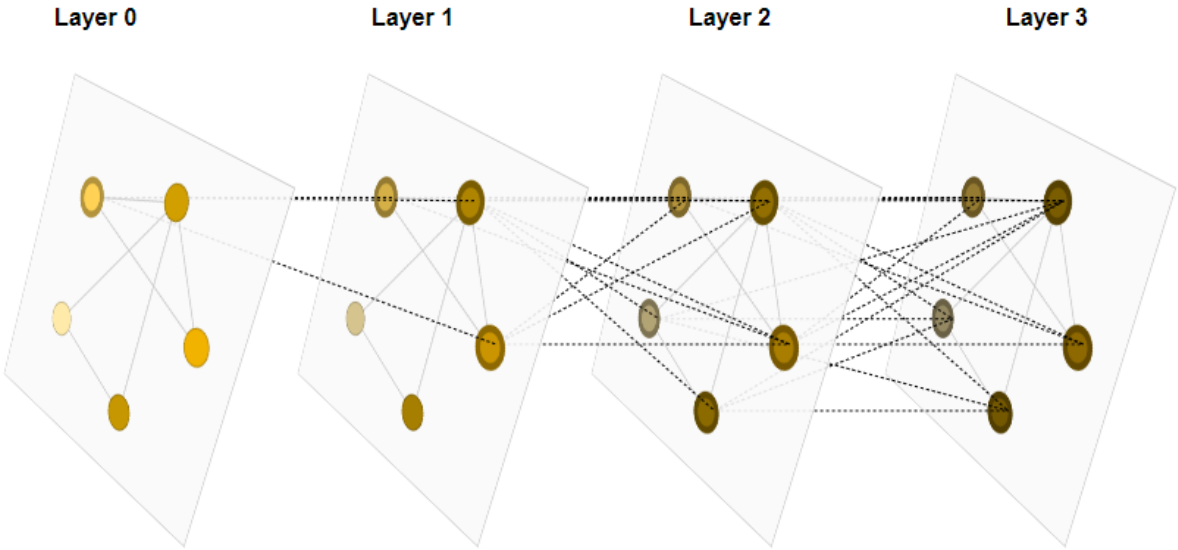
A Gentle Introduction to Graph Neural Networks

> Neural networks have been adapted to leverage the structure and properties of graphs. We explore the components needed for building a graph neural network - and motivate the design choices behind them.



This article explores and explains modern graph neural networks.

- First, we look at what kind of data is most naturally phrased as a graph, and some common examples.
- Second, we explore what makes graphs different from other types of data, and some of the specialized choices we have to make when using graphs.
- Third, we build a modern GNN, walking through each of the parts of the model, starting with historic modeling innovations in the field. We move gradually from a bare-bones implementation to a state-of-the-art GNN model.
- Fourth and finally, we provide a GNN playground where you can play around with a real-word task and dataset to build a stronger intuition of how each component of a GNN model contributes to the predictions it makes.

To start, let's establish what a graph is. A graph represents the relations (edges) between a collection of entities (nodes).

- **V** Vertex (or node) attributes e.g., node identity, number of neighbors
- **E** Edge (or link) attributes and directions e.g., edge identity, edge weight
- **U** Global (or master node) attributes e.g., number of nodes, longest path

To further describe each node, edge or the entire graph, we can store information in each of these pieces of the graph.

The edges can be directed, where an edge e has a source node, vsrc, and a destination node vdst. In this case, information flows from vsrc to vdst. They can also be undirected, where there is no notion of source or destination nodes, and information flows both directions. Note that having a single undirected edge is equivalent to having one directed edge from vsrc to vdst, and another directed edge from vdst to vsrc.

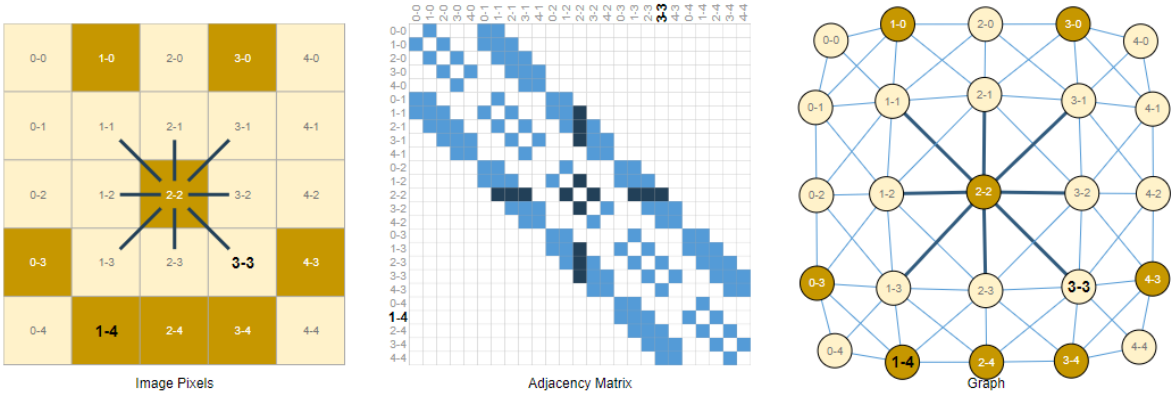
1. Graphs and Where to Find Them

You're probably already familiar with some types of graph data, such as social networks. However, graphs are an extremely powerful and general representation of data, we will show two types of data that you might not think could be modeled as graphs: images and text. Although counterintuitive (出乎意料的), one can learn more about the symmetries and structure of images and text by viewing them as graphs, and build an intuition that will help understand other less grid-like graph data, which we will discuss later.

1.1. Images as Graphs

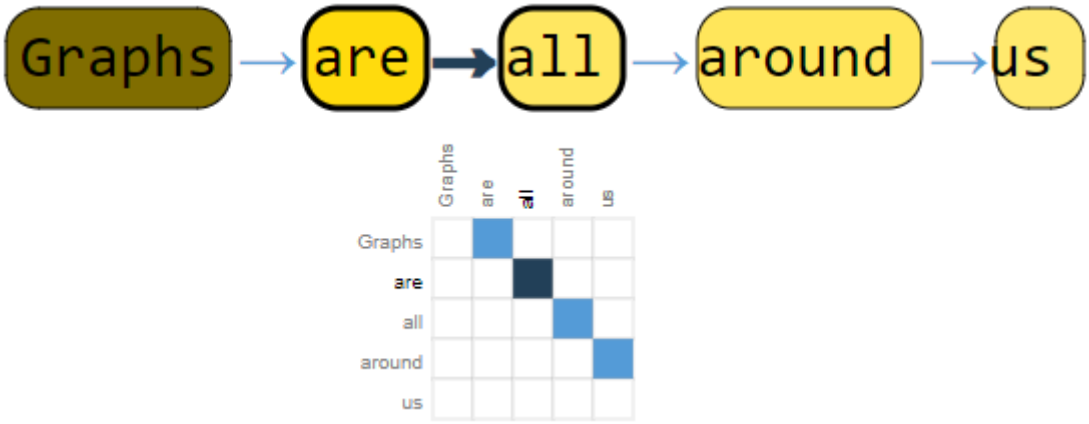
We typically think of images as rectangular grids with image channels, representing them as arrays. Another way to think of images is as graphs with regular structure, where each pixel represents a node and is connected via an edge to adjacent pixels. Each non-border pixel has exactly 8 neighbors, and the information stored at each node is a 3-dimensional vector representing the RGB value of the pixel.

A way of visualizing the connectivity of a graph is through its adjacency matrix. We order the nodes, in this case each of 25 pixels in a simple 5x5 image of a smiley face, and fill a matrix of $n{*nodes*}\times *n*{nodes}$ with an entry if two nodes share an edge. Note that each of these three representations below are different views of the same piece of data.



1.2. Text as Graphs

We can digitize (把...转变成数字形式) text by associating indices to each character, word, or token, and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node and is connected via an edge to the node that follows it.



Of course, in practice, this is not usually how text and images are encoded: these graph representations are redundant since all images and all text will have very regular structures. For instance, images have a banded structure in their adjacency matrix because all nodes (pixels) are connected in a grid. The adjacency matrix for text is just a diagonal line, because each word only connects to the prior word, and to the next one.

1.3. Graph-Valued Data in the Wild

Graphs are a useful tool to describe data you might already be familiar with. Let's move on to data which is more heterogeneously (混杂的) structured. In these examples, the number of neighbors to each node is variable (as opposed to the fixed neighborhood size of images and text). This data is hard to phrase in any other way besides a graph.

Molecules as graphs. Molecules are the building blocks of matter, and are built of atoms and electrons in 3D space. All particles are interacting, but when a pair of atoms are stuck in a stable distance from each other, we say they share a covalent bond. Different pairs of atoms and bonds have different distances (e.g. single-bonds, double-bonds). It's a very convenient and common abstraction to describe this 3D object as a graph, where nodes are atoms and edges are covalent bonds.

Social networks as graphs. Social networks are tools to study patterns in collective behaviour of people, institutions and organizations. We can build a graph representing groups of people by modelling individuals as nodes, and their relationships as edges. Unlike image and text data, social networks do not have identical adjacency matrices.

Citation networks as graphs. Scientists routinely cite other scientists' work when publishing papers. We can visualize these networks of citations as a graph, where each paper is a node, and each directed edge is a citation between one paper and another. Additionally, we can add information about each paper into each node, such as a word embedding of the abstract.

2. What Types of Problems Have Graph Structured Data?

There are three general types of prediction tasks on graphs: graph-level, node-level, and edge-level.

2.1. Graph-Level Task

In a graph-level task, our goal is to predict the property of an entire graph. For example, for a molecule represented as a graph, we might want to predict what the molecule smells like. This is analogous (相似) to image classification problems, where we want to associate a label to an entire image.

2.2. Node-Level Task

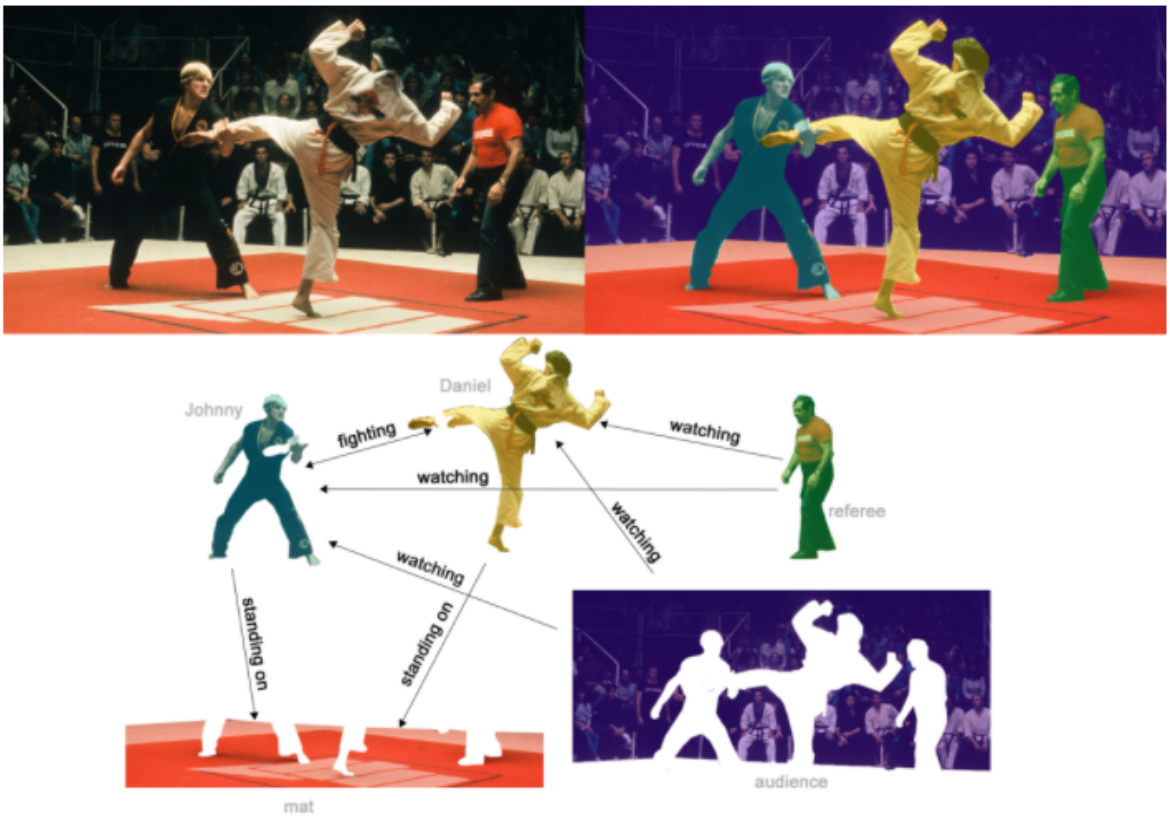
Node-level tasks are concerned with predicting the identity or role of each node within a graph. A classic example of a node-level prediction problem is Zach's karate club. The dataset is a single social network graph made up of individuals that have sworn allegiance to one of two karate clubs after a political rift. The prediction problem is to classify whether a given member becomes loyal to either Mr. Hi or John H, after the feud. In this case, distance between a node to either the Instructor or Administrator is highly correlated to this label.

Following the image analogy, node-level prediction problems are analogous to image segmentation, where we are trying to label the role of each pixel in an image.

2.3. Edge-Level Task

For an edge-level task, we want to predict the property or presence of edges in a graph. One example of edge-level inference is in image scene understanding. Beyond identifying objects in an image, deep learning models can be used to predict the relationship between them. We can phrase this as an edge-level classification: given nodes that represent the objects in the image, we wish to predict which of these nodes share an edge or what the value of that edge is. If we wish to

discover connections between entities, we could consider the graph fully connected and based on their predicted value prune edges to arrive at a sparse graph.



In (b), above, the original image (a) has been segmented into five entities: each of the fighters, the referee, the audience and the mat. (C) shows the relationships between these entities.

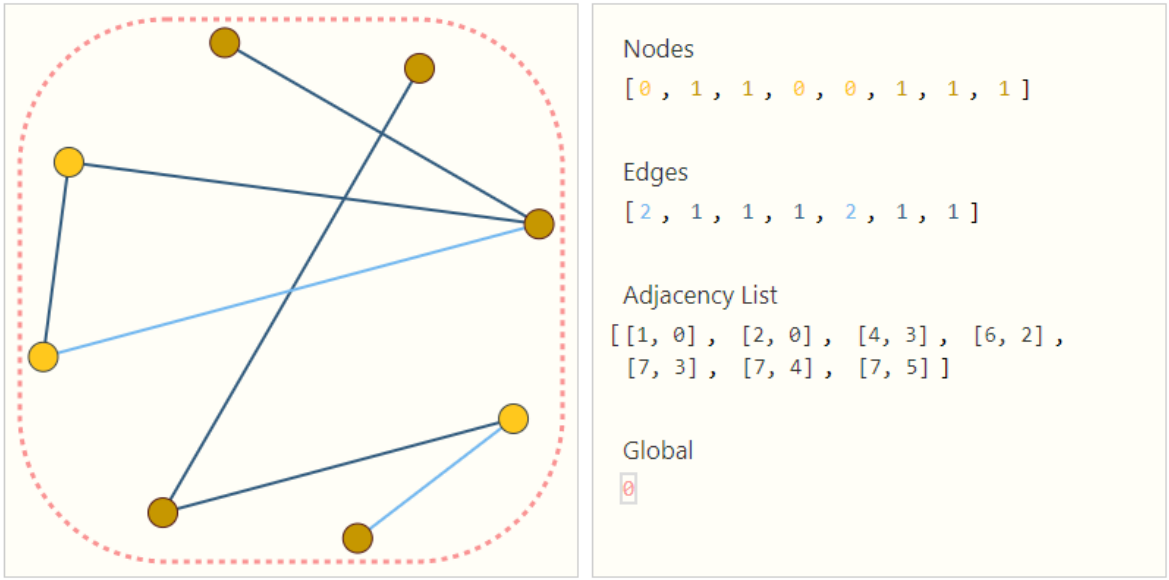
3. The Challenges of Using Graphs in Machine Learning

Machine learning models typically take rectangular or grid-like arrays as input. So, it's not immediately intuitive how to represent them in a format that is compatible with deep learning. Graphs have up to four types of information that we will potentially want to use to make predictions: nodes, edges, global-context and connectivity. The first three are relatively straightforward: for example, with nodes we can form a node feature matrix N by assigning each node an index i and storing the feature for $node_i$ in N. While these matrices have a variable number of examples, they can be processed without any special techniques.

However, representing a graph's connectivity is more complicated. Perhaps the most obvious choice would be to use an adjacency matrix, since this is easily tensorisable. However, this representation has a few drawbacks. From the example dataset table, we see the number of nodes in a graph can be on the order of millions, and the number of edges per node can be highly variable. Often, this leads to very sparse adjacency matrices (稀疏矩阵), which are space-inefficient.

Another problem is that there are many adjacency matrices that can encode the same connectivity, and there is no guarantee that these different matrices would produce the same result in a deep neural network (that is to say, they are not permutation invariant (排列不变))).

One elegant and memory-efficient way of representing sparse matrices is as **adjacency lists**. These describe the connectivity of edge e_k between nodes n_i and n_j as a tuple (i,j) in the k-th entry of an adjacency list. Since we expect the number of edges to be much lower than the number of entries for an adjacency matrix (n_{nodes}^2), we avoid computation and storage on the disconnected parts of the graph.



It should be noted that the figure uses scalar values per node/edge/global, but most practical tensor representations have vectors per graph attribute. Instead of a node tensor of size [$n*{nodes}$] we will be dealing with node tensors of size [$n*{nodes}$,${node}_{dim}$]. Same for the other graph attributes.

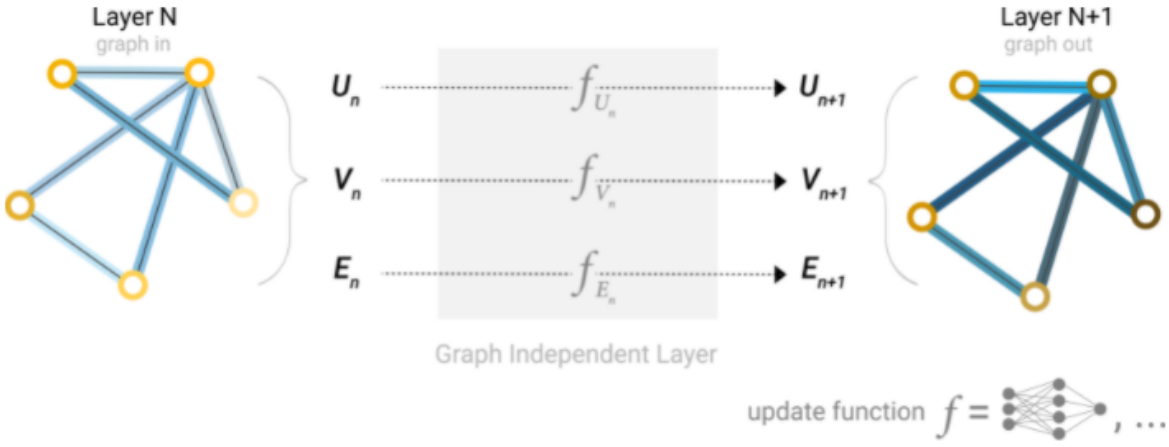
4. Graph Neural Networks

A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances). We're going to build GNNs using the "message passing neural network" framework proposed by Gilmer et al. using the Graph Nets architecture schematics introduced by Battaglia et al. GNNs adopt a "graph-in, graph-out" architecture meaning that these model types accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph.

4.1. The Simplest GNN

With the numerical representation of graphs that we've constructed above (with vectors instead of scalars), we are now ready to build a GNN. We will start with the simplest GNN architecture, one where we learn new embeddings for all graph attributes (nodes, edges, global), but where we do not yet use the connectivity of the graph.

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

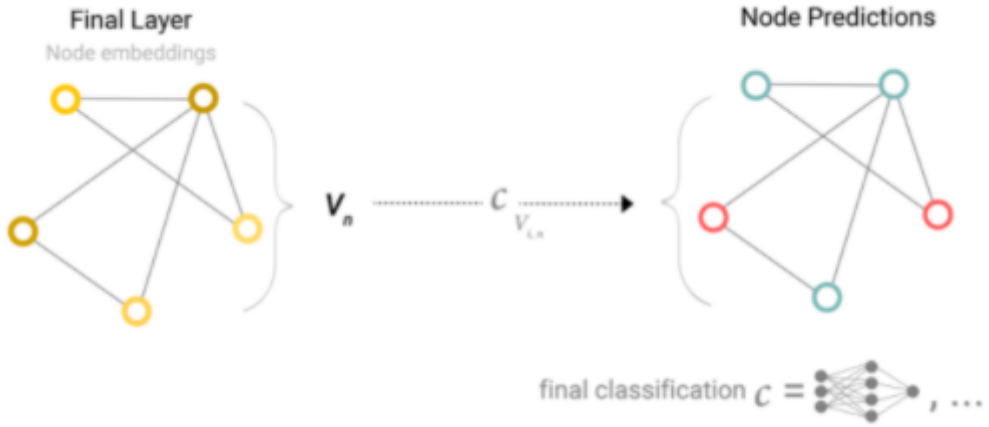


Because a GNN does not update the connectivity of the input graph, we can describe the output graph of a GNN with the same adjacency list and the same number of feature vectors as the input graph. But, the output graph has updated embeddings, since the GNN has updated each of the node, edge and global-context representations.

4.2. GNN Predictions by Pooling Information

We have built a simple GNN, but how do we make predictions in any of the tasks we described above?

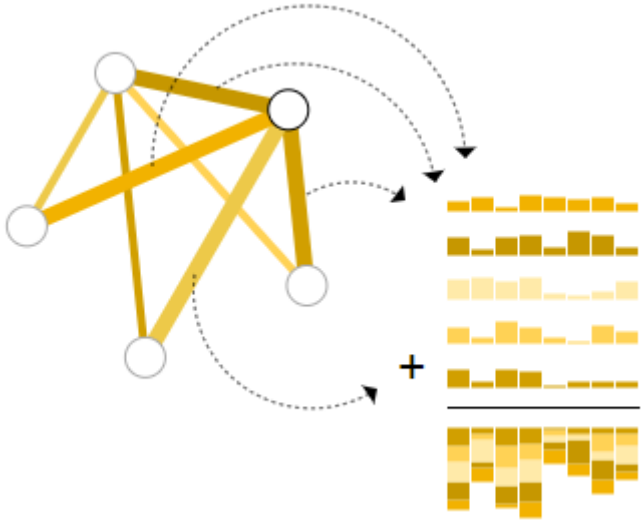
If the task is to make binary predictions on nodes, and the graph already contains node information, the approach is straightforward — for each node embedding, apply a linear classifier.



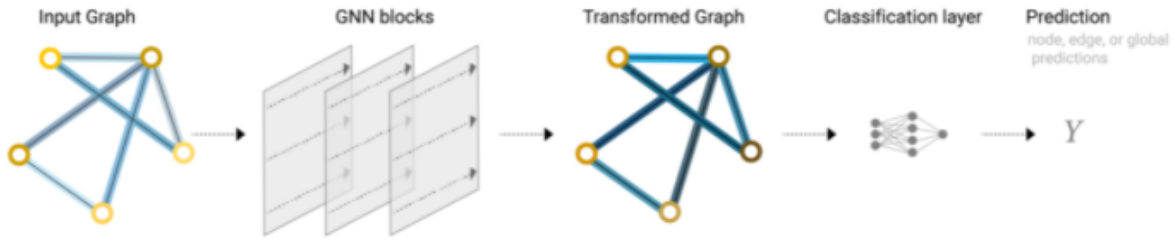
However, you might have information in the graph stored in edges, but no information in nodes, but still need to make predictions on nodes. We need a way to collect information from edges and give them to nodes for prediction. We can do this by pooling:

1. For each item to be pooled, gather each of their embeddings and concatenate them into a matrix.
2. The gathered embeddings are then aggregated, usually via a sum operation.

We represent the pooling operation by the letter ρ, and denote that we are gathering information from edges (and the graph-level information) to nodes as $\rho E_n\to V_n$.



Now we've demonstrated that we can build a simple GNN model, and make binary predictions by routing information between different parts of the graph.



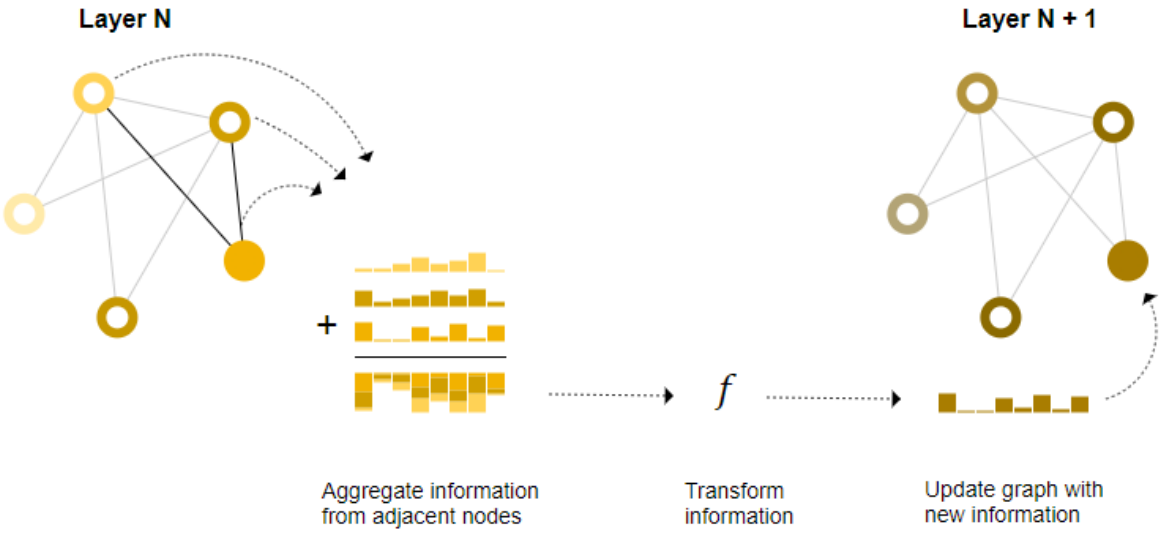
Note that in this simplest GNN formulation, we're not using the connectivity of the graph at all inside the GNN layer. Each node is processed independently, as is each edge, as well as the global context. We only use connectivity when pooling information for prediction.

4.3. Passing Messages between Parts of the Graph

We could make more sophisticated predictions by using pooling within the GNN layer, in order to make our learned embeddings aware of graph connectivity. We can do this using message passing, where neighboring nodes or edges exchange information and influence each other's updated embeddings.

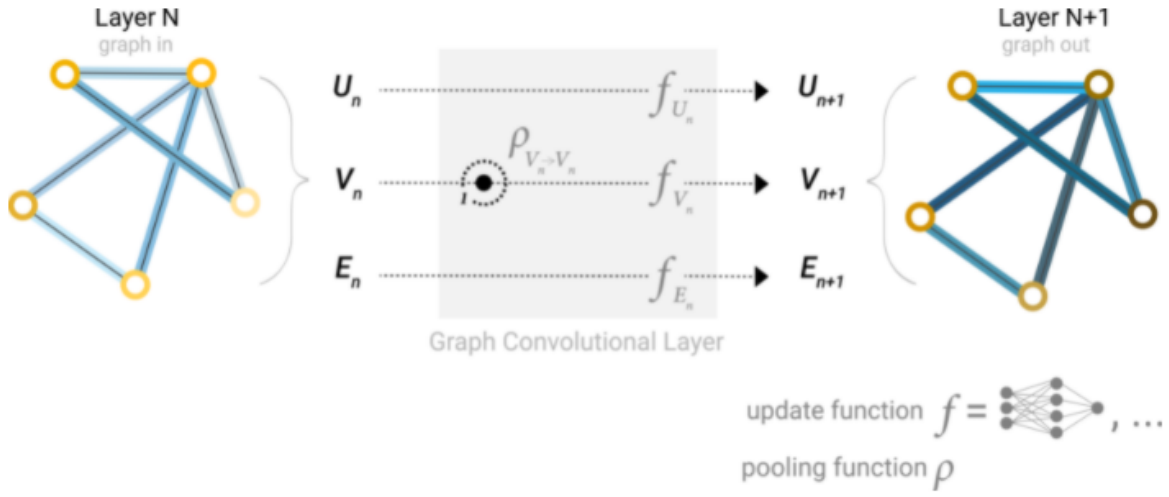
Message passing works in three steps:

1. For each node in the graph, gather all the neighboring node embeddings (or messages), which is the g function described above.
2. Aggregate all messages via an aggregate function (like sum).
3. All pooled messages are passed through an update function, usually a learned neural network.



This sequence of operations, when applied once, is the simplest type of message-passing GNN layer. By stacking message passing GNN layers together, a node can eventually incorporate information from across the entire graph: after three layers, a node has information about the nodes three steps away from it.

We can update our architecture diagram to include this new source of information for nodes:

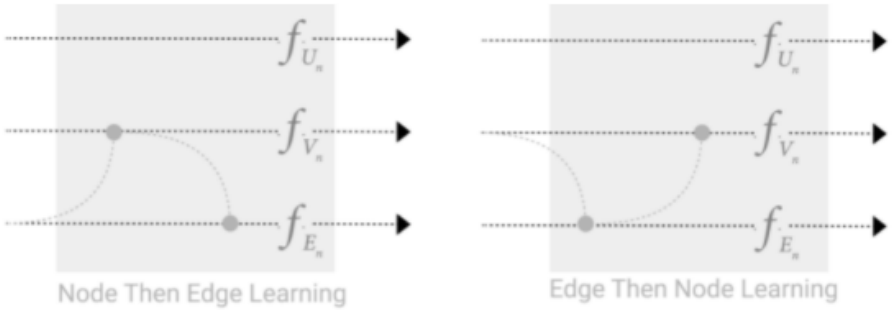


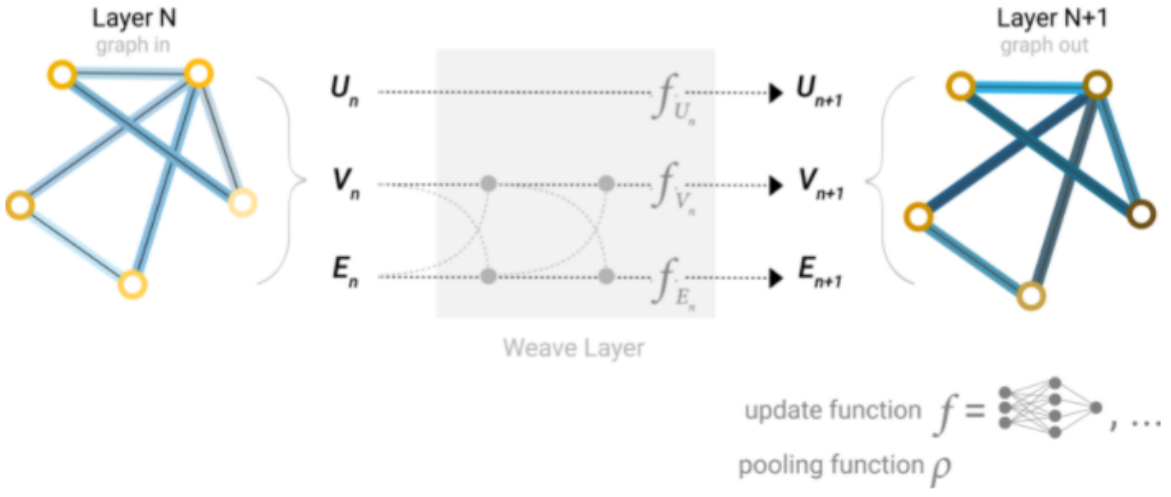
4.4. Learning Edge Representations

Our dataset does not always contain all types of information (node, edge, and global context). When we want to make a prediction on nodes, but our dataset only has edge information, we showed above how to use pooling to route information from edges to nodes, but only at the final prediction step of the model. We can share information between nodes and edges within the GNN layer using message passing.

We can incorporate the information from neighboring edges in the same way we used neighboring node information earlier, by first pooling the edge information, transforming it with an update function, and storing it.

However, the node and edge information stored in a graph are not necessarily the same size or shape, so it is not immediately clear how to combine them. One way is to learn a linear mapping from the space of edges to the space of nodes, and vice versa. Alternatively, one may concatenate them together before the update function.

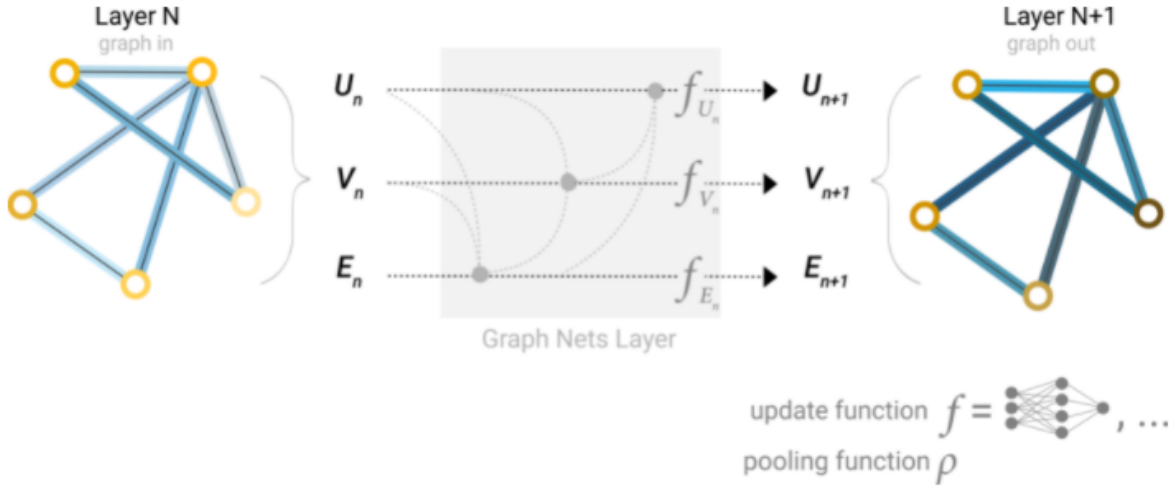




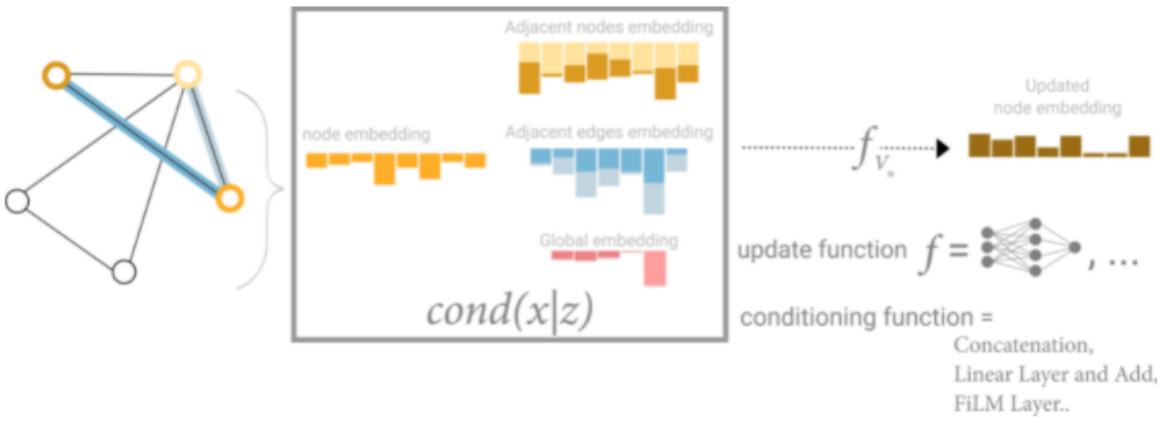
Which graph attributes we update and in which order we update them is one design decision when constructing GNNs. We could choose whether to update node embeddings before edge embeddings, or the other way around. This is an open area of research with a variety of solutions–for example we could update in a 'weave' fashion where we have four updated representations that get combined into new node and edge representations: node to node (linear), edge to edge (linear), node to edge (edge layer), edge to node (node layer).

4.5. Adding Global Representations

There is one flaw (瑕疵) with the networks we have described so far: nodes that are far away from each other in the graph may never be able to efficiently transfer information to one another, even if we apply message passing several times. One solution to this problem is by using the global representation of a graph (U) which is sometimes called a **master node** or context vector. This global context vector is connected to all other nodes and edges in the network, and can act as a bridge between them to pass information, building up a representation for the graph as a whole.



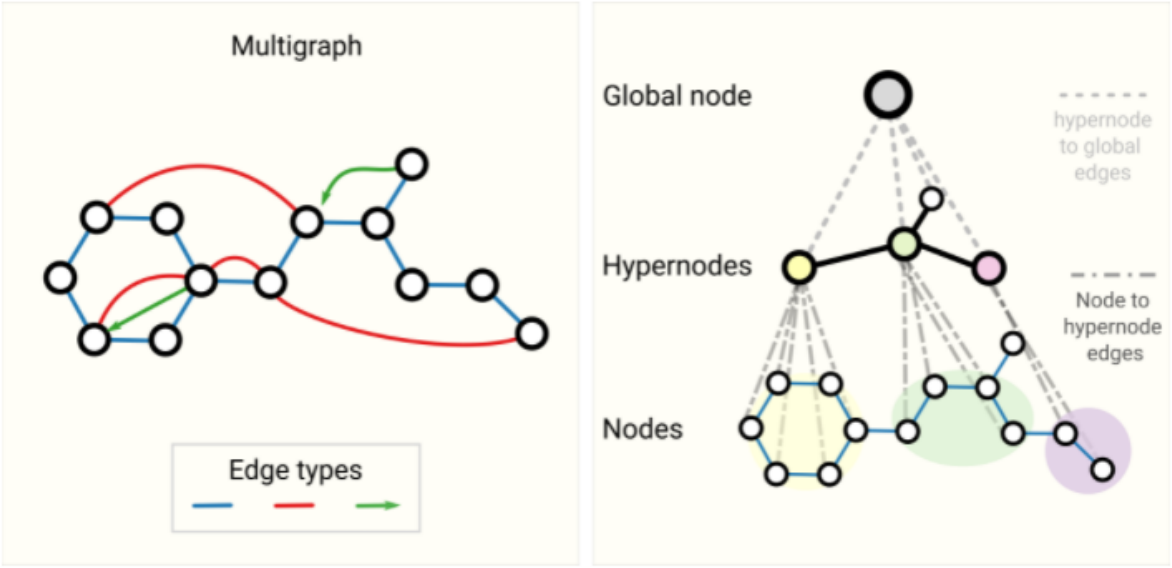
In this view all graph attributes have learned representations, so we can leverage them during pooling by conditioning the information of our attribute of interest with respect to the rest. For example, for one node we can consider information from neighboring nodes, connected edges and the global information. To condition the new node embedding on all these possible sources of information, we can simply concatenate them. Additionally we may also map them to the same space via a linear map and add them or apply a feature-wise modulation layer, which can be considered a type of featurize-wise attention mechanism.



5. Into the Weeds

5.1. Other Types of Graphs

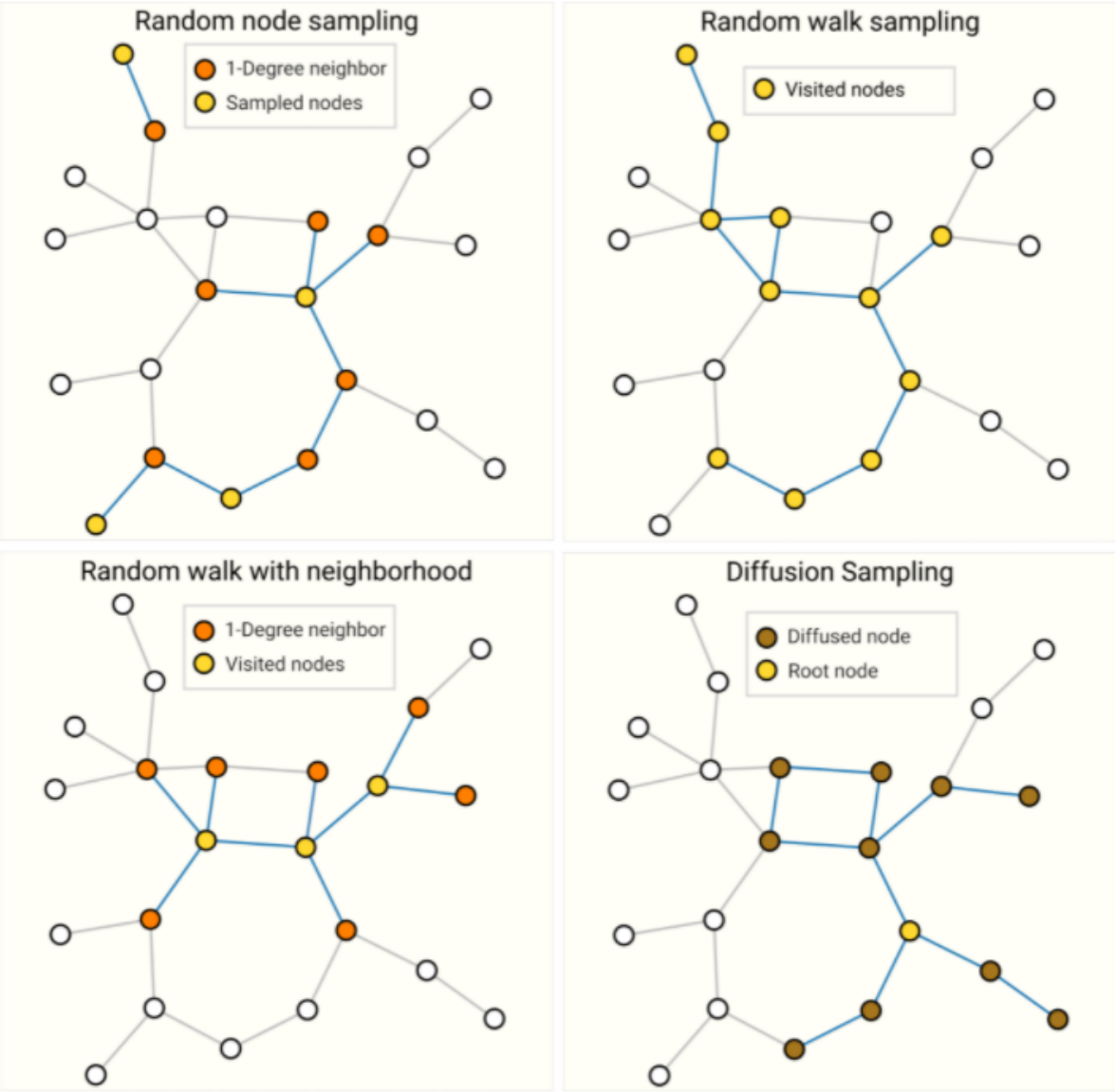
While we only described graphs with vectorized information for each attribute, graph structures are more flexible and can accommodate other types of information. Fortunately, the message passing framework is flexible enough that often adapting GNNs to more complex graph structures is about defining how information is passed and updated by new graph attributes.



Schematic of more complex graphs. On the left we have an example of a multigraph with three edge types, including a directed edge. On the right we have a three-level hierarchical graph, the intermediate level nodes are hypernodes.

5.2. Sampling Graphs and Batching in GNNs

A common practice for training neural networks is to update network parameters with gradients calculated on randomized constant size (batch size) subsets of the training data (mini-batches). This practice presents a challenge for graphs due to the variability in the number of nodes and edges adjacent to each other, meaning that we cannot have a constant batch size. The main idea for batching with graphs is to create subgraphs that preserve essential properties of the larger graph. This graph sampling operation is highly dependent on context and involves sub-selecting nodes and edges from a graph.



5.3. Comparing Aggregation Operations

Pooling information from neighboring nodes and edges is a critical step in any reasonably powerful GNN architecture. Selecting and designing optimal aggregation operations is an open research topic. A desirable property of an aggregation operation is that similar inputs provide similar aggregated outputs, and vice-versa. Some very simple candidate permutation-invariant operations are sum, mean, and max. Summary statistics like variance also work. All of these take a variable number of inputs, and provide an output that is the same, no matter the input ordering.

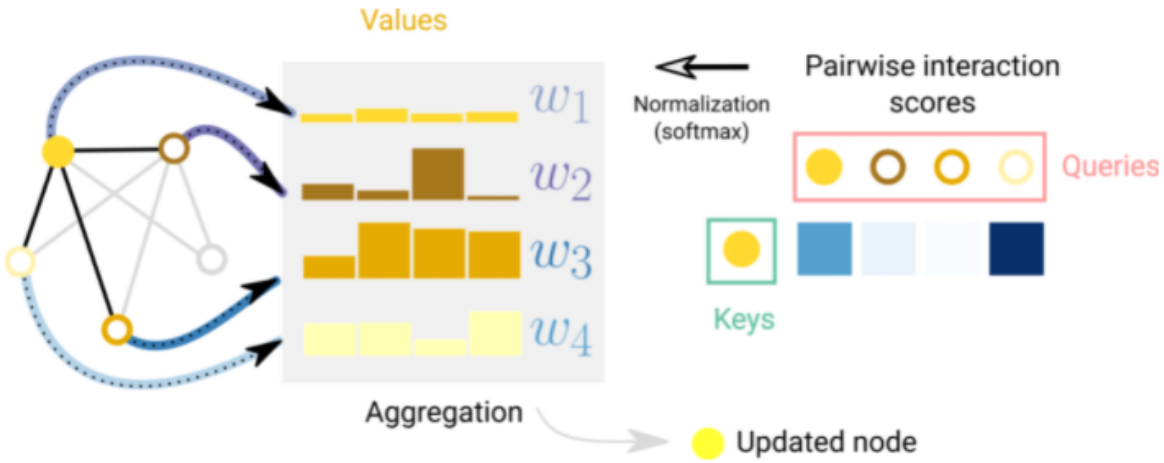
There is no operation that is uniformly the best choice. The mean operation can be useful when nodes have a highly-variable number of neighbors or you need a normalized view of the features of a local neighborhood. The max operation can be useful when you want to highlight single salient features in local neighborhoods. Sum provides a balance between these two, by providing a snapshot of the local distribution of features, but because it is not normalized, can also highlight outliers. In practice, sum is commonly used.

5.4. Graph Attention Networks

Another way of communicating information between graph attributes is via attention. For example, when we consider the sum-aggregation of a node and its 1-degree neighboring nodes we could also consider using a weighted sum. The challenge then is to associate weights in a permutation invariant fashion. One approach is to consider a scalar scoring function that assigns weights based on pairs of nodes ($f(node_i,node_j)$). In this case, the scoring function can be

interpreted as a function that measures how relevant a neighboring node is in relation to the center node. Weights can be normalized, for example with a softmax function to focus most of the weight on a neighbor most relevant for a node in relation to a task. This concept is the basis of Graph Attention Networks (GAT) and Set Transformers.

A common scoring function is the inner product and nodes are often transformed before scoring into query and key vectors via a linear map to increase the expressivity of the scoring mechanism. Additionally for interpretability, the scoring weights can be used as a measure of the importance of an edge in relation to a task.



Schematic of attention over one node with respect to it's adjacent nodes. For each edge an interaction score is computed, normalized and used to weight node embeddings.

Additionally, transformers can be viewed as GNNs with an attention mechanism. Under this view, the transformer models several elements (i.e. character tokens) as nodes in a fully connected graph and the attention mechanism is assigning edge embeddings to each node-pair which are used to compute attention weights. The difference lies in the assumed pattern of connectivity between entities, a GNN is assuming a sparse pattern and the Transformer is modelling all connections.

6. Final Thoughts

Graphs are a powerful and rich structured data type that have strengths and challenges that are very different from those of images and text. In this article, we have outlined some of the milestones that researchers have come up with in building neural network based models that process graphs. We have walked through some of the important design choices that must be made when using these architectures, and hopefully the GNN playground can give an intuition on what the empirical results of these design choices are. The success of GNNs in recent years creates a great opportunity for a wide range of new problems, and we are excited to see what the field will bring.