An algorithmic reasoning approach to GNNs

A project for the *Deep Learning* course

Angela Carraro, Matteo Scorcia

DSSC + IN20 - UNITS

Aim of the project

Graph Neural Networks can have a lot of meanings, there isn't just one architecture that can be recognized as "GNN". We will try to understand the general, abstract structure of a GNN that is presented in the book [4] (which also includes [1]) and to shed light about the relational inductive bias and combinatorial generalization of a GNN.

Our motivation is to better understand the extent to which graph neural networks are capable of **precise and logical reasoning**.

Our motivation is to better understand the extent to which graph neural networks are capable of **predict something, given a graph structured dataset**.

Our goal is to understand why GNNs are needed and how it works in respect to a "classic" deep learning structure.

Graph Analysis

Graphs are a widespread data structure and a universal language for describing and modelling complex systems. In the most general view, a graph is simply a collection of objects (i.e., nodes), along with a set of interactions (i.e., edges) between pairs of these objects.

Graphs are an important building block since they can naturally encode an entity-relationship structure, as well as an invariance to permutations (of both nodes and edges) and awareness of input sparsity.

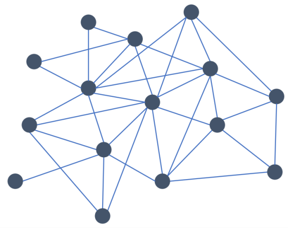


Figure 1: A graph.

Definition

A graph is a tuple $G = (V, E)$ where V is the set of nodes and E is the set of edges between these nodes. We denote an edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$, so $E \subseteq V \times V$. The graph is **undirected** if $(u, v) \in E \iff (v, u) \in E$, otherwise it is **directed**.

Given a node u in the graph, $\mathcal{N}(u)$ is u's graph neighborhood.

A convenient way to represent graphs is through an adjacency matrix $\mathbf{A} \in \mathbb{R}^{|V|\times|V|}$, with $\mathbf{A}[u, v] = 1$ if $(u, v) \in E$ and $\mathbf{A}[u, v] = 0$ otherwise. If the graph is undirected the matrix in *symmetric*. If the graph has weighted edges we have that $\mathbf{A}[u, v] \in \mathbb{R}$.

We also have node-level attributes / features represented using a matrix $\mathbf{X} \in \mathbb{R}^{d\times|V|}$ (with the ordering of the nodes consistent with the ordering in the adjacency matrix).

In some cases we also have graph features, which are real-valued *features associated with entire graphs*.

Machine learning on graphs

Machine learning tasks on graph data fall in one of these four categories:

- *node classification*: predict the label y_u associated with all the nodes $u \in V$
  $\longrightarrow$ E.g., predicting whether a user is a bot in a social network
- *edge prediction*: infer the edges between nodes in a graph
  $\longrightarrow$ E.g., content recommendation in online platforms, predicting drug side-effects, or inferring new facts in a relational databases
- *community detection*: infer latent community structures given only the input graph
  $\longrightarrow$ E.g., uncovering functional modules in genetic interaction networks, uncovering fraudulent groups of users in financial transaction networks
- *graph class./regr./clust.*: given a dataset of *multiple different graphs*, make independent predictions specific to each graph
  $\longrightarrow$ E.g., property prediction based on molecular graph structures

Blurring the boundaries of ML categories

Node classification can appear to be a *standard supervised classification* $\longrightarrow$ but the nodes in a graph are **not** independent and identically distributed (i.i.d.)!!!

Usually in supervised ML models we assume that:

- each datapoint is statistically independent from all the other datapoints
  $\rightarrow$ otherwise we might need to model the dependencies between all our input points
- the datapoints are identically distributed
  $\rightarrow$ otherwise we can not guarantee that the model will generalize to new datapoints.

Node classification completely breaks this i.i.d. assumption! Rather than modeling a set of i.i.d. datapoints, we are instead modeling an interconnected set of nodes.

Instead **graph regression and classification** are analogues of standard supervised learning: each graph is an i.i.d. datapoint associated with a label, and the goal is to use a labeled set of training points to learn a mapping from datapoints (i.e., graphs) to labels.

What are properties and statistics useful to characterize the nodes in a graph?

Node degree d_u

Number of edges incident to a node $u \in V$, so how many neighbors the node has:

$$d_u = \sum_{v \in V} \mathbf{A}[u, v]. \tag{1}$$

To obtain a more powerful measure of *importance*, we can measure the node centrality.

Eigenvector centrality e_u

Recurrence relation in which the node's centrality is proportional, via a constant λ, to the average centrality of its neighbors:

$$e_u = \frac{1}{\lambda} \sum_{v \in V} \mathbf{A}[u, v] e_v \; \forall u \in V \quad \Longleftrightarrow \quad \lambda \mathbf{e} = \mathbf{A}\mathbf{e}, \quad \text{with } \mathbf{e} \text{ vector of node centralities.} \tag{2}$$

Perron-Frobenius theorem $\Longrightarrow$ $\mathbf{e}$ is given by the eigenvector of $\mathbf{A}$'s largest eigenvalue.

Graph-level hand-engineered features

Bag of nodes

The simplest approach to defining a graph-level feature is to just aggregate node-level statistics. For example, one can compute histograms or other summary statistics based on the degrees, centralities, and clustering coefficients of the nodes in the graph. This aggregated information can then be used as a graph-level representation. The downside to this approach is that it is entirely based upon local node-level information and can miss important global properties in the graph.

Iterative neighborhood aggregation

One way to improve the basic bag of nodes approach is using a strategy of iterative neighborhood aggregation. The idea with these approaches is to extract node-level features that contain more information than just their local ego graph, and then to aggregate these richer features into a graph-level representation (e.g. *WeisfeilerLehman (WL) algorithm and kernel, graphlets or path-based methods*).

Statistical measures of neighborhood overlap between pairs of nodes $\longrightarrow$ quantify the extent to which a pair of nodes are related.

E.g., simply count the number of neighbors that two nodes share:

$$\mathbf{S}[u, v] = |\mathcal{N}(u) \cap \mathcal{N}(v)|, \tag{3}$$

where $\mathbf{S}[u, v]$ denotes the value quantifying the relationship between nodes u and v and let $\mathbf{S} \in \mathbb{R}^{|V| \times |V|}$ denote the similarity matrix summarizing all the pairwise node statistics.

Relation prediction $\longrightarrow$ Given $\mathbf{S}[u, v]$, assume that the likelihood of an edge (u, v) is

$$\mathbf{P}(\mathbf{A}[u, v] = 1) \propto \mathbf{S}[u, v], \tag{4}$$

then set a threshold to determine when to predict the existence of an edge.

We know only a subset of the true edges $E_{\text{train}} \subset E$ and our hope is that $\mathbf{S}$ computed on it will lead to accurate predictions about the existence of test (unseen) edges.

The traditional approaches to learning over graphs are limited due to the fact that they *require careful, hand-engineered statistics and measures*.

We will introduce an alternative approach to learning over graphs: graph representation learning. Instead of extracting hand-engineered features, we will seek to *learn* representations that encode structural information about the graph.

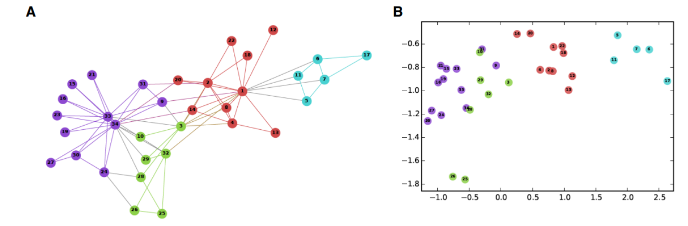


Figure 2: **A**, Graph structure of the Zachary Karate Club social network, the colors represent different communities. **B**, Twodimensional visualization of node embeddings generated from this graph (DeepWalk method).

An Encoder-Decoder Perspective

Node embeddings

There are various methods for learning node embeddings.

Goal: encode nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood. I.e., we want to project nodes into a latent space where geometric relations correspond to relationships (e.g., edges) in the original graph.

Examples: the encoder-decoder framework, factorization-based approaches, random walk embeddings.

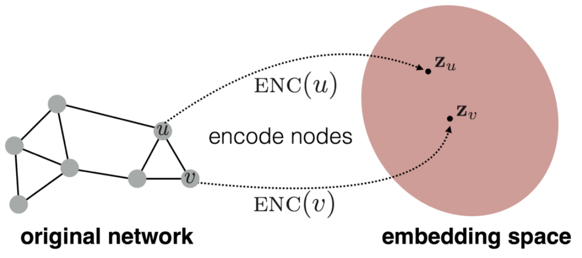
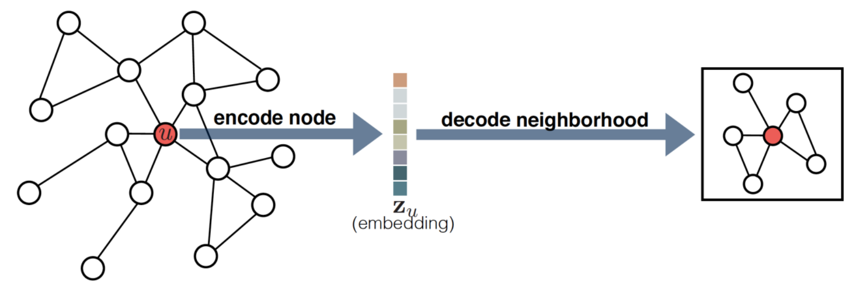


Figure 3: Illustration of the node embedding problem. Our goal is to learn an encoder (enc), which maps nodes to a low-dimensional embedding space. These embeddings are optimized so that distances in the embedding space reflect the relative positions of the nodes in the original graph.

We will focus on the framework of encoding and decoding graphs. We view the graph representation learning problem as involving two key operations:

- First, an encoder model maps each node in the graph into a low-dimensional vector or embedding.
- Next, a decoder model uses the low-dimensional node embeddings to reconstruct information about each node local neighborhood in the original graph.



Formally, the encoder is a function that maps nodes $v \in V$ to a corresponding *vector embeddings* $\mathbf{z}_v \in \mathbb{R}^d$. In the simplest case, the encoder has the following signature:

$$\text{ENC} : V \to \mathbb{R}^d, \tag{5}$$

meaning that the encoder takes node IDs as input to generate the node embeddings.

In most work on node embeddings, the encoder relies on what we call the shallow embedding approach, where this encoder function is simply an embedding lookup based on the node ID. In other words, we have that

$$\text{ENC}(v) = \mathbf{Z}[v], \tag{6}$$

where $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ is a matrix containing the embedding vectors for all nodes and $\mathbf{Z}[v]$ denotes the row of $\mathbf{Z}$ corresponding to node v.

The Decoder

The role of the decoder is to *reconstruct certain graph statistics* from the node embeddings that are generated by the encoder, e.g. predict the set of neighbors $\mathcal{N}(u)$ of a node u or its row $\mathbf{A}[u]$ in the graph adjacency matrix.

Pairwise decoders are the standard and have signature: $\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$.

They can be interpreted as predicting the relationship or similarity between pairs of nodes, e.g. predict whether two nodes are neighbors in the graph.

Goal: optimize the encoder and decoder to minimize the reconstruction loss so that

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v], \tag{7}$$

where $\mathbf{S}[u, v]$ is a graph-based similarity measure between nodes (e.g. *neighborhood overlap statistics*). For example, the simple reconstruction objective of predicting whether two nodes are neighbors would correspond to $\mathbf{S}[u, v] := \mathbf{A}[u, v]$.

To achieve the reconstruction objective (7), the standard practice is to minimize an empirical reconstruction loss $\mathcal{L}$ over a set of training node pairs $\mathcal{D}$:

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell(\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u, v]), \tag{8}$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a loss function measuring the discrepancy between the decoded (i.e., estimated) similarity values $\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v)$ and the true values $\mathbf{S}[u, v]$.

Depending on the definition of the decoder (DEC) and similarity function ($\mathbf{S}$), the loss function ℓ might be a mean-squared error or even a classification loss, such as cross entropy. Thus, the overall objective is to train the encoder and the decoder so that pairwise node relationships can be effectively reconstructed on the training set $\mathcal{D}$. Most approaches minimize the loss in 8 using stochastic gradient descent, but there are certain instances when more specialized optimization methods (e.g., based on matrix factorization) can be used.

Limitations of Shallow Embeddings

Problems: In shallow embedding approaches, the encoder model is simply an embedding lookup (6), which trains a unique embedding for each node in the graph.

Besides, there are some important drawbacks:

- They do not share any parameters between nodes in the encoder, since the encoder directly optimizes a unique embedding vector for each node $\longrightarrow$ both statistically and computationally inefficient.
- They do not leverage node features in the encoder.
- They are inherently *transductive*, i.e. they can only generate embeddings for nodes that were present during the training phase $\longrightarrow$ cannot be used on *inductive* applications, which involve generalizing to unseen nodes after training.

Solution: use more sophisticated encoders that depend more generally on the structure and attributes of the graph $\Longrightarrow$ graph neural networks (GNNs).

The Graph Neural Network Model

The simplest (and worst) GNN

To define a deep neural network over graphs one could simply use the adjacency matrix as input to a deep neural network. For example, to generate an embedding of an entire graph we could simply flatten the adjacency matrix and feed the result to a multi-layer perceptron (MLP):

$$\mathbf{z}_G = \mathsf{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \ldots \oplus \mathbf{A}[|\mathcal{V}|]); \tag{9}$$

where $\mathbf{A}[i] \in \mathbf{R}^{|\mathcal{V}|}$ denotes a row of the adjacency matrix and we use $\oplus$ to denote vector concatenation.

Issue: this approach *depends on the arbitrary ordering of nodes that we used in the adjacency matrix*! In other words, such a model is **not** permutation invariant.

Any function f that takes an adjacency matrix $\mathbf{A}$ as input should ideally satisfy one of the two following properties, given a permutation matrix $\mathbf{P}$:

$$f(\mathbf{PAP}^T) = f(\mathbf{A}) \qquad \text{Permutation Invariance} \tag{10}$$

$$f(\mathbf{PAP}^T) = \mathbf{P}f(\mathbf{A}) \qquad \text{Permutation Equivariance,} \tag{11}$$

Neural Message Passing

How we can take an input graph $G = (V, E)$, along with a set of node features $\mathbf{X} \in \mathbb{R}^{d\times|V|}$, and use this information to generate node embeddings $\mathbf{z}_u, \forall u \in V$?

During each message-passing iteration k in a GNN, a hidden embedding $\mathbf{h}_u^{(k)}$ corresponding to each node $u \in \mathcal{V}$ is updated according to information aggregated from u's graph neighborhood $\mathcal{N}(u)$. This message-passing update can be expressed as follows:

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}; \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \quad (12)$$

$$= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), \quad (13)$$

where UPDATE and AGGREGATE are arbitrary differentiable functions (i.e., neural networks) and $\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}(\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\})$ is the “message” that is aggregated from u's graph neighborhood $\mathcal{N}(u)$ (neighborhood aggregation operation). The different iterations of message passing are also sometimes known as the different “layers” of the GNN.

The initial embeddings at k = 0 are set to the input features for all the nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in \mathcal{V}$. After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node, i.e.,

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{14}$$

Note that since the AGGREGATE function takes a set as input, GNNs defined in this way are permutation equivariant by design.

Basic intuition $\longrightarrow$ 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.

- *structural information* about the graph $\longrightarrow$ after k iterations, $\mathbf{h}_u^{(k)}$ might encode information about the degrees of all the nodes in u's k-hop neighborhood.
- *feature-based information* about the graph $\longrightarrow$ after k iterations, $\mathbf{h}_u^{(k)}$ also encodes information about all the features in its k-hop neighborhood $\rightarrow$ analogous to convolutional kernels in CNNs!

The basic GNN message passing is defined as

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right), \tag{15}$$

where $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ are trainable parameter matrices and σ denotes an elementwise non-linearity (e.g., a tanh or ReLU). The bias term $b^{(k)} \in \mathbb{R}^{d^{(k)}}$ is often omitted for notational simplicity, but including it can be important to achieve strong performance.

We can equivalently define the basic GNN through the UPDATE and AGGREGATE functions:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v\in\mathcal{N}(u)}\mathbf{h}_v, \tag{16}$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right), \tag{17}$$

Any GNNs can also be succinctly defined using graph-level equations. In the case of a basic GNN, we can write the graph-level definition of the model as follows:

As a simplification of the neural message passing approach, it is common to add self-loops to the input graph and omit the explicit update step. In this approach we define the message passing simply as

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)},\ \forall v \in \mathcal{N}(u) \cup \{u\}\}), \tag{19}$$

where now the aggregation is taken over the set $\mathcal{N}(u) \cup \{u\}$, i.e., the node's neighbors as well as the node itself. The benefit of this approach is that we no longer need to define an explicit update function, as the update is implicitly defined through the aggregation method.

In the case of the basic GNN, adding self-loops is equivalent to sharing parameters between the $\mathbf{W}_{\text{self}}$ and $\mathbf{W}_{\text{neigh}}$ matrices, which gives the following graph-level update:

$$\mathbf{H}^{(k)} = \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right). \tag{20}$$

In the following chapters we will refer to this as the self-loop approach.

The most basic neighborhood aggregation operation (16) simply takes the sum of the neighbor embeddings. One issue with this approach is that it can be unstable and highly sensitive to node degrees. One solution to this problem is to simply *normalize the aggregation operation* based upon the degrees of the nodes involved:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|}. \tag{21}$$

Another normalization factor is the symmetric normalization:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{22}$$

The popular graph convolutional network (GCN) employs the *symmetric-normalized aggregation* as well as the *self-loop approach*, using as message passing function

$$h_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{23}$$

Improving the Aggregation Layer

An aggregation function with the following form is a universal set function approximator:

$$\mathbf{m}_{\mathcal{N}(u)} = \mathsf{MLP}_\theta\left(\sum_{v\in\mathcal{N}(u)} \mathsf{MLP}_\phi(\mathbf{h}_v)\right). \tag{24}$$

So any permutation-invariant function that maps a set of embeddings to a single embedding can be approximated to an arbitrary accuracy by a model following (24).
$\longrightarrow$ set pooling leads small increases in performance but increased risk of overfitting

Another strategy is to apply attention: assign an attention weight or importance to each neighbor, which is used to weigh this neighbor's influence during the aggregation step. The first GNN model to apply this style of attention was the Graph Attention Network (GAT):

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v\in\mathcal{N}(u)} \alpha_{u,v}\mathbf{h}_v, \tag{25}$$

where $\alpha_{u,v}$ denotes the attention on neighbor $v \in \mathcal{N}(u)$ (*neighborhood attention*) when we are aggregating information at node u.

Visualize a single GAT step

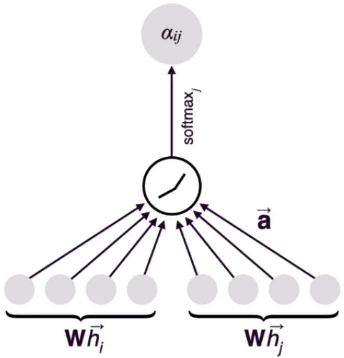


Figure 4: Attention mechanism. It looks at features of node i and its neighbor j, then the attention function computes the coefficient α_{ij}, which signifies the influence of node i to node j.

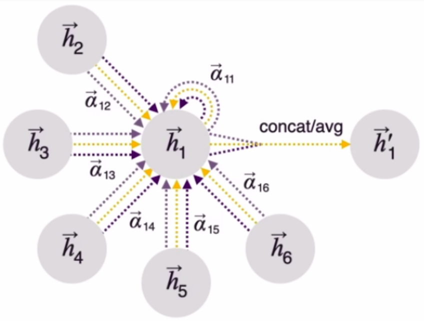


Figure 5: Multi-head attention mechanism. Each colored line indicates a different way in which the node receives information from its immediate neighbors, which is aggregated to produce an updated representation.

Images from [8]

Graph Pooling

What if we want to learn an embedding $\mathbf{z}_G$ for the entire graph G? $\longrightarrow$ graph pooling

We want to design a pooling function f_p, which maps a set of node embeddings $\{\mathbf{z}_1, \dots, \mathbf{z}_{|V|}\}$ to an embedding $\mathbf{z}_G$ that represents the full graph.

One approach is to simply to take a sum (or mean) of the node embeddings:

$$\mathbf{z}_G = \frac{\sum_{v \in V} \mathbf{z}_u}{f_n(|V|)}, \tag{26}$$

where f_n is some normalizing function (e.g., the identity function, so $\mathbf{z}_G = (\sum_{v \in V} \mathbf{z}_u)/|V|$).

Limitation: it does not exploit the structure of the graph! We want to exploit the graph topology at the pooling stage. $\longrightarrow$ use graph clustering or coarsening

The presentation in this chapter so far has focused on the most popular style of GNN message passing, which operates largely at the node level. However, the GNN message passing approach can also be generalized to leverage edge and graph-level information at each stage of message passing. For example, in the more general approach proposed by [1], we define each iteration of message passing according to the following equations:

$$\mathbf{h}_{(u,v)}^{(k)} = \text{UPDATE}_{\text{edge}}(\mathbf{h}_{(u,v)}^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}, \mathbf{h}_G^{(k-1)}) \tag{27}$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}_{\text{node}}(\{\mathbf{h}_{(u,v)}^{(k)}, \ \forall v \in \mathcal{N}(u)\}) \tag{28}$$

$$\mathbf{h}_u^{(k)} = \text{UPDATE}_{\text{node}}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}, \mathbf{h}_G^{(k-1)}) \tag{29}$$

$$\mathbf{h}_G^{(k)} = \text{UPDATE}_{\text{graph}}(\mathbf{h}_G^{(k-1)}, \{\mathbf{h}_u^{(k)}, \ \forall u \in V\}, \{\mathbf{h}_{(u,v)}^{(k)}, \forall (u,v) \in E\}). \tag{30}$$

The important innovation in this generalized message passing framework is that, during message passing, we generate hidden embeddings $\mathbf{h}_{(u,v)}^{(k)}$ for each edge in the graph, as well as an embedding $h_G^{(k)}$ corresponding to the entire graph. This allows the message passing model to easily integrate edge and graph-level features, and recent work has also shown this generalized message passing approach to have benefits compared to a

In the vast majority of current real-world applications, GNNs are used for one of three tasks:

- *GNNs for Node Classification* $\rightarrow$ train GNNs in a fully-supervised manner with a negative log-likelihood loss:

$$\mathcal{L} = \sum_{u \in V_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)), \tag{31}$$

where $\mathbf{y}_u$ is a one-hot vector indicating the class of training node $u \in V_{\text{train}}$ and $\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)$ denotes the predicted probability that the node belongs to the class $\mathbf{y}_u$, computed via the softmax function:

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\top \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^\top \mathbf{w}_j}}, \tag{32}$$

where $\mathbf{w}_i \in \mathbb{R}^d, i = 1, \ldots, c$ are trainable parameters.

- *GNNs for Graph Classification* $\rightarrow$ a softmax classification loss — analogous to 32 — is often used, with the key difference that the loss is computed with graph-level embeddings $\mathbf{z}_{G_i}$ over a set of labeled training graphs $\mathcal{G} = \{G_1, \ldots, G_n\}$. In recent years, GNNs have also witnessed success in regression tasks involving graph data. In these instances, it is standard to employ a squared-error loss of the following form:

$$\mathcal{L} = \sum_{G_i \in \mathcal{G}} \|\mathsf{MLP}(\mathbf{z}_{G_i}) - y_{G_i}\|_2^2, \tag{33}$$

where **MLP** is a densely connected neural network with a univariate output and $y_{G_i} \in \mathbb{R}$ is the target value for training graph G_i.

- *GNNs for Edge Prediction* $\rightarrow$ While classification tasks are by far the most popular application of GNNs, GNNs are also used in in edge prediction tasks, such as recommender systems and knowledge graph completion. In these applications, the standard practice is to employ the pairwise node embedding loss functions. In principle, GNNs can be combined with any of the pairwise loss functions discussed previously, with the output of the GNNs replacing the shallow embeddings.

Our GN framework imposes several strong relational inductive biases when used as components in a learning process. First, graphs can express arbitrary relationships among entities, which means the GN's input determines how representations interact and are isolated, rather than those choices being determined by the fixed architecture. For example, the assumption that two entities have a relationship—and thus should interact—is expressed by an edge between the entities' corresponding nodes. Similarly, the absence of an edge expresses the assumption that the nodes have no relationship and should not influence each other directly.

Second, graphs represent entities and their relations as sets, which are invariant to permutations. This means GNs are invariant to the order of these elements[1], which is often desirable. For example, the objects in a scene do not have a natural ordering (see Sec. **??**).

Third, a GN's per-edge and per-node functions are reused across all edges and nodes, respectively. This means GNs automatically support a form of combinatorial generalization (see Section **??**): because graphs are composed of edges, nodes, and

Combinatorial Optimization and Reasoning with GNN

Thank you for your attention!

☺

References

Peter W. Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *CoRR* abs/1806.01261 (2018). arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261.

Xavier Bresson. *Graph Convolutional Networks (GCNs)*. Youtube. 2020. URL: https://www.youtube.com/watch?v=Iiv9R6BjxHM.

Quentin Cappart et al. "Combinatorial optimization and reasoning with graph neural networks". In: *CoRR* abs/2102.09544 (2021). arXiv: 2102.09544. URL: https://arxiv.org/abs/2102.09544.

William L. Hamilton. *Graph Representation Learning*. Vol. 14. 3. Morgan and Claypool, 2020, pp. 1–159. URL: https://www.cs.mcgill.ca/~wlh/grl_book/.

William L. Hamilton. *Graph Representation Learning*. Youtube. 2021. URL: https://www.youtube.com/watch?v=fbRDfhNrCwo.

William L. Hamilton, Rex Ying, and Jure Leskovec. "Representation Learning on Graphs: Methods and Applications". In: *CoRR* abs/1709.05584 (2017). arXiv: 1709.05584. URL: http://arxiv.org/abs/1709.05584.

Petar Veličković. *Graph Representation Learning for Algorithmic Reasoning*. Youtube. 2020. URL: https://www.youtube.com/watch?v=IPQ6CPoluok.

Petar Veličković. *Intro to graph neural networks (ML Tech Talks)*. Youtube. 2021. URL: https://www.youtube.com/watch?v=8owQBFAHw7E&t=59s.

Petar Veličković. *Theoretical Foundations of Graph Neural Networks*. Youtube. 2021. URL: https://www.youtube.com/watch?v=uF53xsT7mjc.

Yujun Yan et al. "Neural Execution Engines: Learning to Execute Subroutines". In: *CoRR* abs/2006.08084 (2020). arXiv: 2006.08084. URL: https://arxiv.org/abs/2006.08084.