Graph Representation Learning

Tianxiang (Adam) Gao

November 11, 2024

Outline

Recap: Transformers

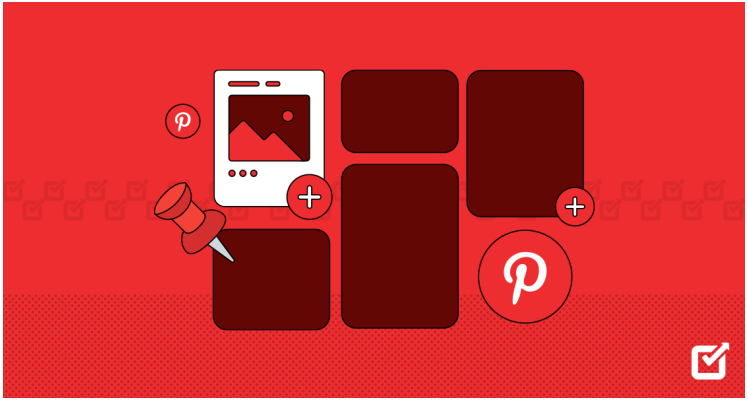
- **Self-Attention**: Refines the representation of each token by learning its relevance to all other tokens, *i.e.*, $\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{v}_i$, where α_i represents attention weights.
- **Multi-Head Attention**: Focuses on different aspects of each token to capture diverse patterns, *i.e.*, $\boldsymbol{z} = [\boldsymbol{z}_1 \ \ldots \ \boldsymbol{z}_H]\boldsymbol{W}_o$, where each $\boldsymbol{z}_h$ represents an individual attention head.
- **Layer Normalization**: Normalizes each layer by computing statistics across the hidden units within a layer.
- **Encoder-Decoder Attention**: Refines the output representation by referencing the input representations.
- **Masked Attention**: Masks future tokens to maintain autoregressive generation, preventing "leakage" of future information.
- **Positional Encoding**: Provides unique, low-dimensional representations to encode token positions, allowing the model to differentiate positional relationships easily.
- **Teacher Forcing**: Uses the correct prior output during the training to facilitate learning.

Recap: Large Language Models

- **BERT**: An encoder-only architecture that utilizes both past and future context in bidirectional self-attention.
- **Masked Language Modeling**: BERT is pretrained by predicting masked tokens based on surrounding context.
- **GPT**: A decoder-only architecture, pretrained as a standard language model that predicts the next token in a sequence.
- **Zero-Shot Inference**: The pretrained model performs inference by prompting with a task description without fine-tuning.
- **In-Context Learning**: The model learns to perform tasks by providing demonstrations before posing the question.
- **Neural Scaling Laws**: Predicts computational efficiency gains as model size increases, showing systematic improvement with model scaling.

Outline

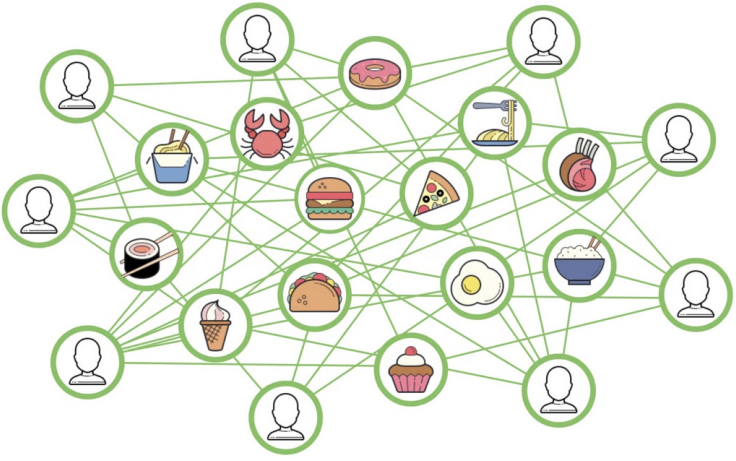
Recommendation System: Pinterest



- Pinterest uses Graph Convolutional Networks (GCNs) to enhance recommendations by modeling user-item interactions as a bipartite graph, allowing the system to capture complex user preferences through aggregated neighbor information.

Source: Ying et al., "Graph Convolutional Neural Networks for Web-Scale Recommender Systems." KDD 2018

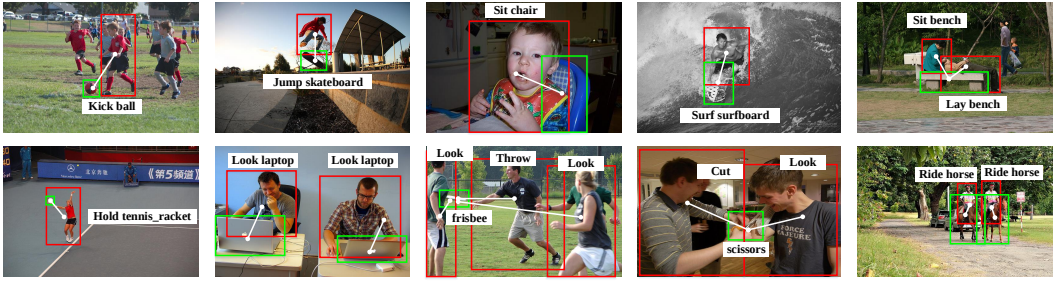
Recommendation System: Uber Eats



- Uber Eats employs Graph Neural Networks (GNNs) to enhance its recommendation system by modeling the relationships between users, restaurants, and dishes as a graph, enabling the platform to provide more personalized and relevant food and restaurant recommendations to users.

Uber Blog: Food Discovery with Uber Eats

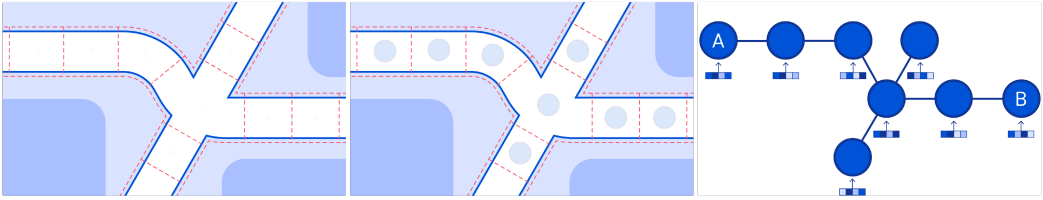
Human-Object Interaction



- This paper introduces the Graph Parsing Neural Network (GPNN), which uses message passing to recognize human-object interactions in images and videos by dynamically constructing a parse graph that captures complex relationships between humans and objects.

Qi et al., "Learning human-object interactions by graph parsing neural networks," ECCV 2018.

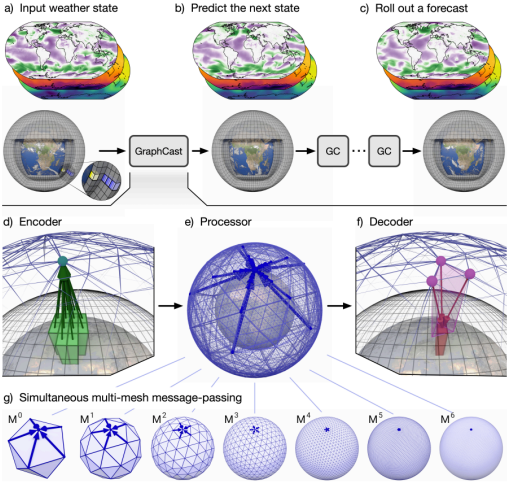
Estimated Time of Arrival (ETA) Prediction with Graph Neural Networks



- Google Maps employs Graph Neural Networks (GNNs) to model road networks as graphs, using message passing to integrate spatial connections and real-time traffic data, enhancing ETA predictions on complex routes.

Derrow-Pinion et al., "ETA Prediction with Graph Neural Networks in Google Maps," CIKM 2021

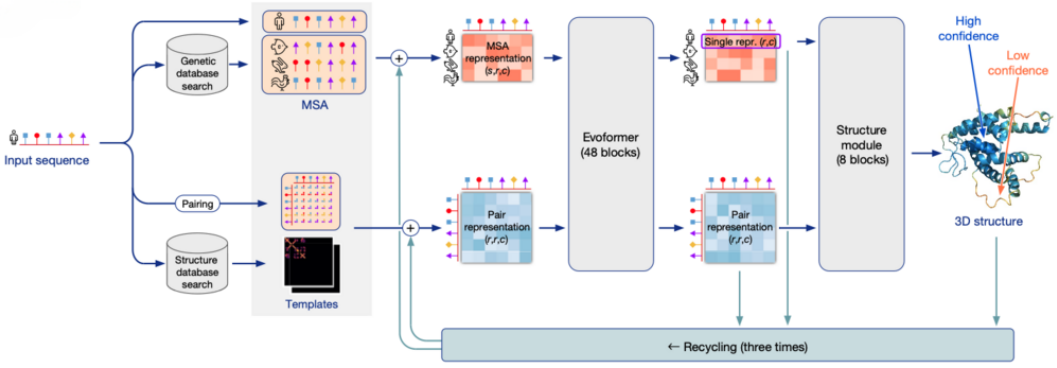
GraphCast: Weather Forecasting



- Google DeepMind introduced GraphCast, a GNN-based weather model with an Encoder-Processor-Decoder architecture that captures global spatial dependencies for improved medium-range forecasts.

Lam, et al. "Learning skillful medium-range global weather forecasting." Science 2023

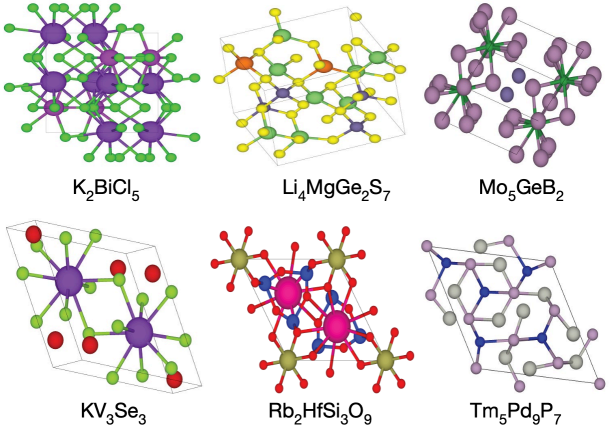
Protein Folding with AlphaFold



- AlphaFold 2 models relationships between amino acid residues using graph-inspired techniques, including attention mechanisms and pairwise representations. This approach captures essential relational information with customized attention layers optimized for protein folding.

Jumper et al. "Highly accurate protein structure prediction with AlphaFold." Nature 2021

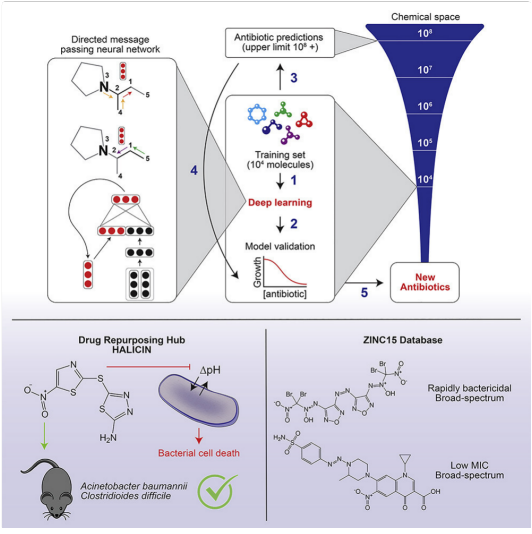
Materials Science



- DeepMind's AI tool, GNoME (Graph Network for Materials Exploration), uses graph neural networks (GNNs) to discover and predict the stability of new crystalline materials. This approach identified over 2 million new structures, with about 380,000 classified as stable.

Merchant, et al. "Scaling deep learning for materials discovery." Nature 2023

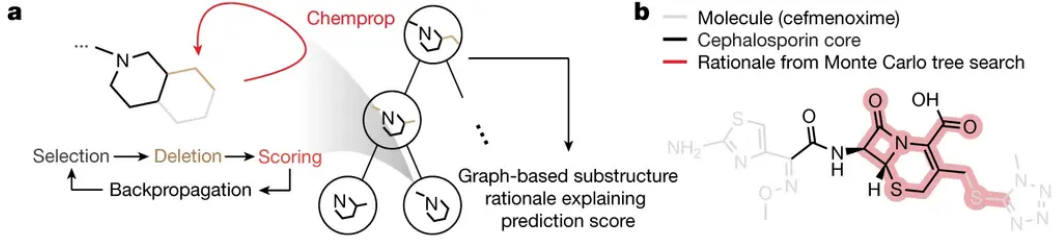
Drug Discovery: Graph Neural Networks for Antibiotic Discovery



- MIT's COLLINS LAB uses a Directed Message Passing Neural Network (D-MPNN) to model molecules as graphs (atoms as nodes, bonds as edges), enabling direct prediction of antibacterial efficacy and efficient exploration of chemical space.

Stokes et al., "A deep learning approach to antibiotic discovery," Cell, 2020.

Explainable AI for Antibiotic Discovery



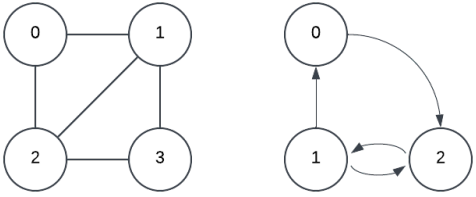
- Researchers from MIT and Harvard have developed an explainable AI system to identify antibiotic activity in molecules by combining a Graph Neural Network (GNN) with Monte Carlo Tree Search (MCTS). This approach isolates specific chemical substructures, known as *rationales*, that are associated with antibiotic effectiveness.

Wong et al., "Discovery of a structural class of antibiotics with explainable deep learning," Nature, 2024.

Outline

Graph Fundamentals

Graph Definition:



- A graph $\mathcal{G}$ is an ordered pair $(\mathcal{V}, \mathcal{E})$.
- $\mathcal{V}$ is a set of **vertices** (or nodes) representing entities like users or items.
- $\mathcal{E}$ is a set of **edges**, representing relationships between vertices, such as interactions.
- **Directed** edges are ordered pairs.
- **Undirected** edges are unordered pairs.

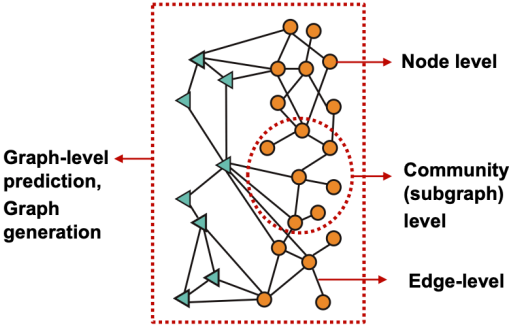
Graph Representations:

- **Adjacency Matrix**: $\boldsymbol{A}_{ij}$ indicates the presence or weight of an edge between nodes i and j.
- **Edge List**: A list of node pairs (or triplets, if edges have weights or attributes).

$$\boldsymbol{A}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \quad E_2 = \{(0,2),(1,0),(1,2),(2,1)\}$$

where $\boldsymbol{A}_1$ is the adjacency matrix for the first graph and E_2 is the edge list for the second graph.

Graph Learning Tasks



- **Node Level Tasks**:
  - **Social Networks**: Predict user interests or groups.
  - **Knowledge Graphs**: Classify entities in a knowledge base.
- **Edge Level Tasks**:
  - **Recommendation Systems**: Predict user-item interactions (e.g., Uber Eats, Pinterest).
  - **Social Networks**: Suggest new connections between users.
- **Graph Level Tasks**:
  - **Drug Discovery**: Classify molecules based on properties (e.g., antibiotic activity).
  - **Protein Function Prediction**: Predict biological function based on protein structures.

Outline

Using MLP on Graphs

MLP for Node Feature Updates:

- Each node i in the graph has a **feature vector** $\boldsymbol{x}_i \in \mathbb{R}^d$.
- Node features can be updated by a Multi-Layer Perceptron (MLP):

$$\boldsymbol{h}_i = \phi(\boldsymbol{W}\boldsymbol{x}_i)$$

  where $\boldsymbol{W} \in \mathbb{R}^{m \times d}$ is the weight matrix, σ is an activation function, and bias terms are omitted here for simplicity.
- In matrix form, the update becomes:

$$\boldsymbol{H} = \phi(\boldsymbol{X}\boldsymbol{W}^\top),$$

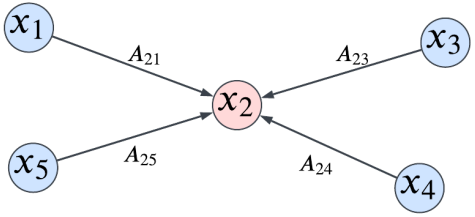
  where:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix}^\top \in \mathbb{R}^{n \times d}, \quad \boldsymbol{H} = \begin{bmatrix} \boldsymbol{h}_1 & \cdots & \boldsymbol{h}_n \end{bmatrix}^\top \in \mathbb{R}^{n \times m}.$$

Limitation of Ignoring Graph Structure:

- This approach does not consider **graph structure**, treating each node as an isolated point.
- **Drawback**: Without adjacency information, we lose valuable relationships and structural insights of the graph, which are often crucial for graph-based tasks.

Simple Aggregation with Neighbors



- Assume the graph is *unweighted* and *undirected*, i.e., $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji} = 1$ if edge (i, j) exists.
- The node feature is updated by **aggregating** with its neighbors:

$$\boldsymbol{h}_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \boldsymbol{W}\boldsymbol{x}_j\right) = \sigma\left(\sum_{j=1}^{n} \boldsymbol{A}_{ij}\boldsymbol{W}\boldsymbol{x}_j\right),$$

  where $\mathcal{N}_i$ is the set of neighbors of node i.
- In matrix form, the update becomes: $\boldsymbol{H} = \sigma(\boldsymbol{A}\boldsymbol{X}\boldsymbol{W}^\top)$
- A **deeper** neural network can be created by repeating this recurrent update:

$$\boldsymbol{H}^{(\ell)} = \sigma(\boldsymbol{A}\boldsymbol{H}^{(\ell-1)}\boldsymbol{W}^{(\ell)\top})$$

  where $\boldsymbol{H}^0 = \boldsymbol{X}$.

Graph Convolutional Network (GCN)

- Standard adjacency matrices omit **self-connections**, so we modify by adding self-loops:

$$\boldsymbol{h}_i^{(\ell)} = \sigma\left(\boldsymbol{W}^{(\ell)}\boldsymbol{h}_i^{(\ell-1)} + \sum_{j\in\mathcal{N}_i\setminus\{i\}} \boldsymbol{W}^{(\ell)}\boldsymbol{h}_j^{(\ell-1)}\right) \quad \Rightarrow \quad \boldsymbol{H}^{(\ell)} = \sigma\left(\hat{\boldsymbol{A}}\boldsymbol{H}^{(\ell-1)}\boldsymbol{W}^{(\ell)\top}\right)$$

  where $\hat{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$.

- **Normalization** or averaging the aggregation prevents output scaling:

$$\boldsymbol{h}_i^{(\ell)} = \sigma\left(\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} \boldsymbol{W}^{(\ell)}\boldsymbol{h}_j^{(\ell-1)}\right) \quad \Rightarrow \quad \boldsymbol{H}^{(\ell)} = \sigma\left(\hat{\mathbf{D}}^{-1}\hat{\boldsymbol{A}}\boldsymbol{H}^{(\ell-1)}\boldsymbol{W}^{(\ell)\top}\right)$$

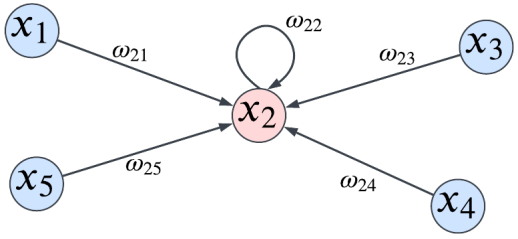
  where $\hat{\mathbf{D}}_{ii} = \sum_{j=1}^{n} \hat{\boldsymbol{A}}_{ij}$ is the degree matrix.

- The **graph convolutional network (GCN)** apply **symmetric** normalization:

$$\boldsymbol{h}_i^{(\ell)} = \sigma\left(\sum_{j\in\mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_i|}\sqrt{|\mathcal{N}_j|}}\boldsymbol{W}^{(\ell)}\boldsymbol{h}_j^{(\ell-1)}\right) \quad \Rightarrow \quad \boldsymbol{H}^{(\ell)} = \sigma\left(\hat{\mathbf{D}}^{-1/2}\hat{\boldsymbol{A}}\hat{\mathbf{D}}^{-1/2}\boldsymbol{H}^{(\ell-1)}\boldsymbol{W}^{(\ell)\top}\right)$$

Kipf and Welling, "Semi-Supervised Classification with Graph Convolutional Networks," ICLR 2017

GCN: Illustration



The note features are updated in **graph convolutional network (GCN)** using **symmetric** normalization:

$$\boldsymbol{h}_i^{(\ell)} = \sigma\left(\sum_{j \in \mathcal{N}_i} \omega_{ij} \boldsymbol{W}^{(\ell)} \boldsymbol{h}_j^{(\ell-1)}\right)$$

where $\omega_{ij} := \hat{\boldsymbol{A}}_{ij} / \sqrt{\hat{\boldsymbol{D}}_{ii} \hat{\boldsymbol{D}}_{jj}}$.

Graph Attention Network (GAT)

GAT uses **attention** mechanisms to assign different levels of importance to neighboring nodes.

- Computes the **attention score** of node j to node i based on their features:

$$\boldsymbol{e}_{ij}^{(\ell)} = a^{(\ell)}(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)})$$

  where a is a *shared* attention mechanism that outputs attention scores for each edge.

- A common choice of a is a fully connected layer:

$$a^{(\ell)}(\boldsymbol{h}_i, \boldsymbol{h}_j) = \text{LeakyReLU}\left(\boldsymbol{a}^{(\ell)\top}[\boldsymbol{W}^{(\ell)}\boldsymbol{h}_i; \boldsymbol{W}^{(\ell)}\boldsymbol{h}_j]\right)$$

  where $[\cdot;\cdot]$ denotes concatenation, and $\boldsymbol{a}^{(\ell)}$ and $\boldsymbol{W}^{(\ell)}$ are learnable.

- Applies a softmax function to normalize attention scores

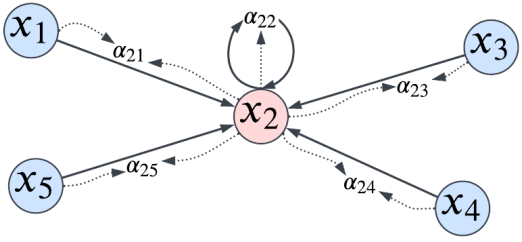
$$\boldsymbol{\alpha}_i^{(\ell)} = \text{softmax}(\boldsymbol{e}_i^{(\ell)})$$

- Aggregates **neighboring** features using normalized attention coefficients to update node features:

$$\boldsymbol{h}_i^{(\ell+1)} = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{(\ell)} \boldsymbol{W}\boldsymbol{h}_j^{(\ell)}\right)$$

Veličković et al., "Graph Attention Networks," ICLR 2018

GAT: Illustration



- Computes the **attention score** of node j to node i based on their features:

$$e_{ij}^{(\ell)} = a^{(\ell)}(\boldsymbol{h}_i^{(\ell-1)}, \boldsymbol{h}_j^{(\ell-1)})$$

- Applies a softmax function to normalize attention scores

$$\boldsymbol{\alpha}_i^{(\ell)} = \text{softmax}(\boldsymbol{e}_i^{(\ell)})$$

- Aggregates **neighboring** features using normalized attention coefficients to update node features:

$$\boldsymbol{h}_i^{(\ell+1)} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(\ell)} \boldsymbol{W} \boldsymbol{h}_j^{(\ell)}\right)$$

Message Passing Neural Network (MPNN)

MPNNs extend GCNs and GATs by incorporating **edge features** (e.g., chemical bonds).

- Given node features $\boldsymbol{h}_i^{(l-1)}$, $\boldsymbol{h}_j^{(l-1)}$, and edge features $\boldsymbol{e}_{ij}$, the **message function** $f_e^{(\ell)}$ computes:

$$\boldsymbol{m}_{ij}^{(l)} = f_e^{(l)}(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}, \boldsymbol{e}_{ij}).$$

- A common choice for f_e is a fully connected layer:

$$f_e^{(l)}(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}, \boldsymbol{e}_{ij}) = \sigma\left(\boldsymbol{W}_e^{(l)}[\boldsymbol{h}_i^{(l-1)}; \boldsymbol{h}_j^{(l-1)}; \boldsymbol{e}_{ij}]\right)$$

  where $[\cdot;\cdot]$ denotes concatenation and $\boldsymbol{W}_e^{(\ell)}$ is learnable matrix.

- The node feature $\boldsymbol{h}_i^{(l)}$ is updated by **aggregating messages** from neighbors:

$$\boldsymbol{h}_i^{(l)} = f_v^{(l)}(\boldsymbol{h}_i^{(l-1)}, \{\boldsymbol{m}_{ij}^{(l)}\}_j)$$

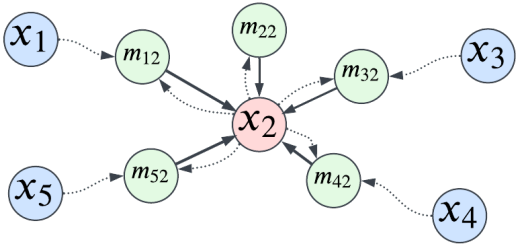
- A common aggregation function f_v sums messages:

$$f_v^{(l)}(\boldsymbol{h}_i^{(l-1)}, \{\boldsymbol{m}_{ij}^{(l)}\}_j) = \sigma\left(\boldsymbol{W}_v^{(l)}\boldsymbol{h}_i^{(l-1)} + \boldsymbol{W}_m^{(l)}\sum_{j\in\mathcal{N}_i}\boldsymbol{m}_{ij}^{(l)}\right)$$

  where $\boldsymbol{W}_v^{(l)}$ and $\boldsymbol{W}_m^{(l)}$ are learnable matrices.

Gilmer et al., "Neural Message Passing for Quantum Chemistry," ICML 2017

MPNN: Illustration



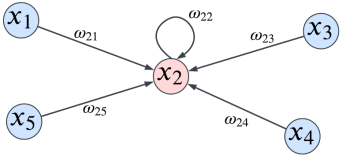
- Given node features $\boldsymbol{h}_i^{(l-1)}$, $\boldsymbol{h}_j^{(l-1)}$, and edge features $\boldsymbol{e}_{ij}$, the **message function** $f_e^{(\ell)}$ computes:

$$\boldsymbol{m}_{ij}^{(l)} = f_e^{(l)}(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}, \boldsymbol{e}_{ij}).$$

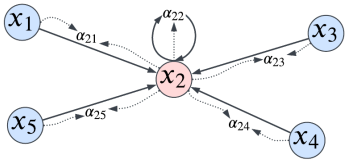
- The node feature $\boldsymbol{h}_i^{(l)}$ is updated by **aggregating messages** from neighbors:

$$\boldsymbol{h}_i^{(l)} = f_v^{(l)}(\boldsymbol{h}_i^{(l-1)}, \{\boldsymbol{m}_{ij}^{(l)}\}_j)$$

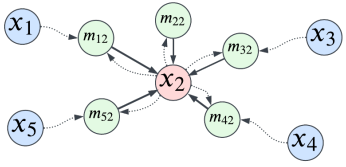
Summary



GCNs



GATs



MPNNs

Outline

Node Classification with GNNs

Goal: Predict a label for each node in the graph.

- **Node Representation**: After L layers, each node i has an updated representation:

$$\mathbf{h}_i^{(L)} = f_{\text{GNN}}(\mathbf{X}, \mathbf{A})$$

  where $\mathbf{X}$ is the node feature matrix and $\mathbf{A}$ is the adjacency matrix.

- **Prediction**: Apply a classifier to the final node representation:

$$\hat{y}_i = \text{softmax}(\mathbf{W}_{\text{out}}\mathbf{h}_i^{(L)})$$

  where $\mathbf{W}_{\text{out}}$ is a learnable weight matrix for classification.

- **Loss Function**: Use cross-entropy loss to compare predictions with ground truth labels:

$$\mathcal{L} = -\sum_{i \in \mathcal{V}} y_i \log(\hat{y}_i)$$

  where y_i is the true label of node i.

Adapted for node classification tasks with GNNs

Graph Classification with GNNs

Goal: Predict a label for the entire graph.

- **Node Representations**: After L layers, each node i has a final representation:

$$\mathbf{h}_i^{(L)} = f_{\text{GNN}}(\mathbf{X}, \mathbf{A})$$

- **Graph-Level Representation**: Aggregate all node representations to form a single graph vector:

$$\mathbf{h}_{\text{graph}} = \text{AGGREGATE}(\{\mathbf{h}_i^{(L)} | i \in \mathcal{V}\})$$

  Common choices for AGGREGATE include mean, sum, or max pooling.

- **Prediction**: Apply a classifier to the aggregated graph representation:

$$\hat{y}_{\text{graph}} = \text{softmax}(\mathbf{W}_{\text{out}} \mathbf{h}_{\text{graph}})$$

- **Loss Function**: Use cross-entropy loss to match predictions with true graph labels:

$$\mathcal{L} = -y_{\text{graph}} \log(\hat{y}_{\text{graph}})$$

  where y_{graph} is the true graph label.

Adapted for graph classification tasks with GNNs

Link Prediction with GNNs

Goal: Predict the existence of an edge between two nodes.

- **Node Representations**: After L layers, each node i has a final representation:

$$\mathbf{h}_i^{(L)} = f_{\text{GNN}}(\mathbf{X}, \mathbf{A})$$

- **Edge Representation**: For a pair of nodes i and j, compute a combined feature vector to represent the link:

$$\mathbf{z}_{ij} = g(\mathbf{h}_i^{(L)}, \mathbf{h}_j^{(L)})$$

  where g is a function such as concatenation $[\mathbf{h}_i^{(L)}; \mathbf{h}_j^{(L)}]$, element-wise product $\mathbf{h}_i^{(L)} \odot \mathbf{h}_j^{(L)}$, or distance-based functions.

- **Prediction**: Apply a scoring function to predict the likelihood of a link:

$$\hat{y}_{ij} = \sigma(\mathbf{W}_{\text{out}}\mathbf{z}_{ij})$$

  where σ is the sigmoid function.

- **Loss Function**: Use binary cross-entropy to compare predictions with actual labels:

$$\mathcal{L} = -\sum_{(i,j)\in\mathcal{E}} y_{ij}\log(\hat{y}_{ij}) + (1 - y_{ij})\log(1 - \hat{y}_{ij})$$

  where y_{ij} indicates if a link exists between i and j.

Adapted for link prediction tasks with GNNs