**Graph Machine Learning — Intro**

[[Prathamesh Sonawane](https://medium.com/@prathameshsonawane?source=post_page-----4e35098f5176--------------------------------)](https://medium.com/@prathameshsonawane?source=post_page-----4e35098f5176--------------------------------)

[Prathamesh Sonawane](https://medium.com/@prathameshsonawane?source=post_page-----4e35098f5176--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F41dc5a50d2ef&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40prathameshsonawane%2Fgraph-machine-learning-intro-4e35098f5176&user=Prathamesh+Sonawane&userId=41dc5a50d2ef&source=post_page-41dc5a50d2ef----4e35098f5176---------------------post_header-----------)

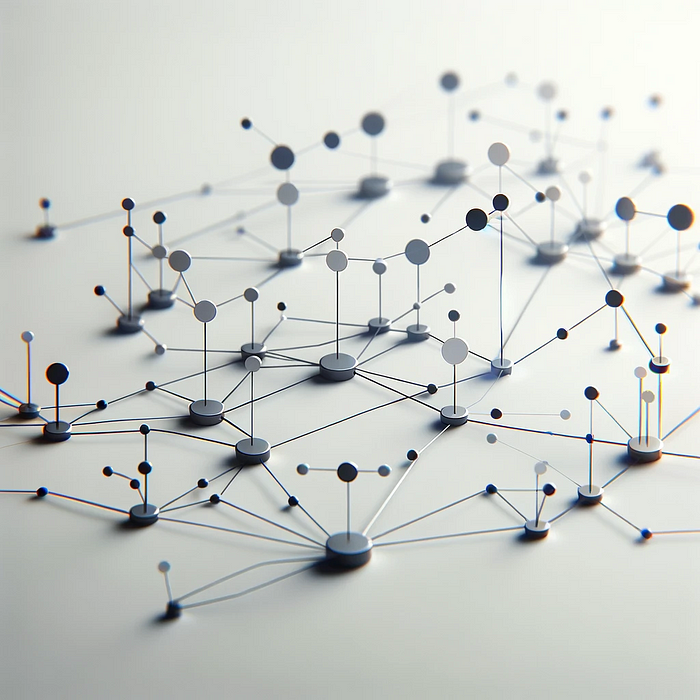
10 min read

·

Dec 3, 2023

27

I am writing this in order to force myself to learn this topic properly. If you end up reading this, pl lmk if you find an inconsistencies/mistakes/skipped topics.



src: Dall-E

**What is Graph ML?**

Its pretty much in the name, Machine learning for Graphs. Common use cases are the study of Social Networks, Drug Discovery, Particle Physics, Traffic and Route Optimization, Knowledge Graphs, Financial transaction networks, Chemistry (atoms & molecules), and so on.

* Uber Eats uses GNNs to recommend food to you.
* Pinterest uses [PinSage GNN](https://medium.com/pinterest-engineering/pinsage-a-new-graph-convolutional-neural-network-for-web-scale-recommender-systems-88795a107f48) to curate your feed.
* X uses GNNs to detect fake news.
* [Icecube labs](https://icecube.wisc.edu/science/icecube/#:~:text=IceCube%2C%20the%20South%20Pole%20neutrino,depth%20of%20about%202%2C500%20meters.) use GNNs to detect neutrino particles on the South Pole.
* GNNs are used in drug discovery to beat [Cancer](https://www.nature.com/articles/s41598-019-45349-y)!
* Also other exciting apps like [predicting quantum properties of molecules](https://arxiv.org/abs/1704.01212), [predicting molecule X’s chemical properties](https://arxiv.org/abs/1509.09292), etc.

NLP: *I manipulate text.* CV: *I manipulate images.*Graph ML: *hold my beer!* 🍺

Jokes aside, there are some nice parallels to be drawn between CV/NLP and Graph ML, which I’ll discuss later in this blog.

**Properties of a Graph**

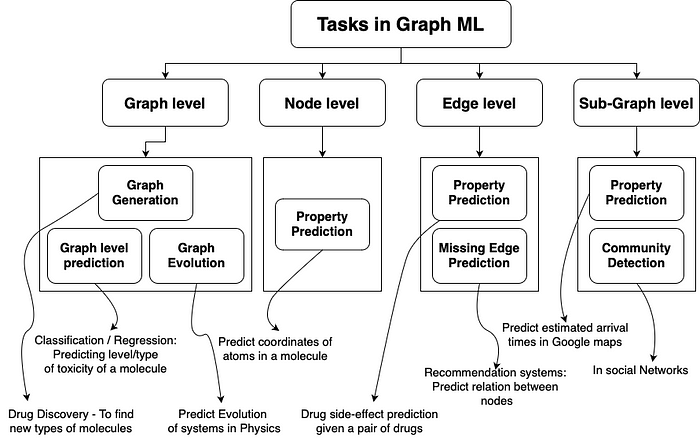
Most things remain the same. You have nodes and edges that make up a graph. There is a whole field dedicated to graphs ([graph theory](https://en.wikipedia.org/wiki/Graph_theory)) but you wont really need that.

Still, some of the properties of Graphs are:

* Directed/Undirected Graphs
* Heterogeneous/Homogeneous Graphs  
  -**Heterogeneous graphs** have ‘typed’ nodes/edges. Typed means that you can have different types of nodes/edges in a graph. Think types of bonds and atoms in molecules.  
  - **Homogeneous graphs** are pretty much normal graphs with a single type of edge & node.
* Graph with/without self-edges.
* Graphs with/without loops.
* Multi/Hyper graphs.

Before you begin, you must first consider the best characterization depending on your problem statement (homogeneous/heterogeneous, directed/undirected, etc)

**What are the Tasks in Graph Machine learning?**



Tasks in Graph ML

**How are Graphs represented?**

Pretty straightforward.

* Set of all edges
* Adjacency list
* Adjacency Matrix (tends to be sparse, so we can make a Laplacian matrix from it if needed.)

… or other ways, but these are the most common ones.

*Note* — Even though we have managed to represent them as ordered objects we cannot treat them as we treat other ordered objects in ML. Intrinsically, they need to be treated at the unordered objects that they are.

Apart from this, each node in the graph has a feature vector which basically describes that node. We can also have feature vectors for edges.

**Pre-[Graph Neural Networks]!**

*Before we dive into Graph Neural Networks (GNNs) let's briefly look at techniques that were used before they came into existence. It helps for more context and allows me to appreciate the current methods.*

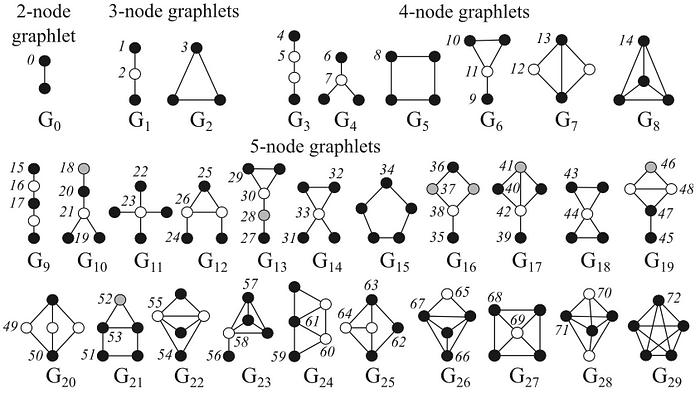
Before GNNs, graphs were represented by arbitrarily extracting certain features from them, and using those to perform whatever task it is that we want to perform.

*It’s quite similar to what I thought Graph ML must be like when someone had asked me a GNN problem for the first time. My solution was to extract features like connections, node types, number of loops, etc based on my intuition of what the model may find useful, put them in a table, and train a normal NN to predict the output. Of course, retrospectively, that was quite naive.*

Nevertheless, lets go through some of the common features —

**Node level features:**

* **Node Type**
* **Node Degree:**Number of direct neighbors of a node
* **Clustering Coefficient:**Measure of how connected the node neighbors are.
* **Centrality:**Measures the node importance in the graph. It can be computed recursively by summing the centrality of each node’s neighbors until convergence, or through shortest distance measures between nodes.
* **Graphlet Degree Vectors:**Amount of different graphlets connected to a given node. Graphlets different mini-graphs you can create given a certain number of nodes.



The 2- to 5-node graphlets and their automorphism orbits(ignore) ([Pržulj, 2007](https://www.researchgate.net/publication/272888925_L-GRAAL_Lagrangian_graphlet-based_network_aligner" \t "_blank))

**Edge level features**

* Edge label (Edge type)
* Shortest distance between 2 nodes.
* Number of common neighbors.
* Katz index — number of possible walks of up to a certain length between two nodes

**Graph level features**

* **Graphlet counts:**Total graphlet counts for each n number of nodes — computationally v. expensive.
* **Graph similarity:  
  -**Can be done using node degree and bag of nodes.  
  - [Walk based methods](https://en.wikipedia.org/wiki/Random_walk) — Using the probability of visiting a node j from a node i on a random walk to define similarity metrics. Ex: [Node2Vec](https://snap.stanford.edu/node2vec/)

However, all of these things pose a lot of limitations which will become more evident in the next section as we begin to understand GNNs.

**Before GNNs lets look at how Embeddings are made**

* [DeepWalk](https://arxiv.org/abs/1403.6652) — it’s reduced to Word2Vec if you do the following: sample “sentences” from the graph by doing random [walks](https://www.geeksforgeeks.org/mathematics-walks-trails-paths-cycles-and-circuits-in-graph/#:~:text=A%20walk%20is%20a%20sequence,can%20be%20open%20or%20closed.).
* [Node2Vec](https://cs.stanford.edu/~jure/pubs/node2vec-kdd16.pdf) — literally the same idea as DeepWalk with the additional control of how you’ll sample from your graph ([BFS](https://en.wikipedia.org/wiki/Breadth-first_search)/[DFS](https://en.wikipedia.org/wiki/Depth-first_search)). If you give more weight to BFS the tightly coupled nodes will have similar embeddings.

A brief clarification on DeepWalk — if you treat the nodes in the graph as words, then if you do a random walk you’re basically sampling a random sentence from your graph.

Once you have the sentence you can do the Word2Vec magic and learn your node embeddings such that those nodes that tend to be close in your sentences have similar embeddings.

**Graph Neural Networks**

How do we build a Neural Network that support Graphs as is?

Let’s formalize the requirements a bit.

Given f(x) is the NN, G is the Graph, P(x) is a random permutation function, we need the NN to satisfy 2 properties.

* **Permutation invariance:***f(P(G))=f(G)*This means that, the output of a graph and any of its valid permutations will be exactly the same of passed through a GNN.
* **Permutation Equivariance:***P(f(G))=f(P(G))*This means that, it doesn’t matter if apply the permutation before of after passing through the GNN. Output remains the same.

Typical NNs do not satisfy these properties, hence we had to create a separate category of NNs, [Graph Neural Networks](https://ieeexplore.ieee.org/abstract/document/1517930).

**Different ways of training GNNs!**

This concept is unique to GNNs (afaik). There are 2 types of learning settings — transductive and inductive.

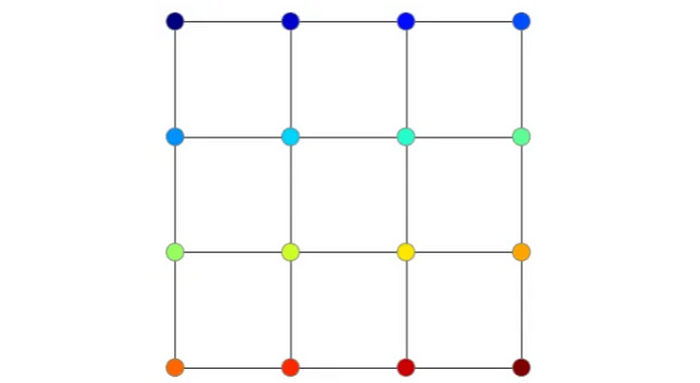
1. Transductive learning:  
   We split a single graph’s nodes into training, validation and test nodes and then train the model on this. During training the model can access the validation nodes if it reaches those during multi-layered aggregation, this is allowed. It is allowed to use the feature vectors but not the labels of those nodes. More on this later.
2. Inductive learning:  
   This is more along the lines of what we are used to in ML. We have separate sets of graphs for train, validation and test sets. (can be multiple graphs in a set.)

**Now onto the actual meat, how do GNNs actually work?**

GNNs are networks that consume Graphs as is. GNNs work by aggregating information from a node’s neighbors (and the neighbors’ neighbors, and so on) to compute a representation (embedding) of each node or edge. These embeddings capture both the features of the nodes and the structure of the graph.

A GNN layer represents a node as an combination of its neighbors (aggregation), node’s own properties (message passing) and lastly an activation to introduce non-linearity in the network. You can draw parallels here between how you need to give positional & word embeddings in a normal Transformer.

Intuition — A CNN can be seen as a generalization of a GNN, with fixed neighbor sizes (sliding filter) and ordering (permutation unequivariance) or a Transformer without positional embeddings can be seen as a GNN.



An example of a 4x4 pixel image — we can treat an image as a grid graph. pixel values become graph signals and you acn imagine just passing a 2x2 filter over this. Aggregation becomes pooling.

**Aggregation Techniques**

* [Graph Convolutional Networks](https://tkipf.github.io/graph-convolutional-networks/) (GCNs)  
  Average the normalized representations of a node’s neighbors.
* [GraphSage](https://snap.stanford.edu/graphsage/)  
  Basically selects only a certain random sub-sample of neighbors for aggregation at each layer. This reduces computation costs.
* [Graph Isomorphism Networks](https://arxiv.org/pdf/1810.00826v3.pdf) (GINs)  
  The weights for the neighbors are determined by passing them through a fully connected NN.
* [Graph Attention Networks](https://petar-v.com/GAT/) (GAT)  
  Assuming you know attention (transformers), this essentially weighs the neighbors based on their importance given by attention. Its like GINs but this is more computationally intensive. It has its pros and cons.

Note — Some aggregation techniques like mean/max can fail in certain cases like if a node has the neighbors (-1, 1) and another node has the neighbors (-2, 0, 2, 0) both will get treated as the equals.

In conclusion, once you have feature vectors from your neighbors at your disposal,

1. You somehow transform them (maybe a linear projection)
2. You somehow aggregate them (maybe weighing them with attention coefficients, voilà, we get GAT )
3. You update the feature vector (somehow) of the current node by combining its (transformed) feature vector with the aggregated neighborhood representation and activation.

**Over-smoothing problem in GNNs**

This is unique to GNNs but it’s in the same class as overfitting, underfitting, etc.

In the first layer of a GNN you aggregate over the neighbors of a node. In the next layer you again aggregate over its neighbors, but this time the neighbors are a aggregation of their neighbors. This means that at every subsequent layer a node get more visibility into the graph (imagine CNNs its quite the same).

Now we can say that at every layer the radius of circle around any particular node increases by 1. After n layers the radius will be n. If n becomes larger than the max radius of the network then all of the nodes start becoming indistinguiable from each other. (because each node is then an aggregation of all of the nodes in the network!)

**How do you solve this?**

* Analyze the graph diameter and shape first, then determine the network depth based on that.
* add skip connections. (throwback to vanishing gradients. Same but different.)
* Dropping Edges or Nodes randomly (like dropout)
* Using different (more complex) aggregation functions — attention, GINs, etc.
* Add self-loops (like RNNs)
* Add non-message passing layers

It’s pretty evident that we can draw many parallels between what we are doing here and what we do to solve overfitting in normal NNs.

**Graph Transformers**

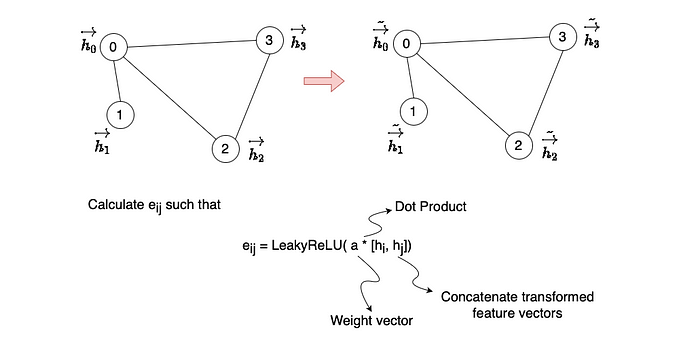
You may remember that we transformers had an inherent flaw compared to its predecessors (LSTMs, etc). It’s constructed in a way that it cannot determine the relative positions of words in a sentence. We need to give positional embeddings for each word for it to understand the order of words.

In GNNs this ‘bug’ becomes a feature. We get permutation invariance by not passing positional embeddings. This is a hot research topic right now ([survey](https://github.com/ChandlerBang/awesome-graph-transformer)).

**Graph Attention Networks**

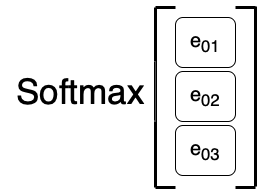
For this I’ll be looking at this [paper](https://arxiv.org/pdf/1710.10903.pdf)

* Each node in the graph will have a feature vector h. Let’s say we transform all feature vectors by multiplying them with certain separate weights (W) and get transformed feature vectors for each node (h^{tilde}).

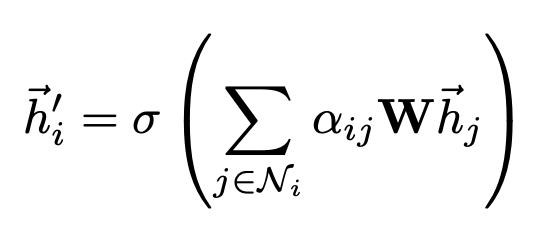


Transformation and calculating eij

* So for example lets do this for h0. We’ll choose all neighbours of h0 and calculate eij. Thus we will get e01, e02, e03. For these 3 values we’ll do softmax.

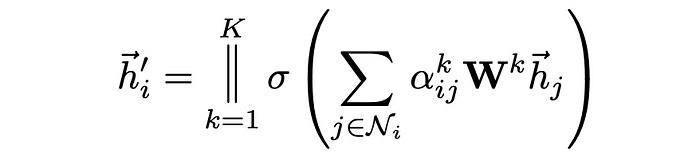


* Softmax gives you weights for each of the neighbors of ‘0' node. We multiply this with the tranformed feature vectors, sum over them and apply a non linearity. The equation is given below (alpha is the softmax o/p).

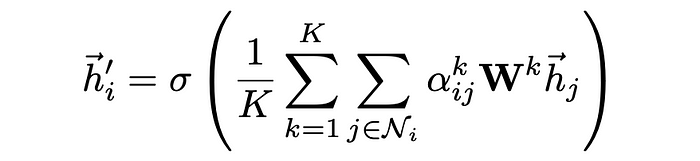


Self-attention in Graph Attention Networks

* Now this is a single attention head. We can do this for multiple heads and then concatenate the outputs exactly like multi-headed attention in LLMs. Formula below.

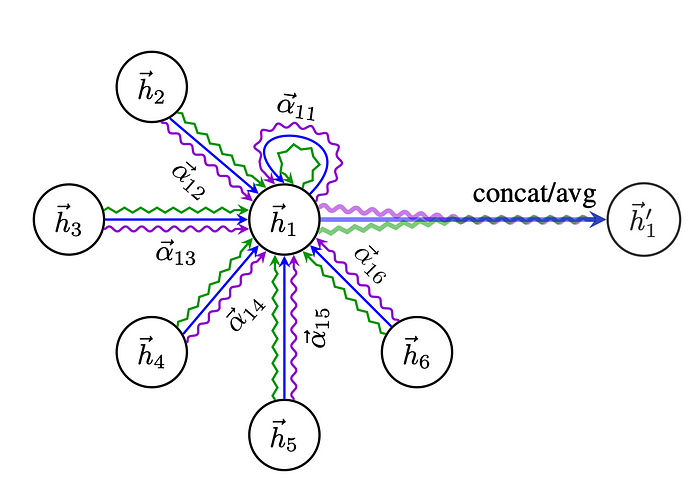


Multi-headed self-attention in Graph Attention Networks



For last layer they don’t concatenate instead we average the results of multiple heads before passing through the non-linearity (which will probably also be different for the last layer.)

* A diagram for the whole process would look like this. Colored squiggly like are multiple attention heads.



An illustration of multi- head attention (with K = 3 heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain the next h1'

* Lastly, the authors also use dropout, meaning that each node at every iteration will see a stochastically sampled neighborhood.
* The authors tried using q, k, v matrices as it is done in traditional attention but that led to overfitting. Instead having just 1 matrix as shown above gave better results.

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —  
(I’ll continue to update this as I learn more!)

Note — I dont aim on making this blog into the goto resource for Graph ML. In fact I’m not trying to make this a resource at all. Although it may work as a resource for me but I’ve realized that without the other things that I am looking at this is starting to become a hot mess of half baked concepts. I’ll try to as clear as possible while maintaining my sanity lol.