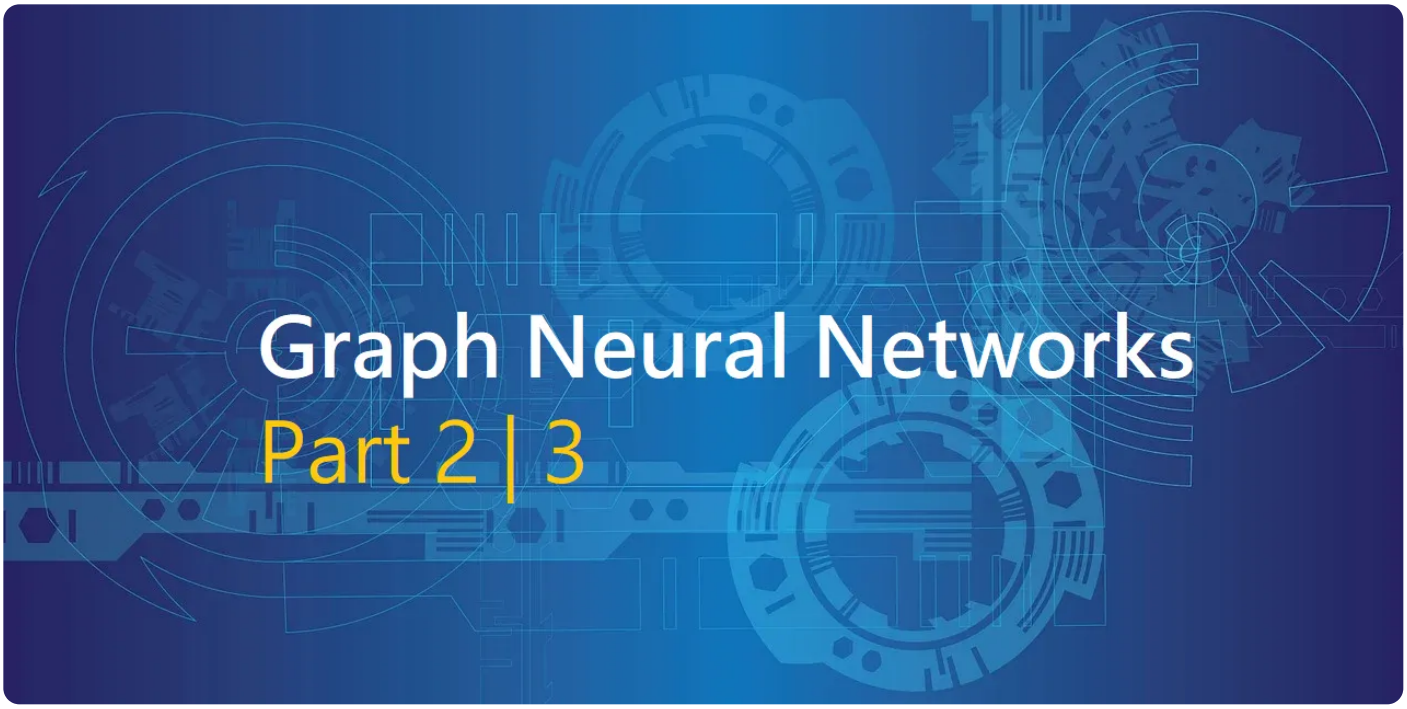
September 2, 2020

Understanding Graph Neural Networks | Part 2/3



Welcome back to this series on Graph Neural Networks. 🙂

After talking about the goal of Graph Neural Networks, and why we need them for graph data in the first part of this series, we are now ready to dig deeper. In the following we will understand mathematically and theoretically what is going on in the message passing layers of GNNs.

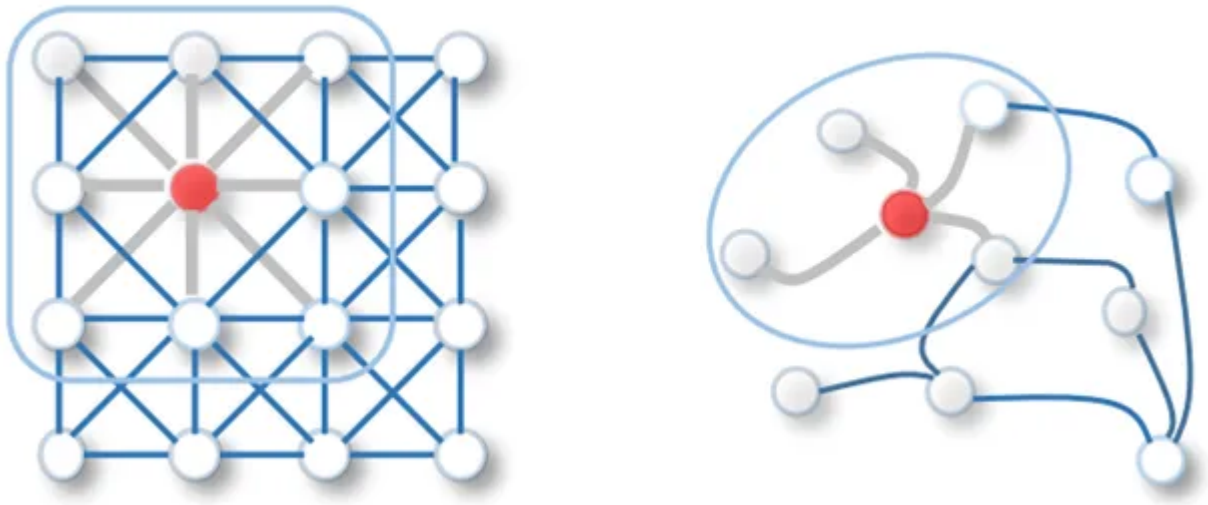
Don’t worry, it is actually pretty simple, yet exciting. After that, we can have a look at Pytorch Geometric to code our own models in the last part.

Here you find Part 2 on YouTube.

Extending Convolutions to Graph Data through Message Passing

Sometimes the literature of Graph Neural Networks can be confusing. Many different variants, and many different terms, such as convolutions and message passing. Eventually, they all point to the same idea of **passing information between the nodes of a graph** in order to learn an embedding that holds all the relevant information.

Have a look at the following illustration, and things will get clearer. For an image, you use Convolutional Neural Networks to apply filters (or kernels) on a local region in the image (this is usually also called *receptive field*). These filters will learn to extract certain information in the local area. Now you can see the image as a special type of graph, with a regular (grid) structure. What the filter simply does is aggregating the features of the *current pixle* (in the image below in red) as well as the features of the *neighboring pixles* .



Source: A Comprehensive Survey on Graph Neural Networks, Zonghan Wu et al., 2019

The same way the nodes in Graph Neural Networks use the features of their direct neighbors and their **own features**(current *representation* or *state*) and combine them somehow (we will see in a second, how exactly).

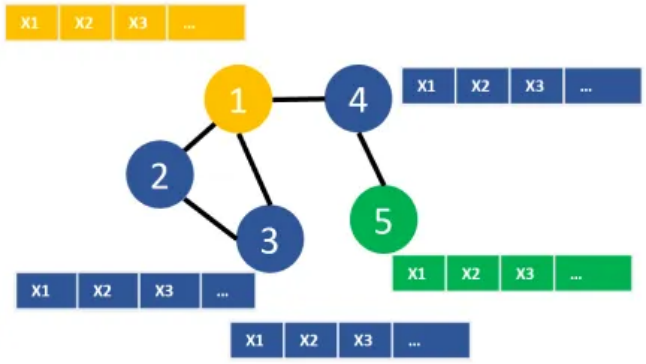
This exchange is called **neural message passing** and is done simultaneously by all nodes in the graph – for each message passing layer. If you think about it, it is like if many messages being sent between the nodes – hence the name.

This procedure leads to an exchange of structural and feature information and after one iteration all nodes are familiar with their **neighbors + their neighbors feature information**. After the next message passing step (this means the next layer in the GNN), the nodes are familiar about the **neighbors neighbor** and so on…. The number of layers basically specifies how many hops the exchange is carried out.

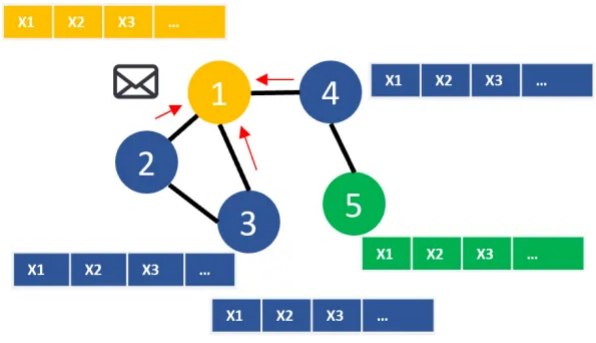
A simple visualized example

Let's have a look at this step by step. The image below is our graph in the following. The bars next to the nodes symbolize the **node features**. For example x_1 could be an atom feature, such as the atom-type (hydrogen, carbon, …), x_2 could be the number of protons and so on…

We will focus on the neural message passing performed *by node 1* (the orange one).

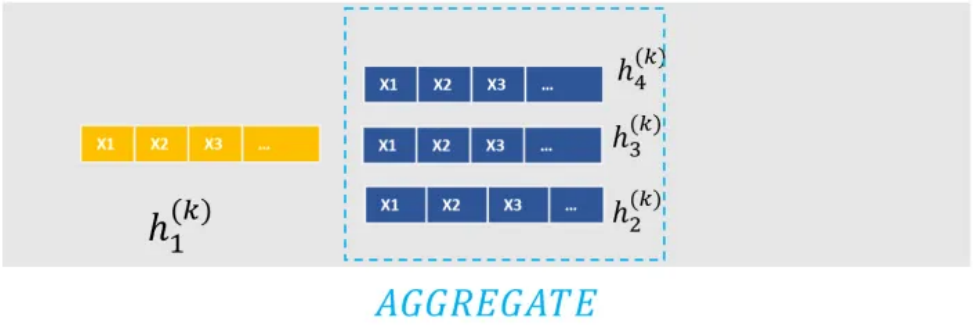


First, it recieves the message of its neighboring nodes, in this example the blue nodes. This means, that the neighbors share their node features – such as the atom type and proton number with our orange node.

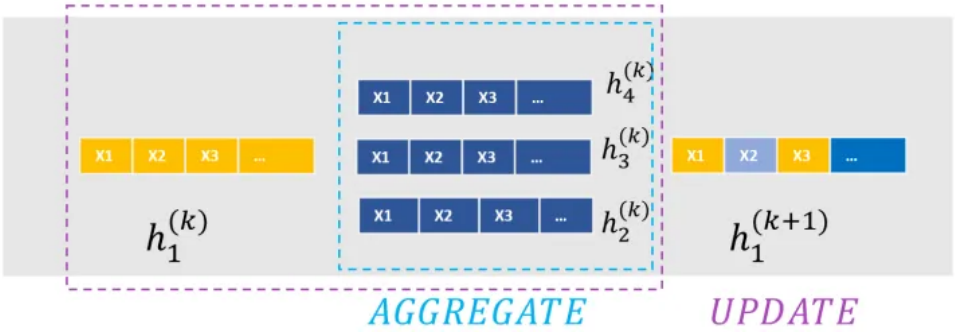


Now we aggregate the state information of the neighbors . In other words, we combine the information about the shared node features into a new vector.

The different $\mathbf{h_i^{(k)}}$ denote the current state of node *i* at step (or layer) *k*. The following illustration summarizes what we've got: our node's state $\mathbf{h_1^{(k)}}$ and the neighbor states $\mathbf{h_2^{(k)}}$, $\mathbf{h_3^{(k)}}$ and $\mathbf{h_4^{(k)}}$, which are aggregated. We will talk in a second about how exactly this AGGREGATE function can look like.

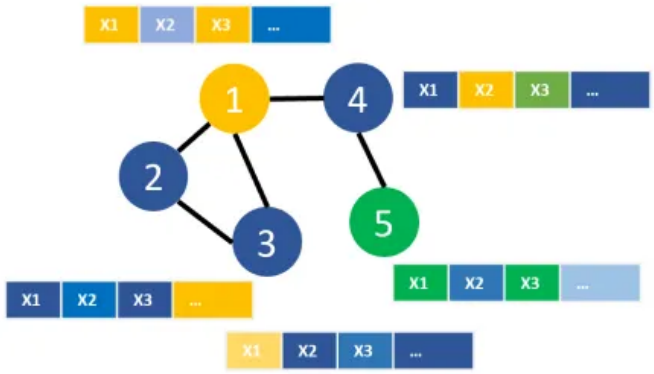


Our final step is to combine our current state with the aggregated states of the neighbors and we are done 🙂 – at least for this layer.

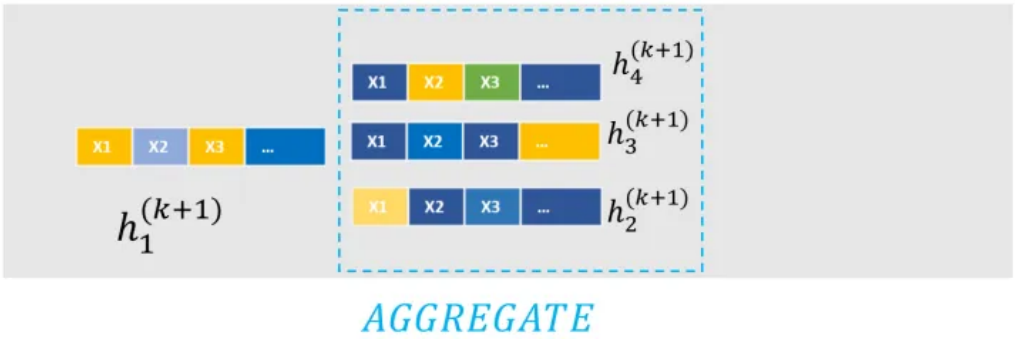


This is done by a specific UPDATE function (details about it in a second), that computes our new state $\mathbf{h_1^{(k+1)}}$. The coloring of the node-states was chosen on purpose, because the shades of blue in the new state $\mathbf{h_1^{(k+1)}}$ are supposed to illustrate what is going on: *our neighbor information (blue) moves into the current state of node 1 (orange)*. This means, that our node 1 knows something about its direct neighbors 2-4.

We have performed our first message passing step for node 1. Of course this happens for all the nodes simultaneously in the first layer. Therefore we end up with something like this. Please note how the green node (node 5) only has information about node 4 and itself. It has no information about our inspected node 1 (orange) yet.

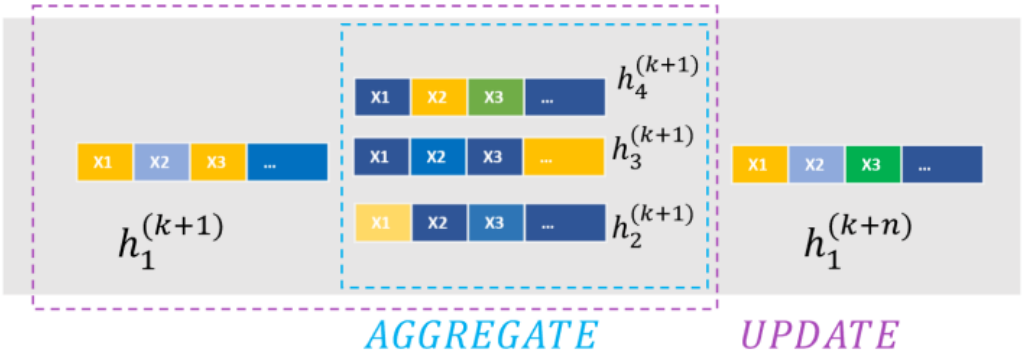


Luckily, with each message passing step we perform, we dig deeper into the graph. Let's perform another step to see what is happening.



We again aggregate the neighbor information of node 1, this means the states of 2, 3 and 4. It is worth to mention that it is not required anymore to have vectors of the same size as our initial node feature vectors. As we add more and more information to these states, we can of course make them bigger e.g. size = 128, 256, ….. This can be controlled in the AGGREGATE and UPDATE function.

Now we update our node 1's state again, using its **current state in step *k+1*** and the aggregated states of the neighbors. We could actually perform this n times, which is indicated by the (k+n) in the new state of node 1.

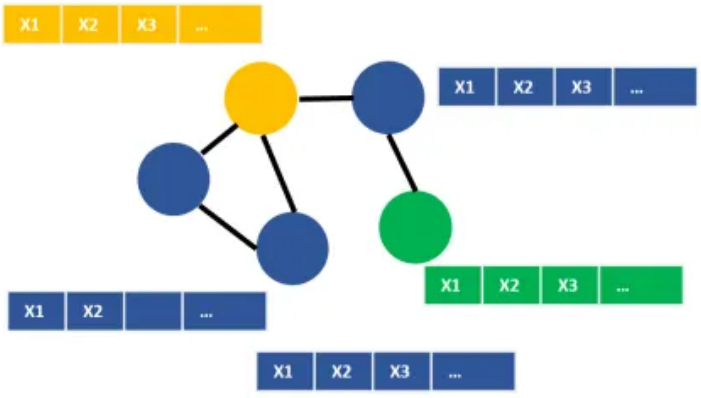


Now look at that. Our node 1 knows something about the green node (5) and its neighbors. **This means our Graph Neural Network has compressed all the relevant information about the graph into one node embedding**. We could use the node 1's embedding now to predict certain attributes about node 1, as we know node 1's features, it's neighbors features and the context of node 1 in the graph – simply brilliant! 🙂

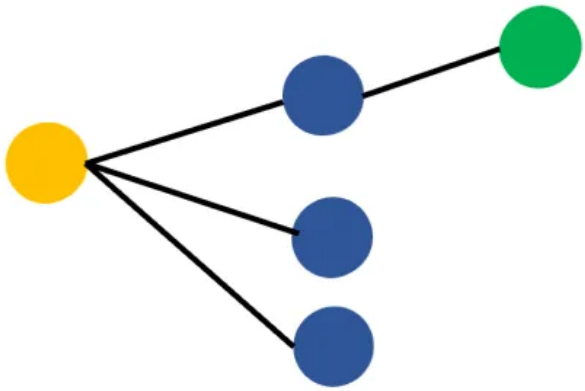
We can also combine the states of all nodes to obtain a representation of the whole graph. This means we are set-up for node-level predictions and graph-level predictions. This local feature aggregation can be compared to the kernels of a CNN, but additionally we also learn structural information!

Computation Graph Representation

The things we've seen a second ago can also be visualized in a different form. Instead of talking about n layers, which our graph flows through, we can also look at this on an **individual** node-level perspective. The representation we get is then called computation graph. If we re-arrange our graph…

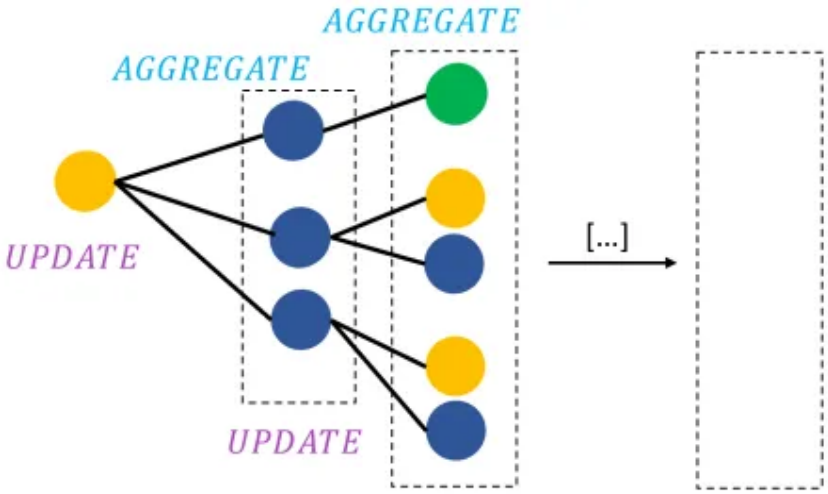


a little bit like this…



…which is still the same graph, we can immediately see the direct neighbors.

Now this is somehow similar to view our GNN as multi-layer network. Because after one message-passing in the GNN step we have the information about our first-level neighbors (the first dashed box in the image below). After the second message-passing layer we have information about the second-level neighbors and so on…



Each node has a **unique computation graph**, as it has different neighbors and therefore the message passing is defined individually depending on the local region in the graph. I just wanted to bring up this representation here, as it often occurs in literature 🙂

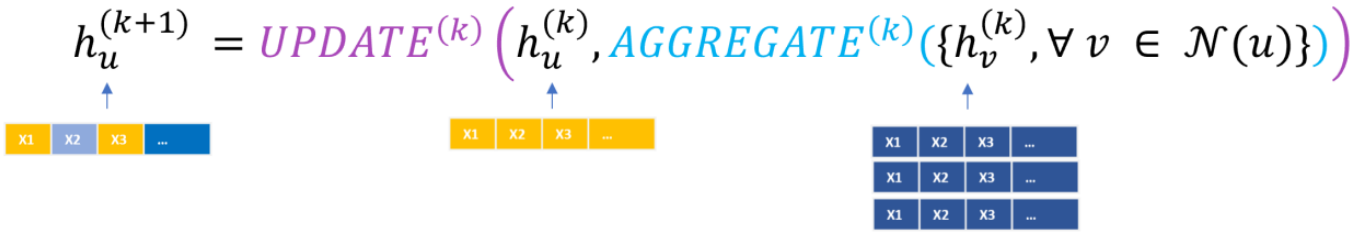
Formal definition of Graph Neural Networks

Now let's formalize the things we saw in the previous visualization. The update of node states is based on this simple formula

$$h_u^{(k+1)} = UPDATE^{(k)}\left(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall\, v \in \mathcal{N}(u)\})\right)$$

All it does is aggregating the state information of a node *u*'s neighbors h_v (from *u*'s neighborhood N) and finally combining it through an update function with the current node state h_u to obtain the new node state.

For our orange node 1 above this means something like this:



Now you might wonder how the AGGREGATE and UPDATE function exactly look like. This is the key point where the many variants of Graph Neural Networks (GAT, GGNN, GCN…) differ. The basic concept of message-passing Graph Neural Network is always the state-update function above, but the ways how this update and aggregation is performed are very different.

For the AGGREGATE – part of the neighbor states you can for example use **the mean**, **the sum**, or even a **Mult-Layer-Perceptron** (MLP). Using a learnable aggregation like the MLP comes closest to the learnable filters in CNNs.

For the UPDATE – part you will often find **the mean**, a **MLP**, or even a **Gated Recurrent Unit** (GRU), which also considers previous states.

Additionally, it is worth to mention that there exist methods to exploit the edge feature information, but we will not go into further detail on this. Still, in many applications the learning can be improved a lot if this information is included.

In the following we will have a quick look at a couple of popular variants of GNNs 🙂

Different Variants of Graph Neural Networks

Let's start with the Graph Convolutional Network (GCN), which does two interesting things. First of all, it completely drops the UPDATE function, by introducing self loops – this means each node has a connection to it self. As a result, we can simply process the current node's state together with its neighbors in the AGGREGATE function. Additionally, the paper performs a normalization (Kipf-Normalization) of the node states to improve the aggregation.

Graph Convolutional Networks, Kipf and Welling [2016]

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

Self-loop

Sum of normalized neighbor embeddings

Another common implementation of GNNs is to use a Multi-Layer-Perceptron as AGGREGATE function. Through the learnable parameters in the model, this allows to learn the best aggregation of the neighbors.

Multi-Layer-Perceptron as Aggregator, Zaheer et al. [2017]

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

Send states through a MLP

Graph Attention Networks (GAT) utilize the attention mechanism for GNNs. This simply means that attention weights are used to aggregate over the neighbor states.

Graph Attention Networks, Veličković et al. [2017]

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v}\mathbf{h}_v \qquad \alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)}$$

Attention weights

Finally, Gated Graph Neural Networks (GGNN) use a Gated Recurrent Unit to perform the UPDATE of the current state. This allows to better include sequential information about the states into the learning.

Gated Graph Neural Networks, Li et al. [2015]

$$\mathbf{h}_u^{(k)} = \mathrm{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})$$

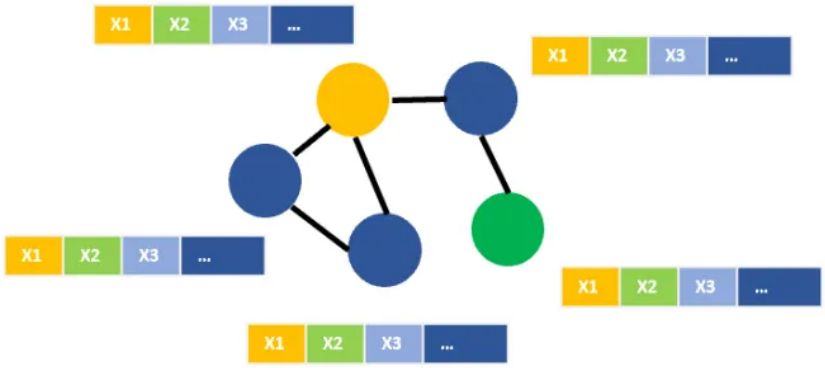
Recurrent update of the state

Finally, I want to mention that there exist a lot of other variants, and a great place to look them up can be found in this paper.

Remarks on the Depth of Graph Neural Networks and Over-Smoothing

Unlike Convolutional Neural Networks, stacking many layers might not make so much sense for Graph Neural Networks. The reason for this is, that the state-updates might oscillate, meaning that the node states already include all relevant information but still send and receive further messages.

This leads to the fact that all states become increasingly similar, which makes them indistinguishable from another. This might look something like this:



The reason for this is that the aggregations and updates are sequentially performed, which mixes up the information among the nodes. Therefore, with “standard” GNNs it is not recommended to go very deep, especially when working with small graphs, such as molecules. Meanwhile, there exist ways to avoid this behavior, such as PairNorm or DropEdge.

The field of Graph Neural Networks is constantly evolving and therefore the list of methods and approaches I collected is non-exhaustive! Nevertheless, this overview should provide you with all the basics you need to get started.

That’s it for this part, I hope you enjoyed it and if you have questions drop a comment!