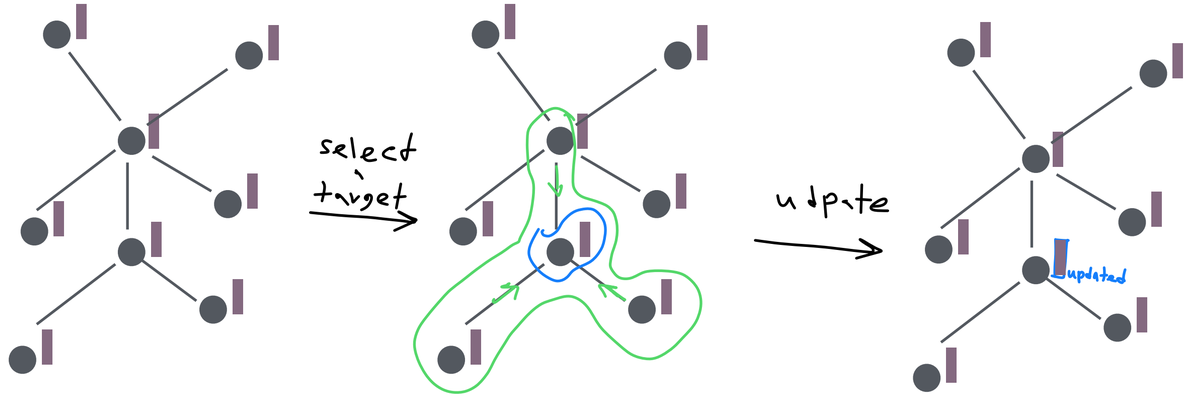
Our GNN is equivalent to a CNN with which Kernel size and which Stride size?



a. Kernel: 2x2 Stride: 3
b. Kernel: 3x3 Stride: 2
c. Kernel: 4x4 Stride: 4
d. Kernel: 3x3 Stride: 3

GNNs: Introduction to the ideas behind GNNs





In CNNs each pass is:

Each pixel has a feature vector (R,G,B)

Convolution centered at that pixel will aggregate some information from it's neighbourhood.

Then update the value of the pixel (in the centre) accordingly

Iterate over all pixels.

In GNNs we have message passing:

Each node has a embedding vector.

Calculate the message that each node sends to the target node.

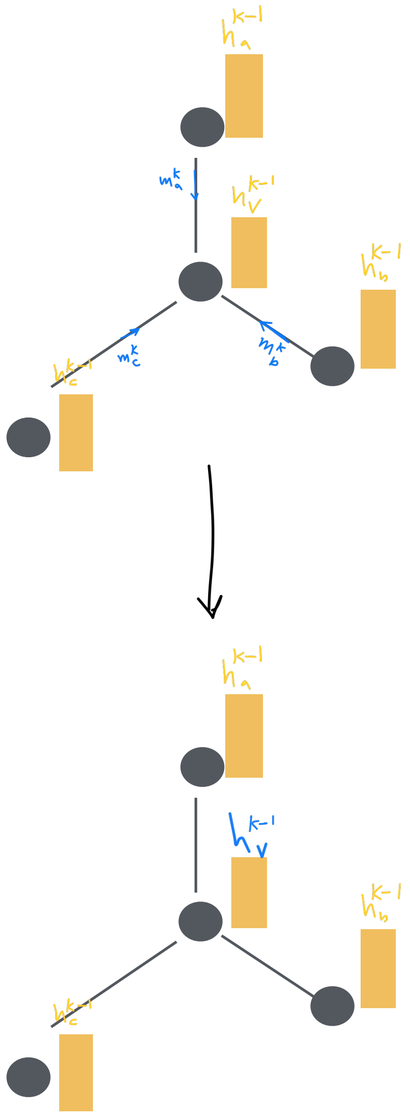
-Aggregate all the messages.

-Then update the embedding of the target node.

Iterate over all nodes

GNNs: Formalizing One round of Message passing:

In each round (**k**) of message passing:



For each node v, that has neighbours N(v):

1-Calculate the **individual messages** from all neighbours individually.

$$m_u^k = \text{Message}\left(h_u^{k-1}\right), \quad \forall u \in N(v)$$

2-**Aggregate** the messages of all neighbours

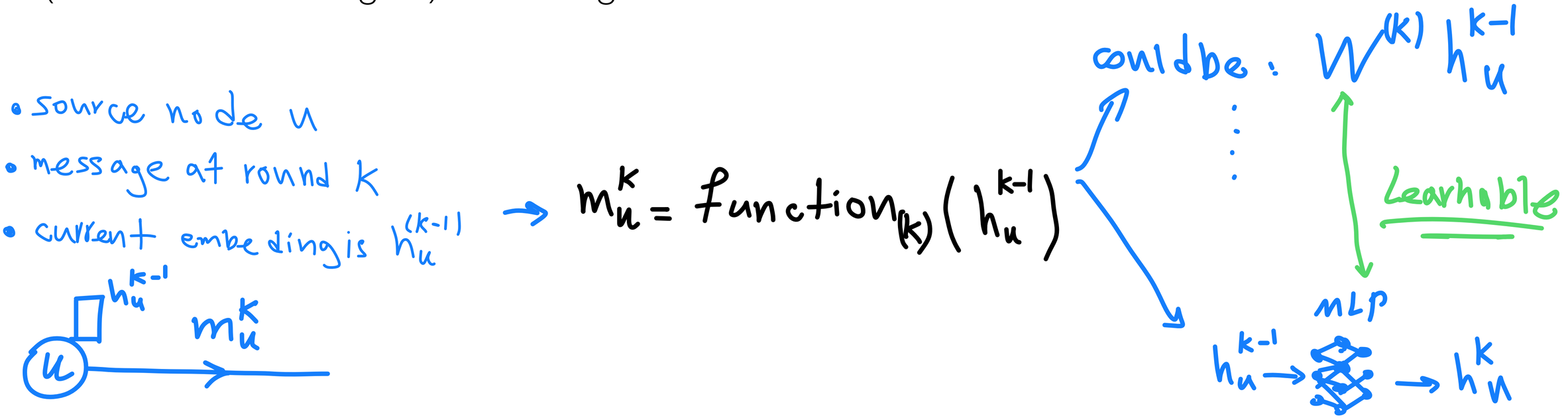
$$m_{N(v)}^k = \text{AGG}^k\left(\left\{m_u^k : u \in N(v)\right\}\right)$$

3- **Update** the embedding of the target node based on the aggregated messages and its own embedding

$$h_v^k = \text{Update}\left(h_v^{k-1}, m_{N(v)}^k\right)$$

GNNs: function: individual Message functions

- The message function, calculates the message that each node (u) wants to pass along. In other words they perform '*transformation*'
- At the **k** th round , the message function takes in the embedding of node u from the previous round (k-1) and convert it into a message the will be sent.
- The message function could be as simple as a linear transformation of the embedding to complex MLPs.
- Note that the message function at round k, should "*not necessarily*" share weights (have the same weights) of message function of round k'.



GNNs: function: Aggregation functions

- The aggregation function, aggregated the messages that all nodes in the neighbourhood of v, N(v) send to the node v.
- At the **k** th round , the aggregation function takes in the messages and generates a final message that should be passed to the node v.
- Aggregation should be **permutation invariant**. This is because the order of the neighbouring nodes should not matter.

$$m_{N(v)}^{(k)} = \text{AGGREGATE}(\{ m_u^k ; u \in N(v) \})$$

Permutation invariant

GNNs: function: Aggregation functions

- There are multiple generic options for Aggregation function.
- It just need to be a set function (permutation invariant)

Sum: $m_{N(v)}^{(k)} = \sum_{u \in N(v)} m_u^k$

Normalized sum: $m_{N(v)}^k = \sum_{u \in N(v)} \frac{m_u^k}{\sqrt{|N(v)||N(u)|}}$

Mean: $m_{N(v)}^{(k)} = \sum_{u \in N(v)} \frac{m_u^k}{|N(v)|}$!! → this washes off information about how many neighbours...

Min/Max: $m_{N(v)}^k = \min\{ m_u^k ; u \in N(v) \}$

- **The most general form (universal approximator of a set function) is based on a specific combination of MLPs as bellow:**

Zaheer et al., 2017 — Deep Sets
Qi et al., 2017 — PointNet
Xu et al., 2019 — GIN (Graph Isomorphism Network)

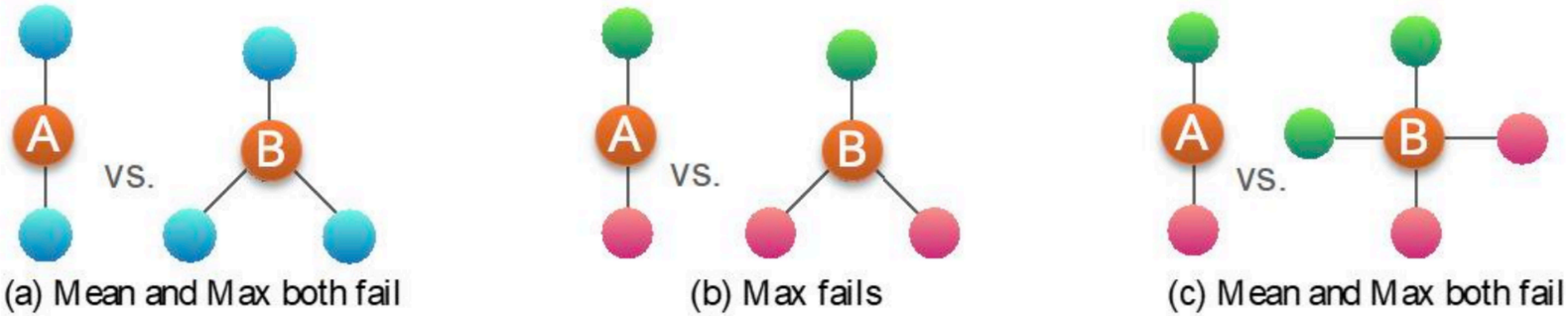
Multi-set MLP: $m_{N(v)}^k = MLP_2^{(k)}\left(\sum_{u \in N(v)} MLP_1^{(k)}(m_u^k, m_v^k)\right)$

GNNs: function: Aggregation functions

- In terms of they discriminative power, we can rank AGGREGATION functions as bellow:

Multi-set MLP >> Sum > Mean > Min/Max

- Mean and Max/Min pooling are not able to distinguish many cases, e.g.:



(a) Mean and Max both fail (b) Max fails (c) Mean and Max both fail

From Jure Leskovec, Stanford University

GNNs: function: Update function

- At update round k, The update function, just takes in the final aggregated message from the aggregation step, along with the target node's embedding.
- Then it will calculate the next updated embedding of node **v** accordingly:

$$h_v^k = \text{Update}^{(k)}(h_v^{k-1}, m_{N(v)}^k)$$

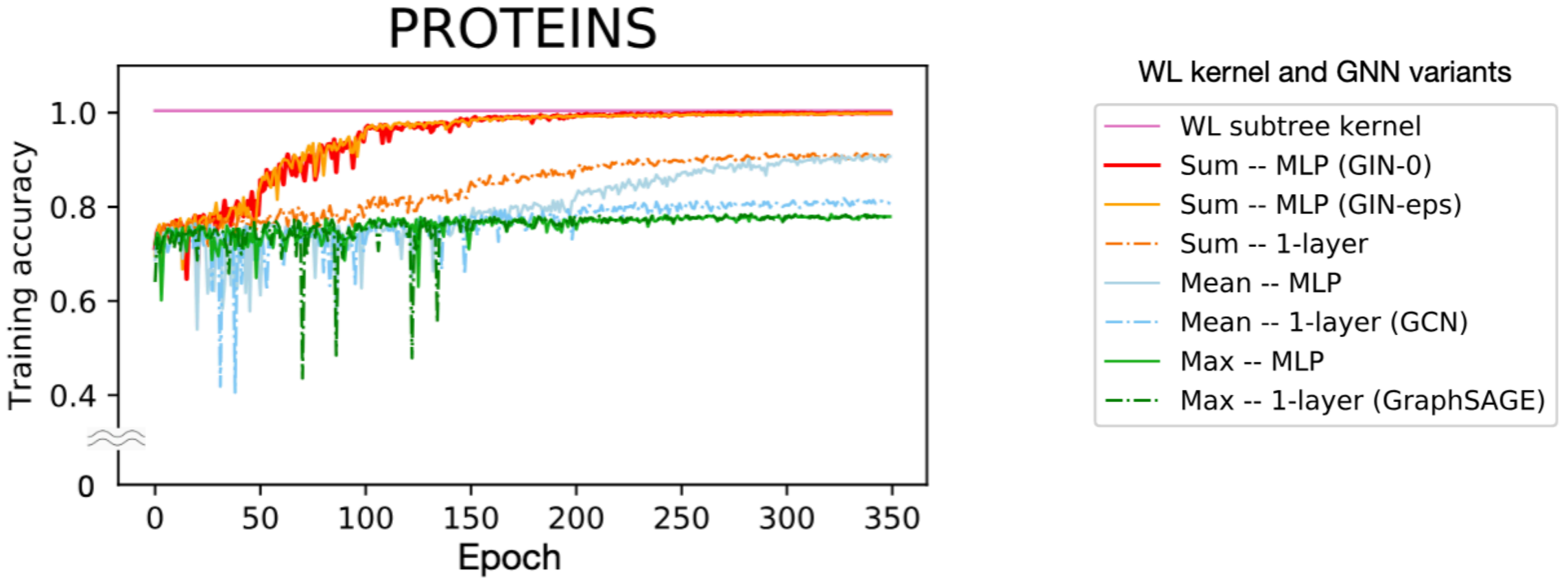
- In the context of GNNs, the update function is usually a MLP, with learnable parameters.

$$h_v^k = \text{MLP}^{(k)}(h_v^{k-1}, m_{N(v)}^k)$$

Update:

$$\mathbf{h}_v^{(k)} = \sigma(\mathbf{W}_{\text{self}}\mathbf{h}_v^{(k-1)} + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(v)}^{(k)} + b)$$

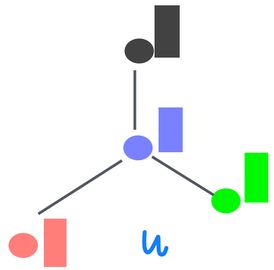
GNNs: function: Aggregation function in practice!

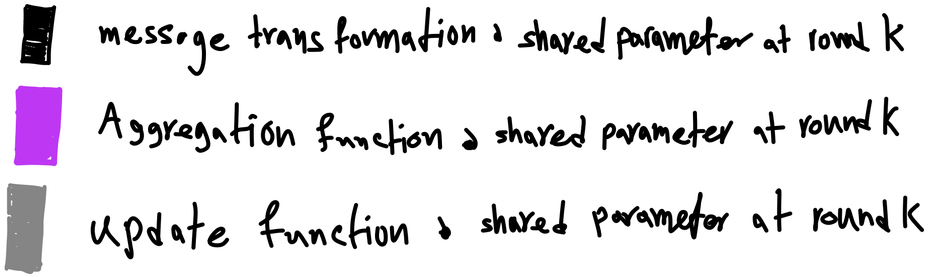


Xu, Keyulu, et al. "How powerful are graph neural networks?." *arXiv preprint arXiv:1810.00826* (2018).

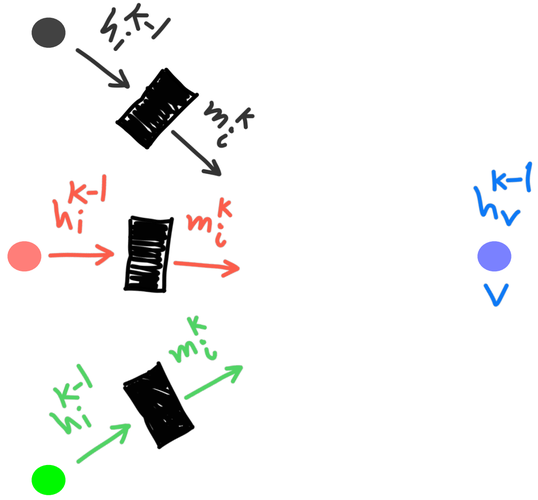
GNNs: Putting Message transformation, Aggregation and update together

- Putting all together, one round of message passing can be visualized as bellow:



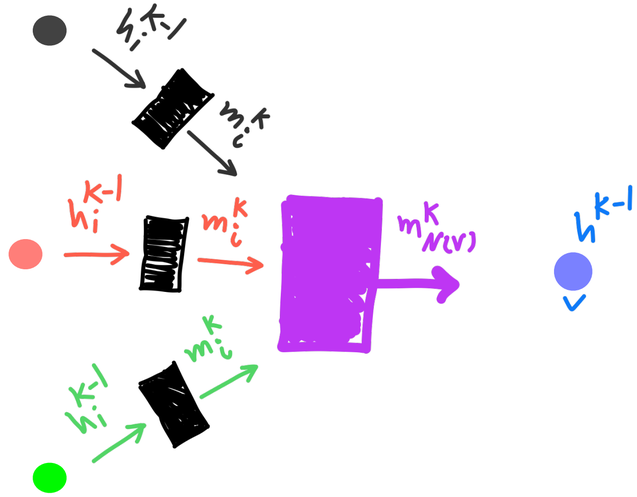


1.Message Transformation



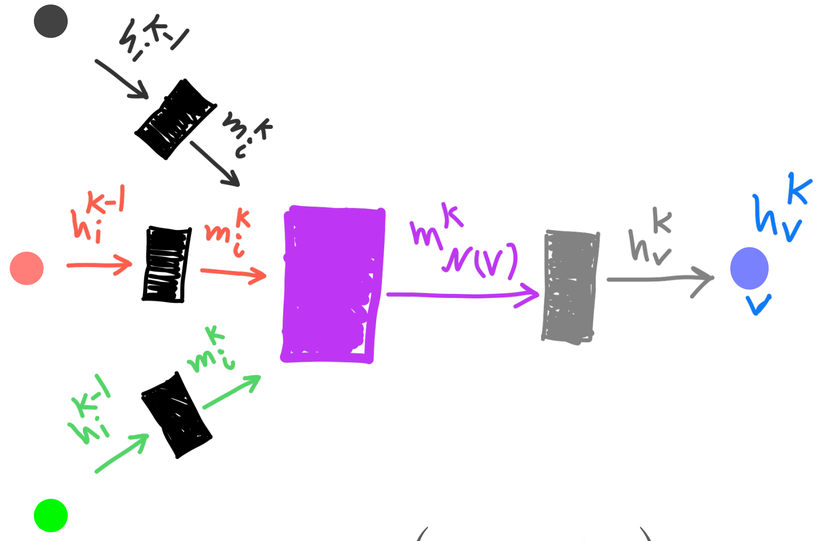
$$m_u^k = \text{Message}\left(h_u^{k-1}\right), \quad \forall u \in N(v)$$

2.Aggregation



$$m_{N(v)}^k = \text{AGG}^k\left(\left\{m_u^k : u \in N(v)\right\}\right)$$

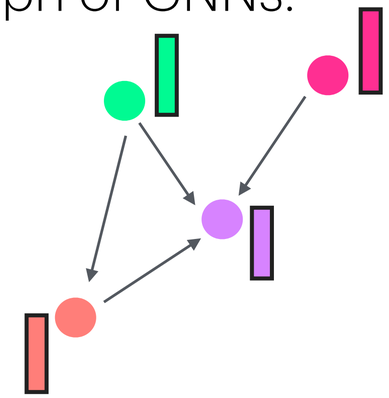
3.Update



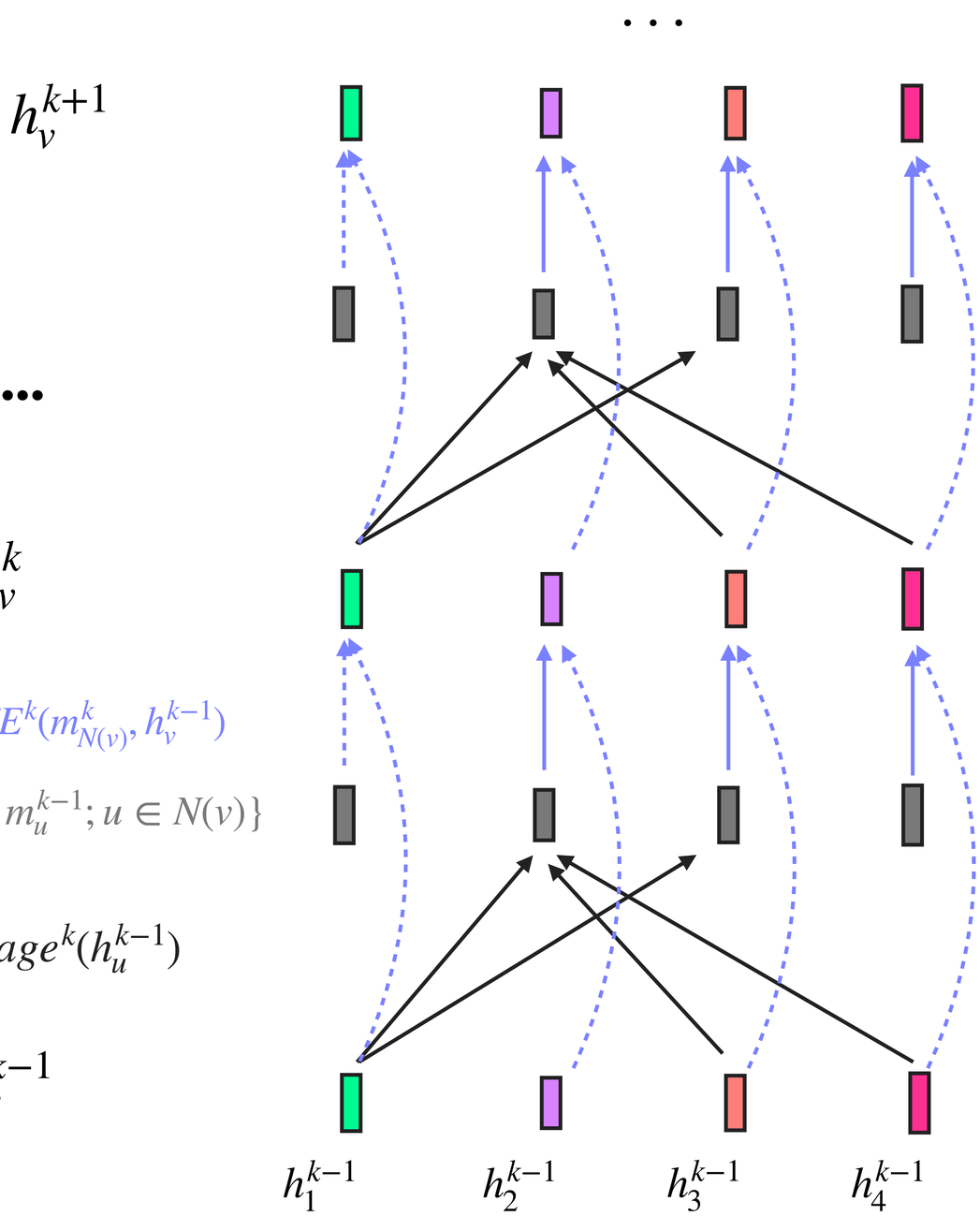
$$h_v^k = \text{Update}\left(h_v^{k-1}, m_{N(v)}^k\right)$$

GNNs: the computation graph of GNNs:

- Computation graph is a key feature: It is important to be able to draw and understand the computation graph of GNNs.



- It is like MLP but:
  - Nodes are vectors (h_v^k), not scalars
  - Edges are complex MLPs: $m_u^k = Message^k(h_u^{k-1})$
- Each GNN message passing is a layer
  - $m_{N(v)}^k = AGG^k\{m_u^{k-1}; u \in N(v)\}$
    is akin to a linear layer
  - Update $h_v^k = UPDATE^k(m_{N(v)}^k, h_v^{k-1})$
    is akin to a point wise layer



GNNs: the computation graph of GNNs:

Important note on learning settings: One can choose that the computational functions of GNNs which are $Message^k$,AGG^k, and $UPDATE^k$ to share parameters across the depth (k values) or not. But:

A. If we allow each k has it's own parameters, the computation becomes more powerful, with the cost of more parameters and potential overfitting

B. If we share parameters across depth ($Message = Message^k$,$AGG = AGG^k$, and $UPDATE = UPDATE^k$), we should make sure that the model converges after iterations of depth.

Edge attribute: One can also assign edge feature if needed, it will change the form of message fucntion only

Hand wavy rule of thumbs: If we have more depth (K), we can get away with less parameter in each layer, but if we have a shallow GNN (e.g K=2), we need more powerful functions

How decide the depth value: look at the receptive field of your problem, and adjust the k accordingly.

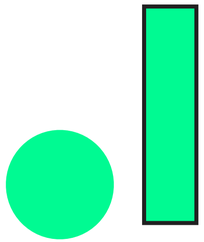
Pause Point

In the language that we used to draw the computation graph of GNNs, what would be the computation graph of a simple MLP?

A- same as GNN

B-same as GNN but with one one

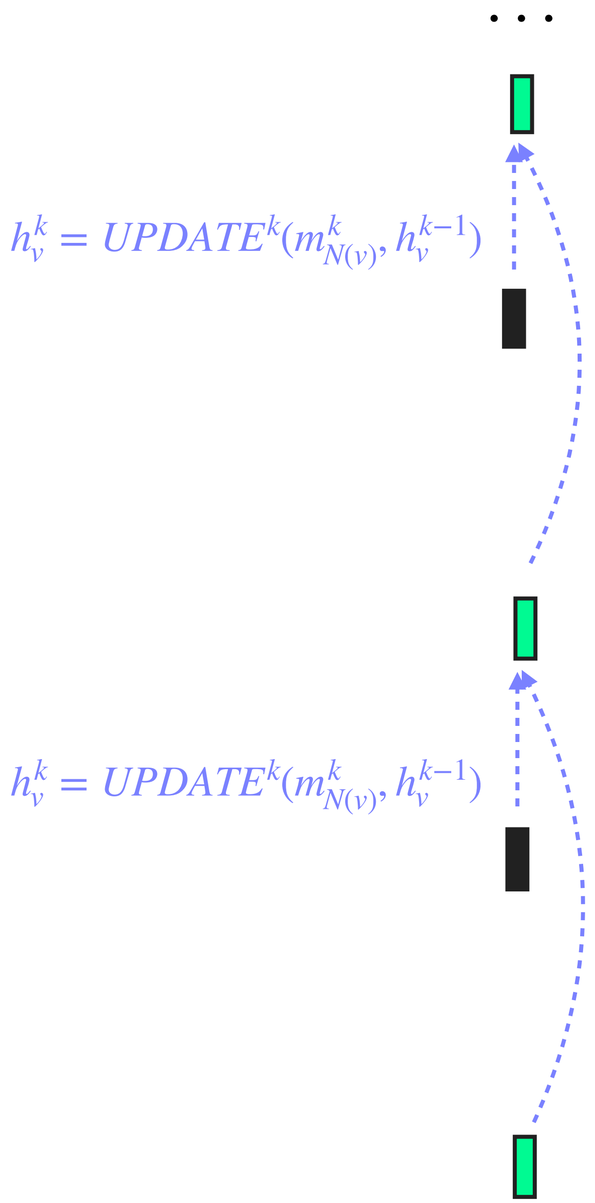
C- only a point



Answer:

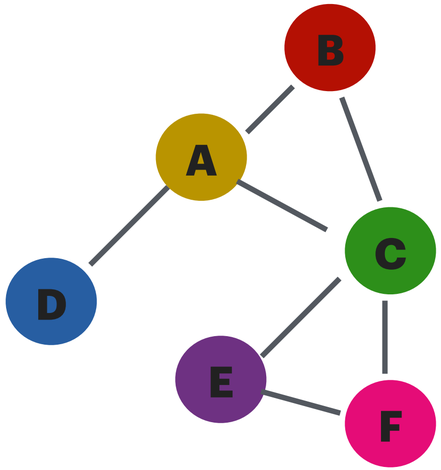
Update:

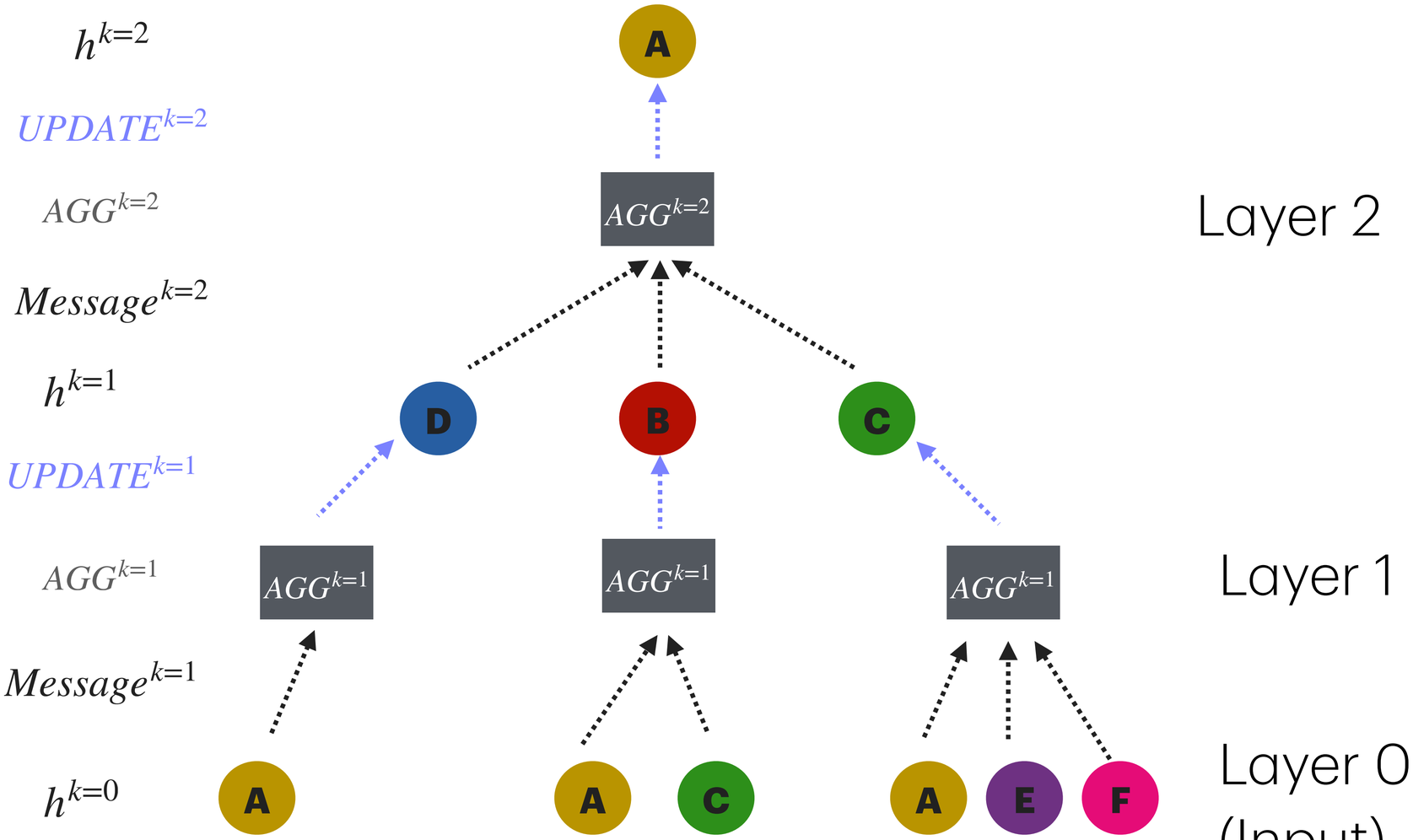
$$\mathbf{h}_v^{(k)} = \sigma(\mathbf{W}_{\text{self}}\mathbf{h}_v^{(k-1)} + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(v)}^{(k)} + b)$$



GNNs: the computation graph of GNNs: Tree view

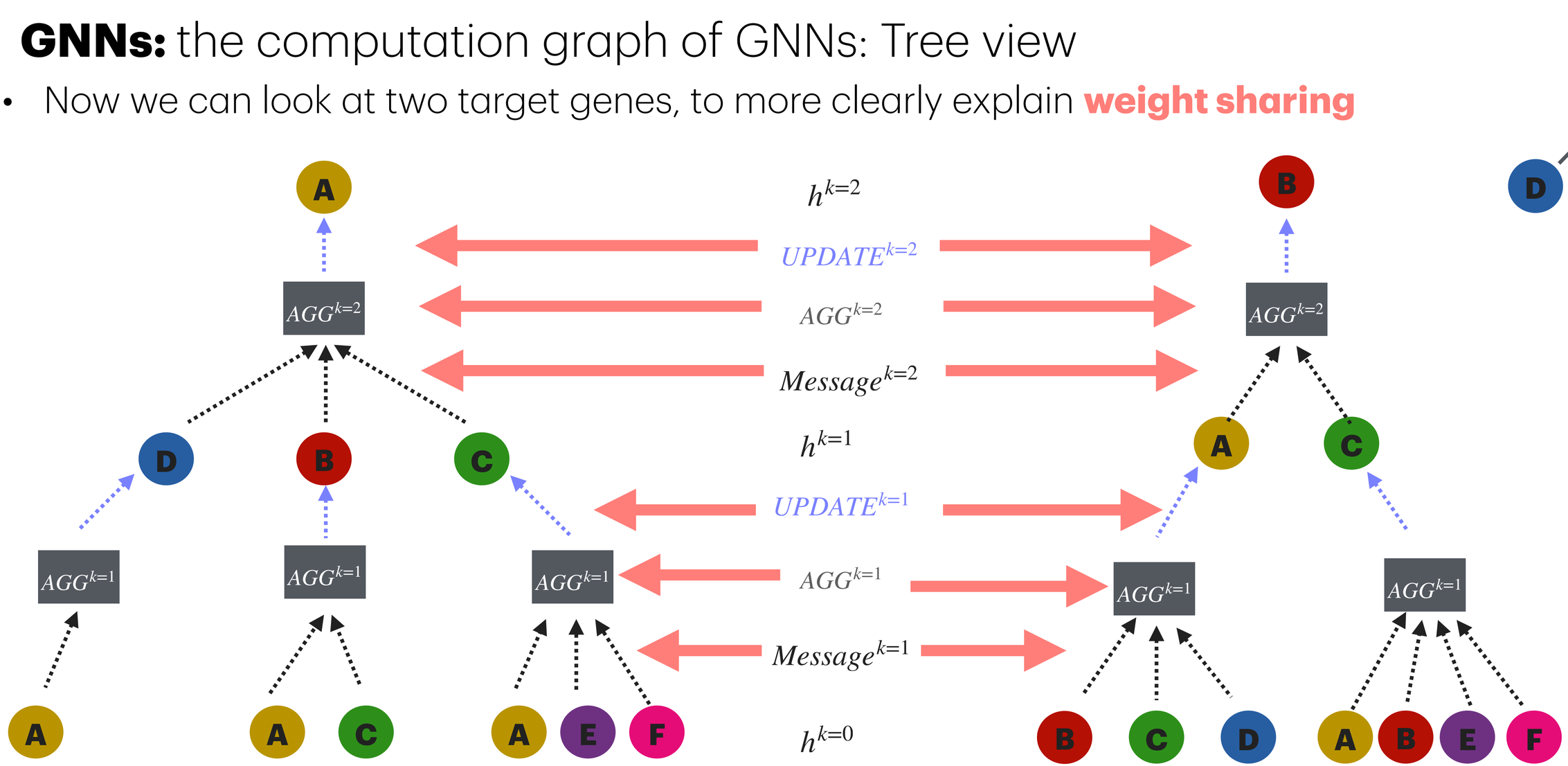
- Another ways of looking at GNNs is how information flows inside the graph as the steps ‘k’ unroll.
- To understand this more clearly, we can draw the computation graph as a tree for one target node (**A**) at layer 2 (**k=2**).





GNNs: the computation graph of GNNs: Tree view

- Now we can look at two target genes, to more clearly explain **weight sharing**



- This is an important property, GNNs can be **inductive:** "learn a rule that can be applied to new graphs too". Because the weights that are learned, are shared between all nodes. Now if we have a graph with different nodes or edges (of course same type of data), the learned GNN can work on it. So we can train on a part of graph and predict the rest etc.

GNNs: Summary

- Encodes graph structure together with node and edge features
- Important property: permutation invariance / equivariance
- Core idea: message passing and aggregation
- Can handle graphs with different sizes and structures, similar to how CNNs handle images
- Connected to graph signal processing, graphical models, distributed computing, and graph isomorphism testing
- Representation can be improved with higher-order structure, node IDs, and augmentation

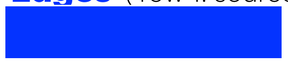
Coding the Most General GNN model in PyG:

A) Preprocessing inputs to GNN algorithms

NodesFeatures

$N_{node} \times N_{NodeFeatures}$

Edges (row 1: source row 2 target)



$2 \times N_{Edge}$

EdgeFeatures

$N_{Edge} \times N_{EdgeFeatures}$

B) PytorchGeometric makes these internally for message passing:

x_j (Source Node Features On Edges)

$N_{Edge} \times N_{NodeFeatures}$

x_i (Target Node Features On Edges)

$N_{Edge} \times N_{NodeFeatures}$

C)propagation

$Message$ (**x_i**,**x_j**, **EdgeFeatures**)

Messages

$N_{Edge} \times N_{MessageFeatures}$

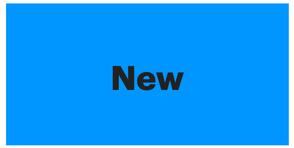
AGG(**Messages, Edges[1,:]**)

Agg_Messages

$N_{Node} \times N_{MessageFeatures}$

$UPDATE_{edge}$(**x_i**,**x_j**, **EdgeFeatures**)

EdgeFeatures



$N_{Edge} \times N_{EdgeFeatures}$

$UPDATE$(**NodesFeatures**, **Agg_Messages**)

NodesFeatures

$N_{node} \times N_{NodeFeatures}$

D) Post processing

Global State (**NodesFeatures, Edges, EdgeFeatures**)

Global State

$N_{GraphFeatures}$

You should only specify:

Tensors:
NodesFeatures
Edges
EdgeFeatures (Optional)

Functions:
$Message$
AGG
$UPDATE_{edge}$(Optional)
$UPDATE$
$GlobalState$(Optional)

Coding the Most General GNN model in PyG: Example Flavors

You should only specify:

Tensors:
NodesFeatures
Edges
EdgeFeatures (Optional)

Functions:

$$m_{u\to v}^k = \text{Message}^k\left(h_v^{k-1}, h_u^{k-1}, e_{uv}\right)$$

$$m_{N(v)}^k = \text{AGG}^k\left(\{\, m_{u\to v}^k : u \in N(v)\,\}\right)$$

$$h_v^k = \text{Update}^k\left(h_v^{k-1}, m_{N(v)}^k\right)$$

Optional:
$GlobalState$(Optional)
$UPDATE_{edge}$(Optional)

Graph Convolutional Networks (GCN)

Key Idea: It's just a weighted sum of neighbors.

$$m_{u\to v}^k = \frac{1}{\sqrt{\hat{d}_u \hat{d}_v}} W^k h_u^{k-1}$$

$$m_{N(v)}^k = \sum_{u\in N(v)\cup\{v\}} m_{u\to v}^k$$

$$h_v^k = \sigma\left(m_{N(v)}^k\right)$$

Graph Attentional Network (GAT)

Key Idea: Learns an attention weight for each neighbor, so neighbors contribute with different importance.

$$m_{u\to v}^k = \alpha_{u\to v}^k(h_v^{k-1}, h_u^{k-1})\,\text{MLP}^k\left(h_u^{k-1}\right)$$

**Where $a^k(h_v^{k-1}, h_u^{k-1})$ is normalized attention of u to v*

$$m_{N(v)}^k = \sum_{u\in N(v)\cup\{v\}} m_{u\to v}^k$$

$$h_v^k = \sigma\left(m_{N(v)}^k\right)$$

* will learn more about attention later in the course

Message Passing Neural Network (MPNN)

Key Idea: Uses heavy-duty neural networks (MLPs or RNNs) for the functions instead of simple linear layers.

$$m_{u\to v}^k = \text{MLP}^k\left(h_v^{k-1}, h_u^{k-1}, e_{uv}\right)$$

$$m_{N(v)}^k = \sum_{u\in N(v)} m_{u\to v}^k$$

(Mean/Max/or LSTM)

$$h_v^k = \text{Update}^k\left(h_v^{k-1}, m_{N(v)}^k\right)$$

(Update can be GRU, MLP, or another learnable node-wise function)

Graph Convolutional Networks (GCN)

$$m_{u\to v}^k = \frac{1}{\sqrt{\hat{d}_u \hat{d}_v}} W^k h_u^{k-1}$$

$$m_{N(v)}^k = \sum_{u \in N(v) \cup \{v\}} m_{u\to v}^k$$

$$h_v^k = \sigma\left(m_{N(v)}^k\right)$$

Graph Attentional Network (GAT)

$$m_{u\to v}^k = \alpha_{u\to v}^k (h_v^{k-1}, h_u^{k-1})\, \mathrm{MLP}^k\left(h_u^{k-1}\right)$$

$$m_{N(v)}^k = \sum_{u \in N(v) \cup \{v\}} m_{u\to v}^k$$

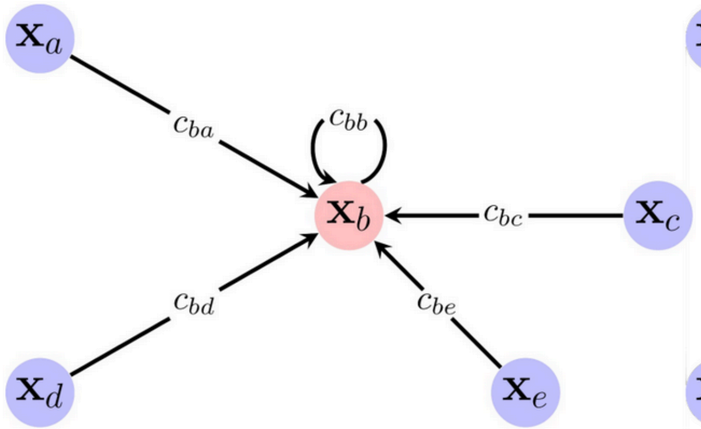
$$h_v^k = \sigma\left(m_{N(v)}^k\right)$$

Message Passing Neural Network (MPNN)

$$m_{u\to v}^k = \mathrm{MLP}^k\left(h_v^{k-1}, h_u^{k-1}, e_{uv}\right)$$

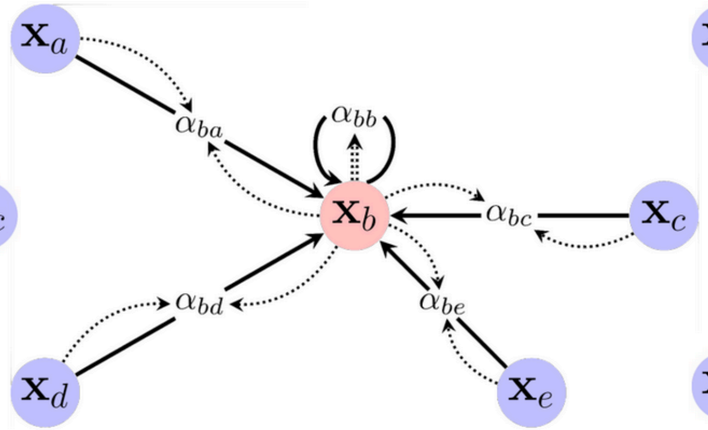
$$m_{N(v)}^k = \sum_{u \in N(v)} m_{u\to v}^k$$

$$h_v^k = \mathrm{Update}^k\left(h_v^{k-1}, m_{N(v)}^k\right)$$



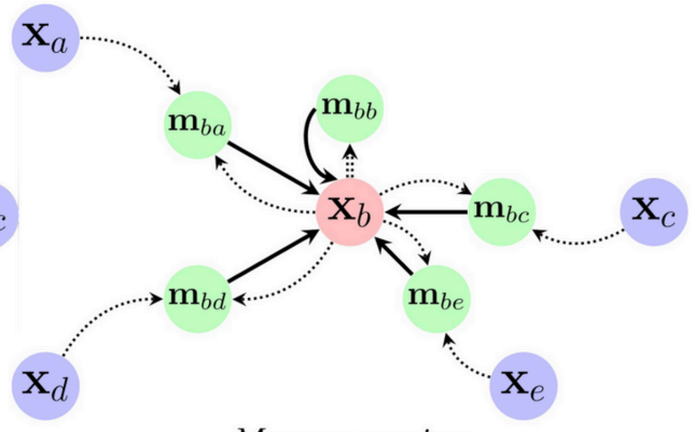
Convolutional

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



Attentional

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



Message-passing

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

Bronstein, Michael M., et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges." *arXiv preprint arXiv:2104.13478* (2021).

Recipe Card: **GCNs**

Type of Learning: Supervised
Data set: Graph: nodes, edges,node features

Data

1- Load libraries: Basics + PyTorch + PyG (pytorch_geometric)

2-Load your graph Dataset **D.** Sometimes D has only one graph (data):

-Make sure each element of $D = [data_1, \ldots data_K]$, $data_i$ is a PyG Data object:

-$data_i \,.\, x$ = NodeFeatures, shape: $[N_{node}, N_{nodeFeatures}]$

-$data_i \,.\, edge_index$= Edges, shape: $[2, N_{edge}]$

-$data_i \,.\, y$= target, shape for graph level: $[N_{target}]$, for node level: $[N_{node}, N_{target}]$

3-Now set up the DataLoader for training/validation/testing: (This depends on the task)

a) If you have many graphs:

-split the dataset into: train_data validate_data test_data.

-then load each split using PyG DataLoader. PyG will batch graphs automatically

b) If you have one big graph for node classification/regression

-keep one Data object then define train_mask val_mask test_mask.

-if these masks are not provided in the dataset, they can be made by RandomNodeSplit function in PyG.

NN model

4-Building the GCN neural network

-GCNs have two main parts. GCN layers and Task heads.

A)define GCNs layers as a module [GCN]: (x is node features and x' is updated node features)

$$Input: \ (x, edge\_index) \rightarrow [(GCNConv) \rightarrow (Relu)] \rightarrow \ldots \rightarrow [(GCNConv) \rightarrow (Relu)] \rightarrow x'$$

-B) define task head as a module [Task Head]:

-For node-level tasks (one per node, could be classification, regression, embedding, etc):

$$x' \rightarrow [Linear\ layer] \rightarrow out, \hat{y}_v$$

-For graph-level tasks (one per graph, could be classification, regression, embedding, etc):

$$x' \rightarrow [GlobalMeanPool] \rightarrow [Linear\ layer] \rightarrow out, \hat{Y}$$

Training

5-Define a proper loss function based on Y

a-If Y is continuous, then use MSELoss or L1Loss

b-If Y is categorical, use CrossEntropyLoss. note that we feed raw logits directly to Loss.

i-For node-level tasks, calculate loss on the selected nodes only (usually via train_mask)

ii-For graph-level tasks, calculate loss on the graph outputs $\hat{Y}$

-Add regularization to prevent overfitting

-The loss will be the sum of all relevant losses

6-Define the train and test functions:

-Train/Validate/test functions are standard

-For node-level tasks it masks for training/validation/testing.

-For graph-level tasks, the model gets batched graphs from PyG DataLoader

7-Define training and run the training:

-Same as standard.

-For each epoch: run train() on training data, then run test() on training/validation data

-At the end: load the best saved model and plot training/validation loss curves if needed

Recipe Card: **MPNN**

Type of Learning: Supervised
Data set: Graph: nodes, edges,node features

Data

1- Load libraries: Basics + PyTorch + PyG (pytorch_geometric)
2-Load your graph Dataset **D.** Sometimes D has only one graph (data)
-All similar to **GCNs**
3-Now set up the DataLoader for training/validation/testing: (This depends on the task)
-All similar to **GCNs**

Training

5-Define a proper loss function based on Y
-All similar to **GCNs**

6-Define the train and test functions:
-All similar to **GCNs**
7-Define training and run the training:
-All similar to **GCNs**

NN model

4-Building the MPNN neural network

-MPNNs have two main parts: Message Passing layers and Task Head.

A) define Message Passing layers as a module [MPNN]:

-In PyG, define each layer using MessagePassing.

-Inside one layer, forward() gets $(x, edge\_index, edge\_attr)$ as input then calls propagate(...).

-$propagate(\dots)$ starts message passing over all edges: it uses $message(\dots)$ to build edge-wise messages, $AGG(\dots)$ to combine incoming messages for each node, and $update(\dots)$ to produce the updated node features x' x '

-So Under $propagate(\dots)$, we define:

$$m_{u\to v}^{k} = Message^{k}(h_v^{k-1}, h_u^{k-1}, e_{uv}), \qquad m_{N(v)}^{k} = AGG^{k}(\{m_{u\to v}^{k} : u \in N(v)\}), \qquad h_v^{k} = Update^{k}(h_v^{k-1}, m_{N(v)}^{k})$$

-So overal a full [MPNN] has k layers (layer in sense of GNNs, which means k itterations) and [MPNN] module is like:

$$Input : (x, edge\_index, edge\_attr) \to [(MessagePassing) \to (Relu)] \to \dots \to [(MessagePassing) \to (Relu)] \to x'$$

-B) define task head as a module [Task Head]:

-all similar to GCN, once the x' is computed by [MPNN] module.