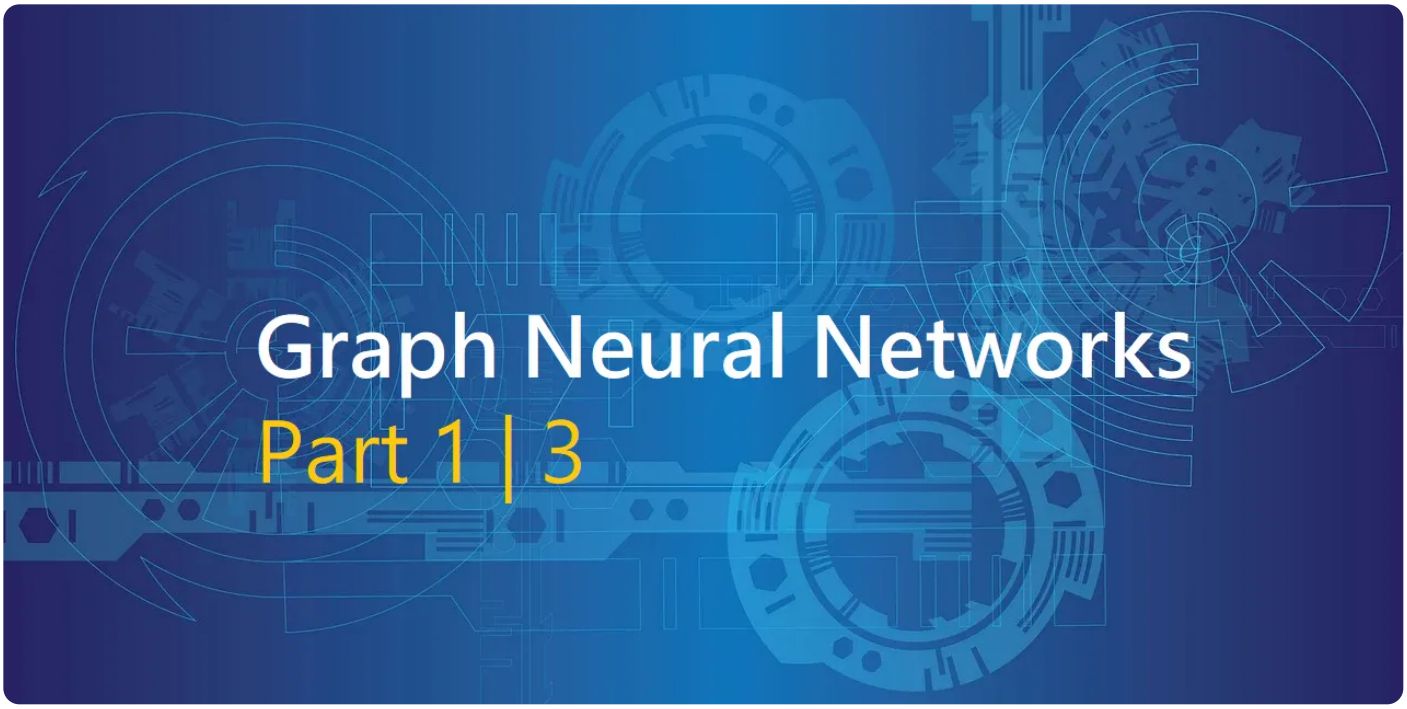
September 1, 2020

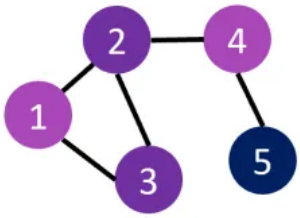
Understanding Graph Neural Networks | Part 1/3



Hello and welcome to this introduction to Graph Neural Networks! This is a short three-part series, which captures the most important things you need to know about Graph Neural Networks (GNNs). The last part shows you how you can build your own GNNs using Pytorch Geometric. Let's get started! 🙂

Sidenote: If you prefer watching videos, you can also find this series on YouTube.

Graphs



G = (V,E)

	V1	V2	...
V1	0	1	...
V2	1	0	...
V3	1	1	...
...	...	...	...

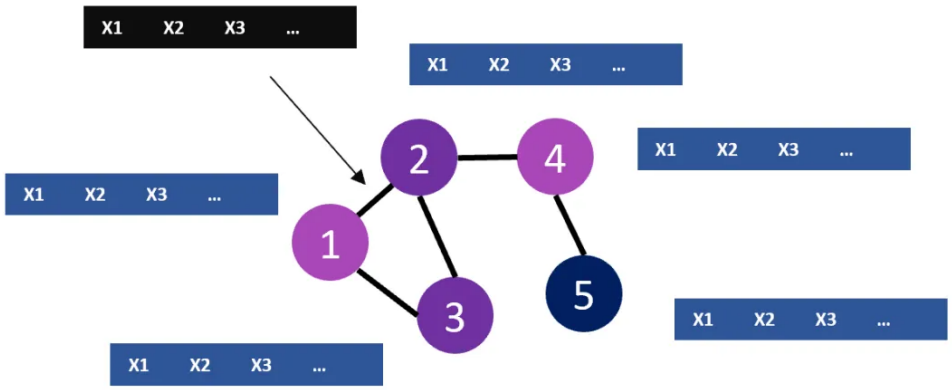
Adjacency matrix

Before talking about Graph Neural Networks, we should first talk about the input for these models: graph data.

Graph data is pretty simple. You have a set of nodes (or vertices = V) and you have edges (= E) between those nodes. The information about the connections in a graph is usually represented by adjacency matrices (or sometimes adjacency lists).

The adjacency matrix is symmetrical and lists all nodes along the rows and columns. If two nodes are connected in a graph, the adjacency matrix will have a 1 at the corresponding position, otherwise, for disconnected vertices, there is a 0.

Finally, the nodes or edges can have further properties – this means more specific information about the node/edge, as illustrated by the bars in the following sketch.



G = (V,E)

	V1	V2	...
V1	0	1	...
V2	1	0	...
V3	1	1	...
...	...	...	...

Adjacency matrix

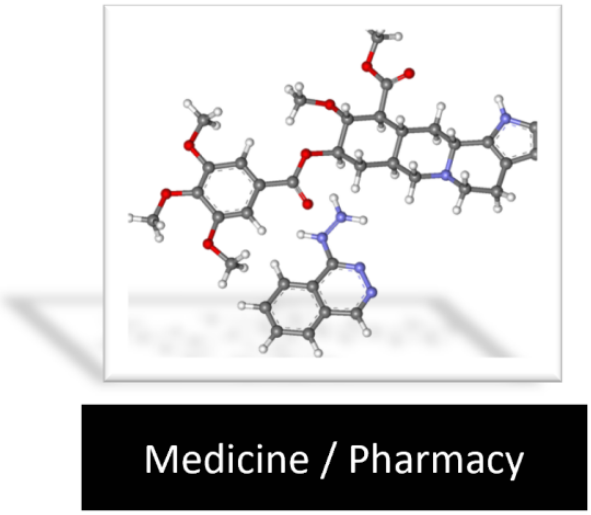
Those additional properties are indicated by the bars next to the vertices (blue) or the bar which is annotated on the edge (black).

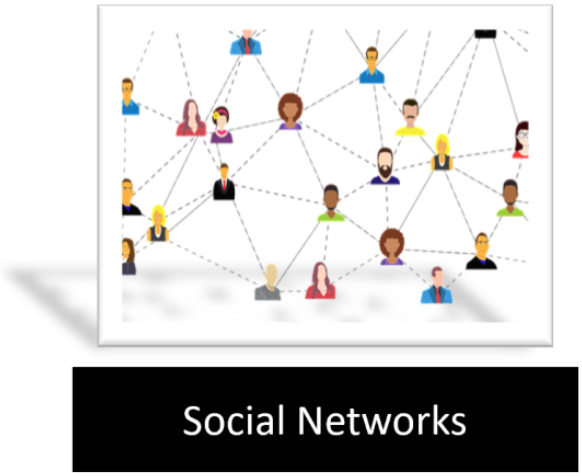
For example the blue bars could stand for properties about atoms in a molecule and the black bars represent the bond type in this atom. These additional attributes are usually called **node features** and **edge features** respectively.

And that's all you need to know in the following about *graph data*! 🙂

Graph Neural Networks operate on Graph Data

First of all, let's clarify why we even need Graph Neural Networks. GNNs are capable of handling graph-structured data. You might have already worked with tabular data, image data or text data. Graph data is also quite common in many areas. The following image shows just some examples.





Examples for Graph Data

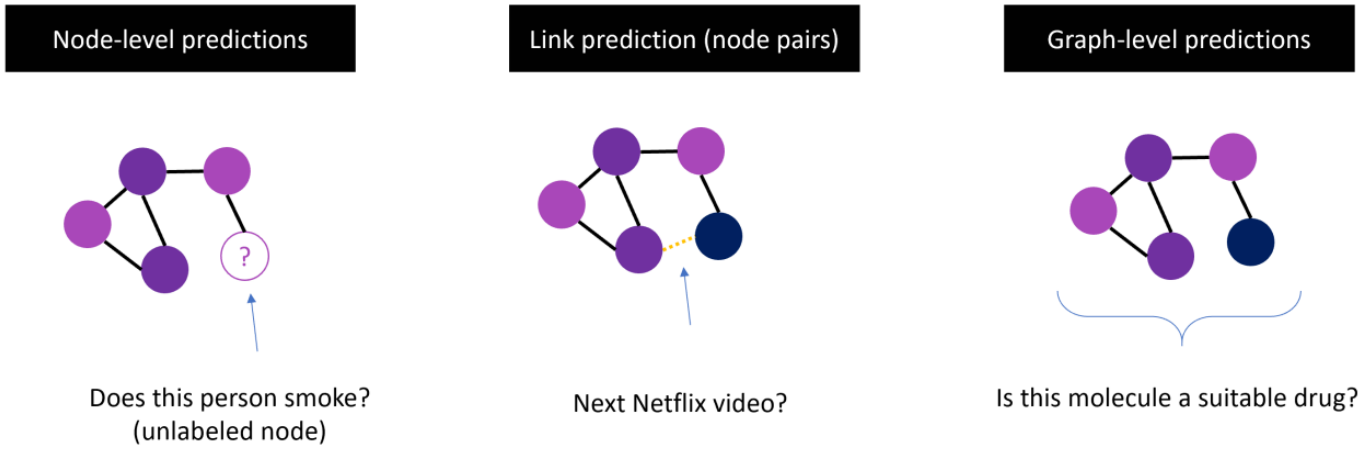
For instance molecules are usually modeled as a graph consisting of atoms (nodes) and edges (bonds). These graphs can have almost any shape. Another interesting example are large knowledge graphs, which are usually utilized for Recommender Systems by E-Commerce companies such as Amazon or Netflix. Besides these examples, there are generally many areas where you can find graph data, such as 3D Computer Games, IT Networks, Social Networks, Fraud Detection, Sports, and many more…

Graph Data is nice. **But what can you do with it *in a Machine Learning sense*?**

Just like any other Machine Learning task, you can perform different kinds of forecasts based on graph data. First of all, you can make **node-level predictions**. This means you have *unlabeled* nodes and try to predict certain properties based on the other nodes and their connections (edges). For example predicting whether a person is smoking or not (node classification), based on the information of the closest friends in a social network.

Another common use-case is to predict *links* in graphs (**link prediction**, **edge-level prediction**). This means you predict if node X and node Y will have a connection “in the future”. For instance you can forecast which movies will be most likely watched by a person on Netflix. Additionally, you can not only make a binary decision (yes / no), but also include *what kind of* connection it is.

Finally, you can of course also use the whole graph as input (**graph-level predictions**) and perform predictions based on the entire representation. This is usually done in medicine, when predicting the performance / suitability of a new molecule as a drug (this is what we will do in part 3 of this series ;-)). The pictures below summarize the most important Machine Learning tasks on graph data.



How to handle Graph Data

If you are familiar with neural networks, you know that they require a **fixed-size** input vector, as they have a fixed number of neurons in the first layer. **Before continuing quickly thing about this:**

> **How would YOU represent graph data with nodes and edges as an input for a neural network?**

If you think about feeding in the adjacency matrix, or a vector of nodes, you will unfortunately encounter a couple of issues with your approach.

Difficulty 1: Size and Shape of Graphs

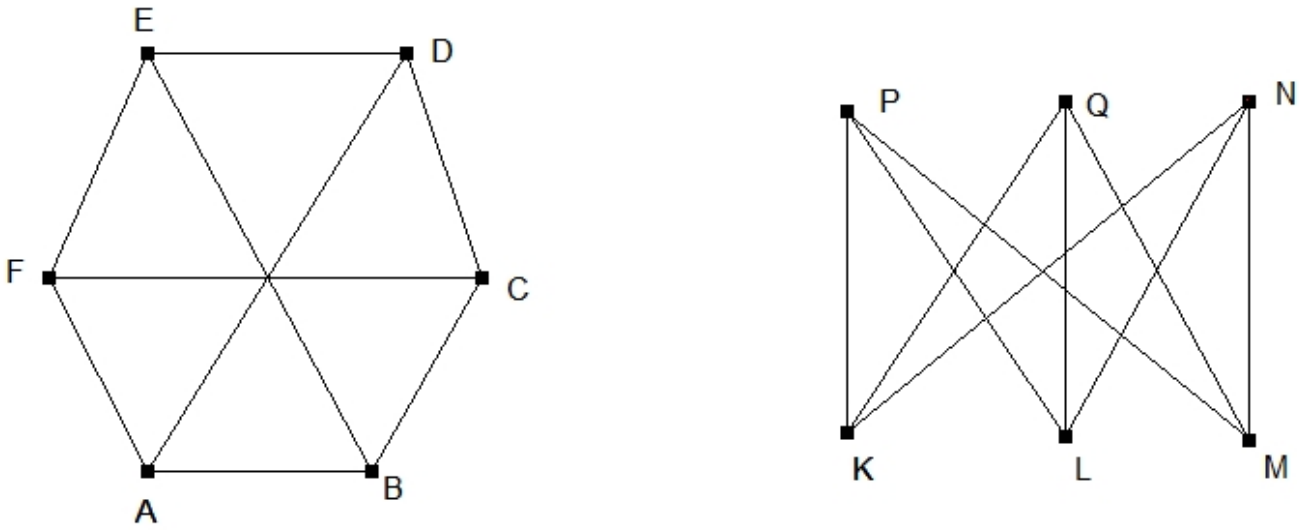
Graphs come in many different shapes and sizes. We cannot obtain a fixed-size vector simply by using the available nodes and edges. One graph might have 120 nodes and 300 edges, another one might come with 45 nodes and 70 edges. For images with different sizes, you could simply apply resizing or cropping, so that for example all images are 500 x 500 pixles. For graphs however, such methods are not applicable – *how would you resize a graph with a different number of nodes or edges*?

Therefore, we need an algorithm that compresses the information into a fixed-size vector (an *embedding*), independent of the number of nodes and edges of the graph.

You might come up with the idea to simply calculate hand-crafted features, e.g. by combining the adjacency matrix with the distance of nodes and their properties. However, there are still some issues with manual approaches, especially the difficulty that is presented in the following (Isomorphism) can usually not be tackled with such methods. You can read more about hand-crafted features here.

Difficulty 2: Isomorphism

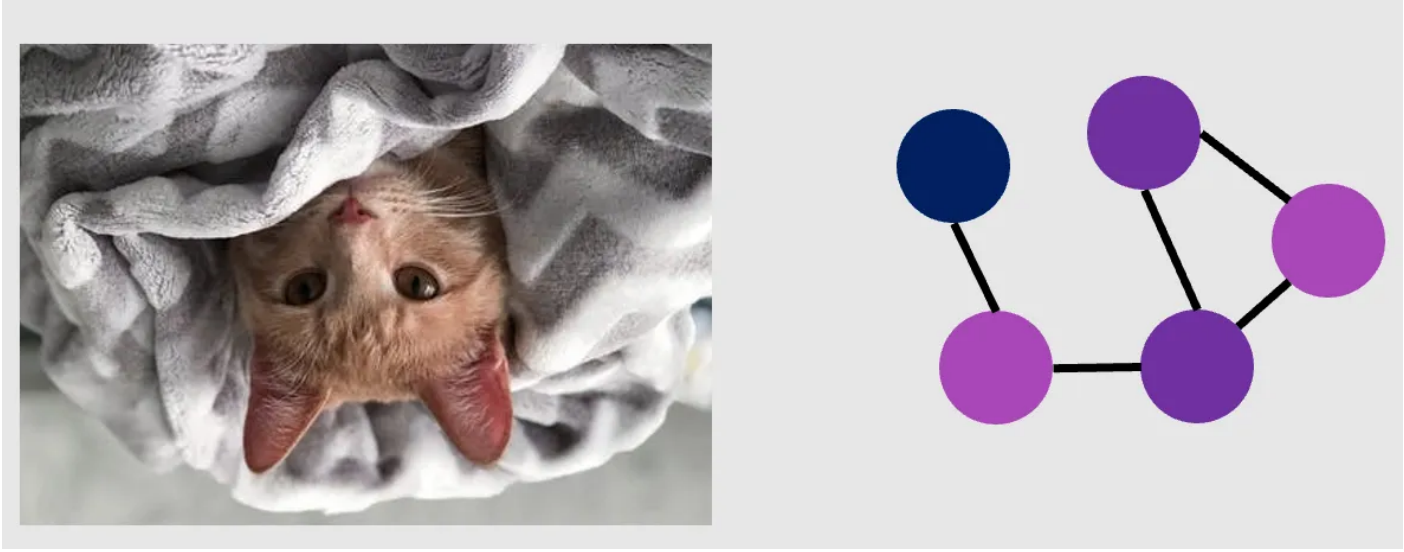
A special property of some graphs is called **Isomorphism** and simply means that graph structures, that lie differently in the space, might still represent **the same graph**. An extreme example of this is the following (source):



These two graphs have **exactly** the same nodes and edges. Even though, they look differently, and the order of nodes might be different, they still represent the same object. Now if you change the ordering, this will affect the adjacency matrix, and eventually your input for the neural network will change. However, we look for a representation, that is **identical** for both of the illustrated graphs.

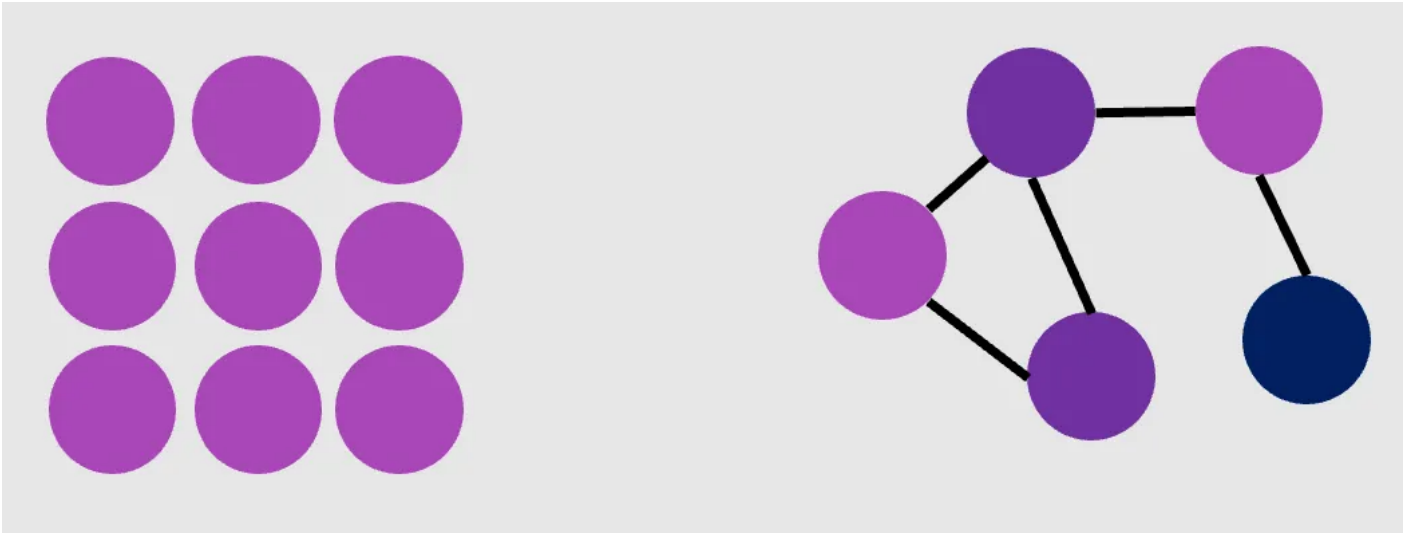
Speaking more mathematically, we have the requirement that our algorithm needs to be **permutation invariant**.

Another good example is if you flip an image, vs flipping a graph. The graph structure is completely unchanged from this, except for the node ordering (if you start with the left node as first node). An image however, completely changes.



Difficulty 3: Non-euclidean space

Finally, we have to fight with the fact that our domain is non-euclidean. For images we can simply define fixed-size filters, that operate on a grid structure, as it is done with Convolutional Neural Networks (CNNs). In our graph-case however, we have no fixed grid and therefore we need to operate on somewhat dynamic structures, that might change from neighborhood to neighborhood and lie randomly in space. There is no way to map the graph in a coordinate system such as it is possible with other data sources.



This (non-euclidean setup) is the reason why the whole machine learning area about graph data is also termed ***geometric deep learning***. And that in turn explains why the Python library we will use in the third part of this series is called PyTorch **Geometric**.

Graph Neural Networks are our friend

Luckily, Graph Neural Networks are able to handle all of the aforementioned problems for us. In a nutshell, they simply extend the notion of convolutions to graph data and perform a special form of **representation learning**. This idea is also called message passing, and will be explained in detail in the second part of this GNN series.

While a CNN has convolutional layers with learnable filters, that perform automatic feature extraction, GNNs have message passing layers, that learn the information about the graph structure and its features. Besides that, GNNs work just like any other feed forward neural network.

What you usually end up with are **node-level embeddings**, which are basically artificially generated vectors, that contain all the knowledge about the graph – the features of the node, the features of the neighbors, the connection info and the edge features. **This is generally seen as an extension of convolutions to non-euclidean structures**.

These node levels are fixed-size and can be used as direct input for a classification or regression network. Also, the message passing can be performed on *any* graph structure to incorporate node and edge information. Therefore, Graph Neural Networks have solved all three of our previous difficulties.