Lecture 10

Chemical Prediction with Graph Neural Networks

Contents

Graph neural networks (GNNs) are particularly useful for chemical learning tasks, as most chemical systems, including molecules and materials, can be efficiently represented as a graph, as a graph encodes the connectivity information and is permutationally invariant. GNNs have been widely used in modern ML architectures for chemical tasks such as chemical prediction and generation.

In this lecture, we will discuss GNN. In particular, we focus on GNNs constructed using the message-passing framework. A GNN differs fundamentally from the feedforward neural networks (FNNs) discussed previously. An FNN is typically organized as a sequence of layers: feature layer $\rightarrow$ hidden layers $\rightarrow$ label layer. In contrast, a GNN follows a graph-in, graph-out paradigm: the input to the network is a graph with associated features, and each layer produces a new graph with the same topology but updated features.

At each GNN layer, the graph connectivity is kept fixed, while node features (and, in some formulations, edge or global features) are updated by aggregating information from neighboring nodes in the previous layer. This update mechanism is known as **message passing**. Each layer therefore corresponds to one round of neighborhood information exchange on the graph. An illustration of a GNN is shown in Fig. 1.

1 Message Passing GNN (MPGNN)

As the name suggests, message passing refers to the propagation of information between nodes on a graph. This idea is closely related to the ACSF and SOAP feature engineering methods discussed in the previous lecture, in the sense that both aim to encode the local atomic environment through interactions with neighboring atoms.

In a message passing neural network, a "message" can be interpreted as learned information describing interactions between connected nodes (atoms). By aggregating messages from its

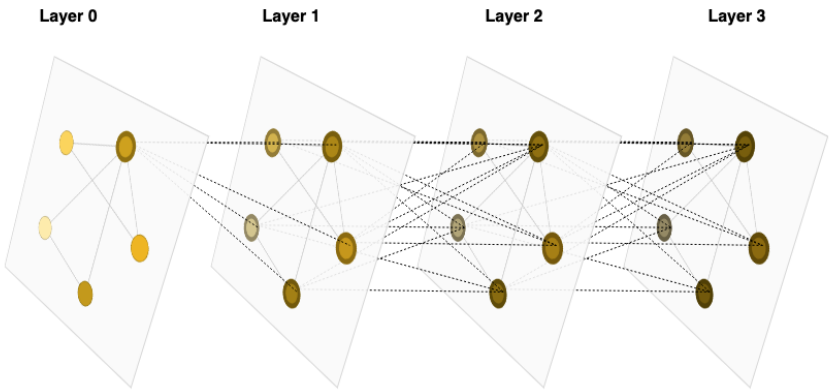

Figure 1: An illustration of Graph Neural Network (GNN). Figure adapted from Reference [1]

neighboring nodes, each node constructs a representation of its local environment. The node representation is then updated to encode information about both the node itself and its neighbors.

1.1 Math Notation

For a graph

$$G = (V, E), \tag{1}$$

where V represents all nodes, and E for all edges.

We use the following notation for the feature vectors of nodes and edges

- h_i: the feature vector of node i. Since the node feature is updated in each layer, we use $h_i^{(n)}$ to represent the node features at layer n.
- e_{ij}: the edge feature of the edge connecting nodes i and j.
- $m_{ij}^{(n)}$: the message sent from j to i at the nth layer.

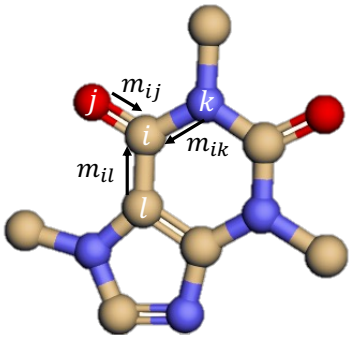

Figure 2: Message passing to atom i from bonded neighbors j, k and l.

Dimension of $m_{ij}^{(n)}$: The dimension of $m_{ij}^{(n)}$ is the same as the node feature (dimension of $h_i^{(n)}$).

1.2 Framework

The MPGNN framework contains three steps: (1) message construction, (2) message aggregation, and (3) node update.

1.2.1 Message Construction

The message $m_{ij}^{(n)}$ is a function of the node features $h_i^{(n)}$, $h_j^{(n)}$ and the edge feature e_{ij}:

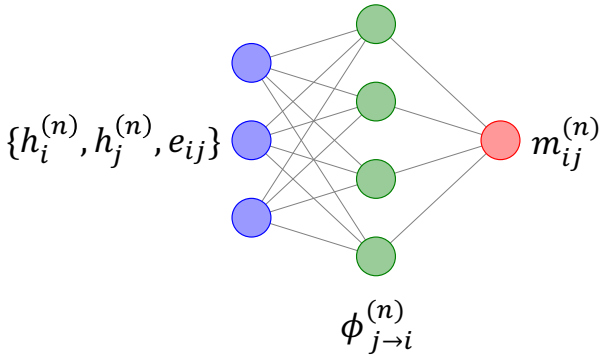

Figure 3: Using a shallow FNN to evaluate the message.

$$m_{ij}^{(n)} = \phi_{j\to i}^{(n)}\left(h_i^{(n)}, h_j^{(n)}, e_{ij}\right), \tag{2}$$

where $\phi^{(n)}$ is a function that we want to learn, called the **message function**. Usually, we can use a shallow FNN to represent $\phi^{(n)}$. In some references, you will hear people using *multi-layer perceptron (MLP)*[1] to denote the shallow FNN. An illustration is shown in Fig. 3.

This relationship means that the message sent from j to i depends on the node features, and how they are connected, e.g., bond type, distance and geometry.

1.2.2 Message Aggregation

After the node i receives messages from all its neighbors, we need to *process* the messages. Typical processing strategies include *sum, mean or max*, which are **permutation-invariant** operations.

$$m_i^{(n)} = \underset{j\in\text{neighbors}}{\text{AGGREGATE}}\left(m_{ij}^{(n)}\right). \tag{3}$$

For example, we can evaluate the final message to i as

$$\begin{aligned}
\text{sum:} \quad & m_i^{(n)} = \sum_{j\in\text{neighbors}} m_{ij}^{(n)} \\
\text{mean:} \quad & m_i^{(n)} = \frac{1}{N_{\text{neighbors}}} \sum_{j\in\text{neighbors}} m_{ij}^{(n)} \\
\text{max:} \quad & m_i^{(n)} = \max_j\left(m_{ij}^{(n)}\right)
\end{aligned} \tag{4}$$

[1] Historically, a neuron in an FNN has been called a perceptron.

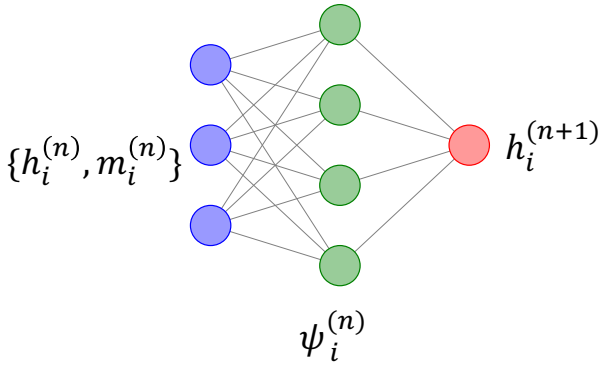

Figure 4: Using a shallow FNN to update the node.

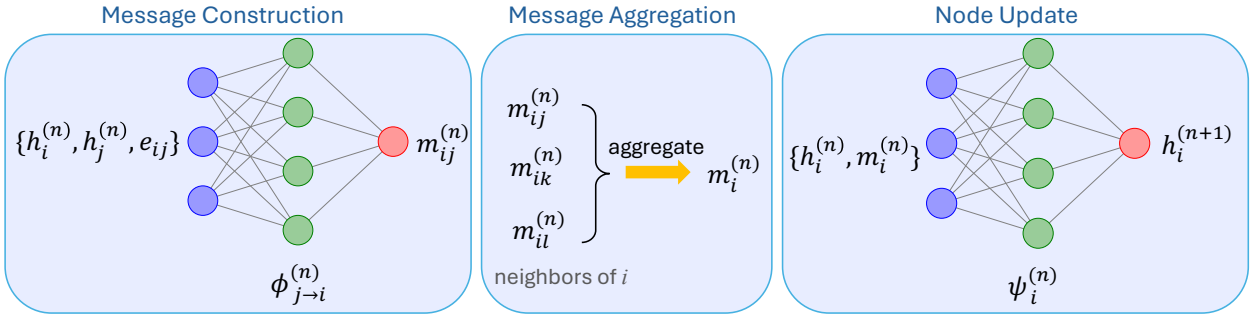

Figure 5: The three steps in message passing.

Note that we need the aggregation operation to be permutation-invariant since the order of the messages doesn't matter in a molecular system. This strategy is similar to the *pooling* method we introduced previously.

1.2.3 Node Update

Finally, based on the current node feature $h_i^{(n)}$ and the message it recieves $m_i^{(n)}$:

$$h_i^{(n+1)} = \psi_i^{(n)}\left(h_i^{(n)}, m_i^{(n)}\right), \tag{5}$$

where $\psi^{(n)}$ is another function that can be represented with a simple neural network, as shown in Fig. 4, called the **node update function**.

After computing the messages from neighbors, each node i updates its features based on the aggregated information. Once all nodes have been updated, one layer of the GNN update is complete.

The illustration of the above three steps is shown in Fig. 5.

1.3 Long-range interactions

Standard message passing updates node features using information from immediate neighbors (bonded atoms). However, many chemical properties depend on long-range interactions.

In a GNN, non-neighbor nodes can influence each other indirectly by stacking multiple layers. In the first layer, each node receives messages only from its directly connected neighbors. In the second layer, each node receives information from nodes that are two hops away, and in general, after k layers, each node can incorporate information from nodes within its k-hop neighborhood. Increasing the depth of the GNN thus allows long-range interactions to be captured through successive local message passing, illustrated by Fig. 6.

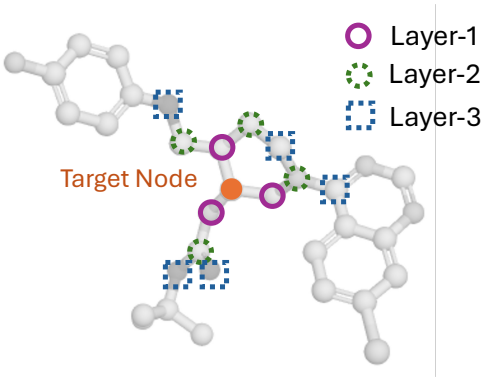

Figure 6: Illustration of capturing long-range interactions as adding more GNN layers.

While stacking multiple message-passing layers increases the receptive field of each node, standard GNNs still capture long-range interactions only indirectly. As the distance between nodes increases, the influence of distant nodes becomes weaker due to repeated aggregation and potential over-smoothing of features. Consequently, deep GNNs may struggle to fully model long-range effects that are crucial in chemical systems, such as electrostatic interactions or non-bonded van der Waals forces.

To address this limitation, several strategies have been proposed, including:

- Assign weights to different neighbors in node update.
- Using global or virtual nodes that can exchange information across the entire graph.
- Combining GNNs with physics-informed models that account for long-range interactions analytically.

2 Molecules as Graphs (Review)

As discussed in Lectures 2 and 8, a molecule can be represented as a graph with **nodes** (atoms) and **edges** (bonds):

- **Nodes (V):** Atoms with features such as element type, charge, and hybridization.
- **Edges (E):** Bonds with features such as bond order, conjugation, and stereochemistry.

The graph is vectorized using three matrices:

- **Feature matrix** X: $N \times d_n$, where N is the number of atoms and d_n the number of node

features.

- **Edge index**: $2 \times 2n_e$, storing source and target nodes for each of the n_e edges. Each undirected edge is stored in both directions.
- **Edge attribute matrix**: $2n_e \times d_e$, storing edge features for each edge in both directions, where d_e is the number of bond features.

3 From Graphs to Chemical Properties

Now suppose we represented a molecule as a graph and updated it after L layers. We would like to connect the graph to a chemical property, such as the energy, HOMO-LUMO gap, etc.

We use the same strategy in constructing the message passing GNN part. Let

$$y = \rho\left(\underset{i \in V}{\text{AGGREGATE}}\left(h_i^{(L)}\right)\right), \tag{6}$$

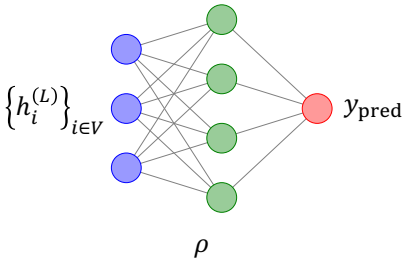

Figure 7: Making predictions with the node features.

where the aggregation operation is again permutation-invariant, and can be sum, mean, max, or weighted sum. The function ρ is again learned by a simple FNN (e.g., MLP), shown in Fig. 7, called the **readout function.**

Eventually, we built a prediction GNN that can be optimized by minimizing loss functions.

3.1 Parameters

In a message-passing GNN, the learnable parameters are those of the message function, node update function, and readout function, which are usually implemented as MLPs. Aggregation operations like sum or mean do not have parameters.

3.2 Loss functions

Nothing special compared to the previous prediction tasks. We use mean squared error (MSE) for regression and cross entropy for classification.

4 Lab - Chemical Property Prediction

The implementation of a GNN involves several parts. We will use `PyTorch` to implement each part.

4.1 MPNN Layer

`PyTorch Geometric` provides a class called `MessagePassing` from `torch_geometric.nn.MessagePassi` which contains basic attributes and methods for performing message passing. The website is `https://pytorch-geometric.readthedocs.io/en/2.7.0/generated/torch_geometric.nn.conv.MessagePassing.html`.

We will customize our class to implement MPNN layer inherited from the `MessagePassing` class. Below is an example.

```
from torch_geometric.nn import MessagePassing
import torch.nn as nn
class MPNNLayer(MessagePassing):
    def __init__(self, node_dim, edge_dim):
        super().__init__(aggr='add')  # aggregation: sum, can also be 'mean', 'max',
        ↪  or 'mul'

        # define the NN for message function
        # input: features of the two connected node and the edge
        self.message_mlp = nn.Sequential(
            nn.Linear(2*node_dim + edge_dim, 64), # hidden layer is 64
            nn.ReLU(),
            nn.Linear(64, node_dim) # dim of message = dim of node features
        )

        # define the NN for node update function
        # input: old node feature and aggregated message
        self.update_mlp = nn.Sequential(
            nn.Linear(node_dim + node_dim, 64),
            nn.ReLU(),
            nn.Linear(64, node_dim)
        )

    # function to update the nodes
    # returns updated node features x
    # self.propagate() calls message(), aggregate() and update() functions
    def forward(self, x, edge_index, edge_attr):
        # x: node features [num_nodes, node_dim]
```

```
        # edge_index: [2, num_edges]
        # edge_attr: [num_edges, edge_dim]
        return self.propagate(edge_index, x=x, edge_attr=edge_attr)

    # The following two functions are standard in the MessagePassing class, and
    ↪  will be automatically called by self.propogate()
    # the message function
    def message(self, x_i, x_j, edge_attr):
        # x_i: receiver node, x_j: sender node
        m = torch.cat([x_i, x_j, edge_attr], dim=-1) # attach all the features
        return self.message_mlp(m)

    # the update function
    def update(self, aggr_out, x):
        # aggr_out: aggregated messages
        out = torch.cat([x, aggr_out], dim=-1)
        return self.update_mlp(out)
```

In this setting, each layer contains two MLPs (simple FNNs), where we do not distinguish NLPs among nodes.

4.2 GNN Predictor

Next, we incorporate the `MPNNLayer` into a GNN model, where we repeatedly call the `MPNNLayer` to perform multi-layer GNN and prediction task.

Now we inherit from `nn.Module`. We examplify the process with the QM9 dataset, where the features are encoded with one-hot encoding.

```
import torch.nn as nn
import torch_geometric
class GNNPredictorModel(nn.Module):

    def __init__(self, node_dim, edge_dim, num_layers=3):
        '''
        num_layers defines how many layers we want to incorporate.
        '''
        super().__init__()
        # we use a simple affine function to do embedding.
```

```
        self.node_embedding = nn.Linear(11, node_dim)  # QM9 has 11 atom features
        self.layers = nn.ModuleList([MPNNLayer(node_dim, edge_dim) for _ in
        ↪  range(num_layers)]) # attach 3 MPNNLayer models

        # define the readout NN
        self.readout = nn.Sequential(
            nn.Linear(node_dim, 64), # message is of the same dim of node
            ↪  features
            nn.ReLU(),
            nn.Linear(64, 1) # predicting a number
        )

    def forward(self, data):
        # QM9 data here is already in graph representation
        x, edge_index, edge_attr, batch = data.x, data.edge_index, data.edge_attr,
        ↪  data.batch
        x = self.node_embedding(x)

        # update node features in each layer
        for layer in self.layers:
            x = layer(x, edge_index, edge_attr)

        # Graph-level readout
        x = torch_geometric.nn.global_mean_pool(x, batch) # evaluate the mean
        y_pred = self.readout(x)
        return y_pred
```

After we define the GNNPredictorModel, we simply train it as training an FNN model with PyTorch.

In this definition, we have $2 \times N_{\text{layer}} + 1$ MLPs defined: N_{layer} message functions, N_{layer} update functions, and 1 readout function.

4.3 Creating Graph Datasets

In the Lab of Lecture 2, we saw how to turn a molecule into a graph. In this Lab, I will provide a module for you to save your own data into graphs, just like the QM9 dataset we used. Since this part is more technical, you only need to know how to call it.

The links to the two labs:

- Lab 10-1: GNN for Chem Prediction: https://colab.research.google.com/drive/1LI0Yc73YE37TZnxxCp-IUi7t2VpNRhOR?usp=sharing
- Lab 10-2: Make graph datasets. https://colab.research.google.com/drive/11FHre3BTlzHkawrOKzc_sfKrAA3llDRd?usp=sharing

References

[1] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill.Pub*, 2020.