9 Graph Neural Network

Previously, we discussed how to use convolutional neural network or recurrent neural network to encode regular grid-like data. However, many real-world data are irregular and are represented as graphs. Specifically, a graph is a data structure that consists of a set of nodes and a set of edges connecting these nodes. Graph Neural Networks (GNNs) are a specialized class of neural networks designed for processing such network data. GNNs also have various healthcare domains encompassing diverse graphs, including molecular representations [57, 58, 59], spatiotemporal networks [60], drug-drug interaction networks [61], protein-protein interaction networks [62], gene expression networks [63], biomedical knowledge graphs and medical ontologies [64, 65]

A unique strength of GNNs lies in their capacity to model intricate relationships among entities. Consider a medical network: its nodes could represent patients, diseases, medications, or healthcare providers, while the edges encode various relations between these concepts, such as "patient A has disease B", "drug A could interact with drug B." Out of all deep learning models, GNNs are particularly good at understanding the connectivity between entities and extracting nuanced information from graph-structured data based on neighborhood similarity patterns.

One of the most exciting applications of Graph Neural Networks (GNNs) is in drug discovery. GNNs provide a powerful tool for predicting the characteristics of prospective drug candidates by representing a molecule as a graph, where nodes represent atoms and edges represent chemical bonds. With this representation, GNNs can predict important properties such as the solubility or toxicity of a new molecule, thus expediting the drug discovery process [66]. This approach can streamline the identification of promising compounds and is a promising area of research in drug development.

In this chapter, we introduce different variants of graph neural networks, including graph convolutional networks (GCN) [67], graph attention networks (GAT) [68], and message passing neural networks (MPNN) [69]. We also present two hands-on GNN applications in healthcare: 1) GNN application in drug property prediction, and 2) GNN pipelines for clinical predictive tasks using the PyHealth package.

9.1 Introduction to Graph Neural Networks

Learning representations of graph structures, such as node embedding or edge embedding, has found numerous applications in modeling real-world graphs, including protein networks [70], social networks [71], and co-author networks [67]. Traditional graph-based models use well-defined heuristics for learning graph node embeddings. For example, DeepWalk [72] borrows ideas from random walk on graphs, LINE [73] leverages local and global node proximity, and Node2Vec [74] relies on both breadth-first search (BFS) and depth-first search (DFS) over the graphs.

In 2016, Kipf and Welling proposed graph convolutional networks (GCN) [75], which elegantly inject the graph adjacency information into a neural network architecture (simply aggregating node features by matrix product with adjacency matrix) and show consistently better performance against traditional baselines.

Extending GCN, graph attention networks (GAT) proposed by Velickovic et al. (2017) [68] dynamically learn edge weights of the networks. While GCN and GAT learn embedding vectors of nodes not for edges, message-passing neural networks (MPNN) developed by Gilmer et al. (2017) [69] provide a method to learn embedding vectors for both nodes and edges.

In this chapter, we conduct an in-depth analysis of these models, accompanied by an illustration depicted in Figure 9.1, aiming to facilitate a better understanding of different neighborhood aggregation mechanisms.

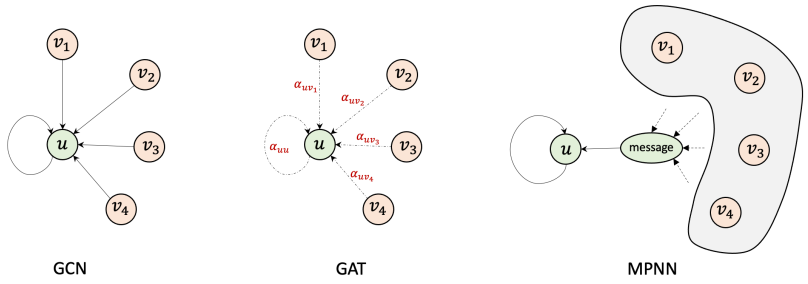

Figure 9.1 Illustration of three graph neural network models

Graph convolutional networks (GCN)

Kipf et al. [67] introduced *graph convolutional networks* (GCN) to extend the concept of image convolution (i.e., grid-like graph) to any arbitrary graph. In this approach, a graph structure is defined as a set of nodes, denoted by $\mathcal{N}$, where the number of nodes is represented by $|\mathcal{N}| = N$. Additionally, each node has a feature vector $\mathbf{x} \in \mathbb{R}^d$, and we stack the feature vectors for all N nodes to form a matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$. For instance, the graph can represent a molecular graph where each node is an atom with a d-dimensional feature vector (e.g., atomic number, atomic mass, valence, aromaticity, etc.) and edges denote the chemical bonds.

Let us assume $\mathbf{A} \in \{0,1\}^{N \times N}$ be the adjacency matrix of the graph structure,

which indicates node connections: if node i and node j are connected in the graph, $\mathbf{A}[i, j] = 1$, otherwise 0. Additionally, we assume $\mathcal{E}$ as the edge set of the graph. For example, $(i, j) \in \mathcal{E}$ if $\mathbf{A}[i, j] = 1$, which means that there is an edge between node i and j. Note that GNN works mostly with undirected graphs, i.e., $\mathbf{A}[i, j] = 1$ implies $\mathbf{A}[j, i] = 1$, and thus $\mathbf{A}$ is symmetric.

Example. For certain prediction tasks, such as forecasting the COVID-19 cases at every county in the US, traditional DNN models might treat each county as an independent training sample and build a multi-layer model to predict the counts based on county-level features. Differently, GCN models could leverage the spatio-temporal connections between counties. It first connects nearby counties together to form an adjacency graph $\mathbf{A}$ since they are geographically similar and thus share similar COVID responses. Then, GCN models will leverage the graph structure $\mathbf{A}$ to aggregate the features among nearby counties to forecast the target.

Formally, one graph convolution layer could be defined as,

$$\mathbf{H}^{(t)} = \text{ReLU}(\tilde{\mathbf{A}}\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}), \tag{9.1}$$

where the initial hidden embeddings, $\mathbf{H}^{(0)} = \mathbf{X}$, are the county-level features. $\tilde{\mathbf{A}}$ represents the normalized adjacency matrix (we will discuss it below) and $\mathbf{W}^{(t)}$ represents the layer-wise parameter matrix (t is the layer index). In comparison, the layer-wise propagation of a simple DNN can be represented as $\mathbf{H}^{(t)} = \text{ReLU}(\mathbf{H}^{(t-1)}\mathbf{W}^{(t)})$, and the difference is the multiplication of the adjacency matrix. Essentially, GNN adds dependency between different counties when making the predictions.

Normalizing the graph adjacency matrix For numerical stability purposes, the graph learning algorithm usually requires a normalized adjacency matrix. There are two common ways of normalizing the adjacency matrix: *random walk normalization* and *symmetric normalization*.

The matrix $\mathbf{A}$ is symmetric by definition. Researchers often add a self-loop to connect a node with itself and improve the numeric stability of graph operations. This is achieved by setting $\mathbf{A}[i, i] = 1, \forall\, i$, which is equivalent to adding an identity matrix to the adjacency.

$$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}. \tag{9.2}$$

Random Walk Adjacency Normalization: The first type is named *random walk normalization*, which calculates the degree of each node, resulting into a matrix $\mathbf{D} \in \mathbb{N}^{N\times N}$. $\mathbf{D}$ is a diagonal matrix and each element is the row sum of the self-looped adjacency matrix.

$$d_{ii} = \sum_{j=1}^{N} a_{ij}. \tag{9.3}$$

Then, the random walk normalization will normalize $\mathbf{A}$ over each row, making the row sum become 1, which aligns with the concept of random walk transition matrix

in Markov process (will discuss that in Chapter 9). Thus, its name is random walk normalized adjacency $\tilde{\mathbf{A}}_{rw}$.

$$\tilde{\mathbf{A}}_{rw} = \mathbf{D}^{-1}\mathbf{A}. \tag{9.4}$$

Example: Consider a simple graph with the following adjacency matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Adding the self-loop gives:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Calculating the degree matrix $\mathbf{D}$, we have:

$$\mathbf{D} = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

Then, the random walk normalized adjacency $\tilde{\mathbf{A}}_{rw}$ is:

$$\tilde{\mathbf{A}}_{rw} = \mathbf{D}^{-1}\mathbf{A} = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix}$$

Symmetric Adjacency Normalization: The second type uses the same degree matrix $\mathbf{D}$, however, on both sides of $\mathbf{A}$ to keep the output matrix symmetric.

$$\tilde{\mathbf{A}}_{sym} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}. \tag{9.5}$$

Example: Continuing from the previous example with the self-looped adjacency matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

and the degree matrix:

$$\mathbf{D} = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

The symmetrically normalized adjacency $\tilde{\mathbf{A}}_{sym}$ is calculated as follows:

$$\tilde{\mathbf{A}}_{sym} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} = \begin{pmatrix} \sqrt{\frac{1}{3}} & 0 & 0 \\ 0 & \sqrt{\frac{1}{2}} & 0 \\ 0 & 0 & \sqrt{\frac{1}{2}} \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \sqrt{\frac{1}{3}} & 0 & 0 \\ 0 & \sqrt{\frac{1}{2}} & 0 \\ 0 & 0 & \sqrt{\frac{1}{2}} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{3} & \sqrt{\frac{1}{6}} & \sqrt{\frac{1}{6}} \\ \sqrt{\frac{1}{6}} & \frac{1}{2} & 0 \\ \sqrt{\frac{1}{6}} & 0 & \frac{1}{2} \end{pmatrix}$$

In fact, the Equation (9.1) can also be re-written into node-level updating formula (with self-loop and random walk normalization applied to the adjacency matrix),

$$\mathbf{Z}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{W}^{(t)}, \tag{9.6}$$

$$\alpha_{uv}^{(t)} = \frac{1}{|\mathcal{N}(u)| + 1}, \ \forall (u, v) \in \mathcal{E}, \tag{9.7}$$

$$\mathbf{h}_u^{(t)} = \text{ReLU}\left(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(t)} \mathbf{z}_v^{(t)} \right). \tag{9.8}$$

Here, $\mathcal{N}(u)$ means the neighborhood set of node u.

- Equation (9.6) is the same as DNN model, which is a linear transformation.
- Equation (9.7) is the weight score for all edges connected to node u. With the random walk normalization, the GCN model treats each edge (within the neighborhood of node u) equally and consistently use $\frac{1}{|\mathcal{N}(u)|+1}$ (the denominator is the number of neighbors of u $|\mathcal{N}(u)|$ plus 1) as the weight. Essentially, $\alpha_{uv}^{(t)}$ is the (u, v) element in the inverse degree matrix (after adding self-loop) $\mathbf{D}^{-1}$.
- Equation (9.8) takes the weighted sum of all the neighborhood sets (including the node u itself) and applies the ReLU function to add non-linearity.

The PyTorch implementation of the GCN model can be found below. Readers might refer to this repository[1] as well, where we provide a notebook on applying the GCN model to the Zachary's karate club dataset[2].

```
class GraphConvolutionLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super(GraphConvolutionLayer, self).__init__()
        self.transform = nn.Linear(in_features, out_features)

    def forward(self, adjacency_matrix, input_features):
        """
        adjacency_matrix: adjacency matrix, N x N
        input_features: node feature matrix, N x d_i
        """
        # apply the adjacency matrix first
        adj_X = adjacency_matrix @ input_features
        output = self.transform(adj_X)
        return output

class GCN(nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes):
        super(GCN, self).__init__()
```

[1] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap7-GNN/notebook
[2] https://en.wikipedia.org/wiki/Zachary's_karate_club

```
        # define two graph covolutional layers
        self.layer1 = GraphConvolutionLayer(num_features, hidden_dim)
        self.layer2 = GraphConvolutionLayer(hidden_dim, num_classes)

    def forward(self, adjacency_matrix, features):
        # add relu between two GCN layers
        h = F.relu(self.layer1(adjacency_matrix, features))
        output = self.layer2(adjacency_matrix, h)
        return output

model = GCN(num_features=10, hidden_dim=128, num_classes=1)
```

- `GraphConvolutionLayer` class: This class defines a single graph convolutional layer. It takes in the number of input features and the number of output features as parameters. Inside the constructor (`__init__`), it initializes a linear transformation (`self.transform`) which will be applied to the input features. In the forward method, it takes an adjacency matrix and input features as input, performs the graph convolution operation by multiplying the adjacency matrix with the input features, and then applies the linear transformation to the result.
- `GCN` class: This class represents the entire graph convolutional network. It takes the number of input features, hidden dimension size, and the number of output classes as parameters. In the constructor, it initializes two graph convolutional layers (`GraphConvolutionLayer`) with appropriate input and output feature dimensions. In the forward method, it performs the forward pass through the network. It first applies the first graph convolutional layer followed by a ReLU activation function. Then, it applies the second graph convolutional layer to produce the final output.

GCN for Directed Graphs. For directed graphs, the adjacency matrix $\mathbf{A}$ is not necessarily symmetric: the presence of an edge from node i to node j does not guarantee an edge in the opposite direction. To adapt GCN to directed graphs, we can, for example, (1) use different normalized adjacency matrices for inbound and outbound edges, apply separate convolutional operations for each, and then combine the results; or (2) include edge features in the convolutional operation. Both modifications can enable the GCN to capture the directional properties of the edges.

Graph attention networks (GAT)

GAT model is invented by Velivckovic et al. [68], which incorporates the attention mechanism (we discussed in Chapter 7) into GCN model. The main difference is that GCN implicitly treats all edges as equal while GAT learns to use the attention mechanism between two connected nodes to re-weight the edges. The attention score will be used as the edge importance in graph convolution. The layer-wise propagation

in GAT can be formulated similarly in the node level as,

$$\mathbf{Z}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{W}^{(t)}, \tag{9.9}$$

$$e_{uv}^{(t)} = \text{LeakyReLU}\left(\left[\mathbf{z}_u^{(t)} || \mathbf{z}_v^{(t)}\right]\theta^{(t)}\right), \ \forall (u,v) \in \mathcal{E}, \tag{9.10}$$

$$\alpha_{uv}^{(t)} = \frac{\exp\left(e_{uv}^{(t)}\right)}{\sum_{k\in\mathcal{N}(u)}\exp\left(e_{uk}^{(t)}\right)}, \tag{9.11}$$

$$\mathbf{h}_u^{(t)} = \text{ReLU}\left(\sum_{v\in\mathcal{N}(u)} \alpha_{uv}^{(t)}\mathbf{z}_v^{(t)}\right). \tag{9.12}$$

Here, both the layer-wise weight matrix $\mathbf{W}^{(t)}$ and the attention weight $\theta^{(t)}$ are parameters.

- Equation (9.10) calculates the attention score using a non-linear neural network (the third approach in Section 7.2.3). Basically, we concatenate the embedding of two connected nodes, u and v, and then apply a linear transformation with a LeakyReLU activation.
- Equation (9.11) applies the Softmax activation on the attention score, ensuring that the normalized attention scores sum up to 1 for the neighborhood of node u. In GCN, these two steps are merged into one with equal weights in the same neighborhood.
- Equation (9.12) similarly takes the weighted sum of all the neighborhood sets (including the node u itself) and applies the ReLU function to add non-linearity.

The PyTorch implementation of GAT is presented below, and we provide the notebook showing its application on Karate dataset in this repository [3].

```python
class GraphAttentionLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super(GraphAttentionLayer, self).__init__()
        self.in_features = in_features
        self.out_features = out_features

        self.W = nn.Linear(in_features, out_features)
        self.a = nn.Linear(2*out_features, 1)
        self.leakyrelu = nn.LeakyReLU(0.05) # leaky relu has a hyper param

    def forward(self, adj, X):
        """
        adj: adjacency matrix, N x N
        X: node feature matrix, N x d_i
        """
        # step 1
        h = torch.mm(X, self.W)
        N, _ = h.shape

        # step 2
```

[3] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap7-GNN/notebook

```
        # attention input: concatenating embedding of node u and node v
        a_input = torch.concat([h.unsqueeze(1).repeat(1, N, 1), \
                                h.unsqueeze(0).repeat(N, 1, 1)], 2)
        # calculate attention scores
        e = self.leakyrelu(torch.matmul(a_input, self.a).squeeze(2))
        # add mask to attention matrix
        zero_vec = -1e15*torch.ones_like(e)
        attention = torch.where(adj > 0, e, zero_vec)
        # apply softmax to attention
        attention = F.softmax(attention, dim=1)

        # step 3
        # use attention to reweight the nodes and sum them up
        h = torch.matmul(attention, h)
        return h


class GAT(nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes=4):
        super(GAT, self).__init__()
        self.layer1 = GraphAttentionLayer(num_features, hidden_dim)
        self.layer2 = GraphAttentionLayer(hidden_dim, num_classes)

    def forward(self, adjacency_matrix, features):
        h = F.relu(self.layer1(adjacency_matrix, features))
        output = self.layer2(adjacency_matrix, h)
        return output

model = GAT(num_features=10, hidden_dim=128, num_classes=1)
```

The main difference lies in the `GraphAttentionLayer` class. This class defines a single graph attention layer. It takes in the number of input features and the number of output features as parameters. Inside the constructor (`__init__`), it initializes parameters for linear transformations (`self.W` and `self.a`) which will be applied to the input features. Additionally, it initializes a leaky ReLU activation function (`self.leakyrelu`). In the forward method, it takes an adjacency matrix (`adj`) and input features (`X`) as input. It first applies a linear transformation to the input features (`h = torch.mm(X, self.W)`, step 1). Then, it calculates attention scores using a learned attention mechanism (step 2). It concatenates embeddings of node pairs, calculates attention scores, applies a mask to the attention matrix, then a softmax function to get attention weights. Finally, it reweights the nodes based on attention weights (step 3).

Message Passing Neural Network (MPNN)

MPNN is a generalized framework of GCN, GAT, proposed in [69]. It unifies different graph convolution variants into two stages: (i) aggregate message from neighbors; (ii) use message to update the node representation.

$$\mathbf{m}_u^{(t)} = \text{Aggregate}^{(t)}\left(\mathbf{h}_u^{(t-1)}, \{\mathbf{h}_v^{(t-1)} : v \in \mathcal{N}(u)\}\right), \ \forall u \in \mathcal{V}, \tag{9.13}$$

$$\mathbf{h}_u^{(t)} = \text{Update}^{(t)}\left(\mathbf{h}_u^{(t-1)}, \mathbf{m}_u^{(t)}\right). \tag{9.14}$$

- Equation (9.13) takes the last-layer embedding from neighborhood set of node u and aggregates them into a message $\mathbf{m}_u^{(t)}$. For example, the aggregation function in GCN is an equally weighted sum while the aggregation in GAT is an attention sum.
- Then, Equation (9.14) takes the message $\mathbf{m}_u^{(t)}$ as well as the previous embedding of node u as two inputs and updates the current embedding by an update function. For example, the update function in both GCN and GAT is adding the previous embedding to the message and then applying a nonlinear transformation to be the updated embedding $\mathbf{h}_u^{(t)}$.

The PyTorch implementation of various MPNN models could refer to this repository[4]. Below, we show one type of MPNN implementation from a recent drug recommendation paper [76] (refer to Equation 7 and Equation 8 of the paper), which uses MPNN for drug molecule graph representation. More concrete applications on Karate datasets could be found in this repository [5].

```
class MessagePassingLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super(MessagePassingLayer, self).__init__()
        self.message_passing = nn.Linear(2 * in_features, out_features)
        self.read_out = nn.Linear(out_features, out_features)

    def forward(self, adj, X):
        """
        adj: adjacency matrix, N x N
        X: node feature matrix, N x d_i
        """
        N, _ = X.shape
        # combine features of node u and node v
        # X.unsqueeze(1).repeat(1, N, 1): N x N x d_i, repeat X at dimension 1
        # X.unsqueeze(0).repeat(N, 1, 1): N x N x d_i, repeat X at dimension 0
        # Z: N x N x 2d_i, features of every node pair
        Z = torch.concat([X.unsqueeze(1).repeat(1, N, 1), \
                                X.unsqueeze(0).repeat(N, 1, 1)], 2)
        # apply message passing to aggregate their features
        # Z: N x N x 2d_i -> N x N x d_o
        Z = self.message_passing(Z)
        # apply graph convolution to sum the embedding of (u,v) pairs
        # N x N x d_o, N x N -> N x N x d_o
        # use Einstein summation to perform a batch matrix multiplication
        Z = torch.einsum("abc,ad->dbc", Z, adj)
        # N x d_o
        # extract the diagonal elements from the tensor Z
        Z = Z[torch.arange(N), torch.arange(N)]
        # apply the read out function
        # the updated representations for each node after message passing
        H = self.read_out(torch.relu(Z))
        return H
```

[4] https://github.com/priba/nmp_qc
[5] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap7-GNN/notebook

```python
class MPNN(nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes=4):
        super(MPNN, self).__init__()
        self.layer1 = MessagePassingLayer(num_features, hidden_dim)
        self.layer2 = MessagePassingLayer(hidden_dim, num_classes)

    def forward(self, adjacency_matrix, features):
        h = F.relu(self.layer1(adjacency_matrix, features))
        output = self.layer2(adjacency_matrix, h)
        return output

model = MPNN(num_features=10, hidden_dim=128, num_classes=1)
```

Essentially, GCN, GAT, MPNN, and their variants are fundamental tools for learning graph node embeddings, and the embedding vectors could be used for various different purposes, such as node classification, node value regression, edge classification, graph classification, etc.

9.2 Advancements in Graph Neural Networks

Deep learning on graphs has made much exciting progress in both practical deployments and various application domains. This section will cover some advancements in graph neural network research.

9.2.1 Improvements on GNN training

To apply common GNN models (including GCN, GAT, etc) on real-world graphs, some issues require attention from researchers.

- **Neighborhood sampling.** First, the real-world graph is often large (tens of millions of nodes on the graph), and each node could connect to many edges (e.g., a dense graph). Applying graph convolution on such graphs is expensive and sometimes unnecessary. To approximate the convolution effect from all neighbors, GraphSAGE [77] leverages Monte-Carlo methods and samples a small subset of the neighbors via edge sampling to calculate the neighborhood aggregation effects. Various experiments have shown that this method could greatly reduce the computation head of graph convolution while providing similar outcomes.
- **Distributed graph training.** GraphSAGE could alleviate the dense graph issue to some extent. However, some real-world graphs could be huge (such as social networks on Facebook) and may not fit into the memory of a single machine, or a single party could not access the whole graph structure. To address this issue, researchers have proposed methods of distributed graph learning at scale [78, 79] that decompose the large graph into different subgraphs and run graph learning algorithms in a distributed way. Then, the model broadcasts the gradients from other machines to sync the global parameters.

- **Learning on heterogeneous graphs.** Real graphs (such as medical concept graphs) can contain different types of concepts (such as patient node, medication node, diagnosis node) and various relations among nodes. In contrast, standard graph learning algorithms can only be applied to simple graphs. Thus, recent researchers also propose graph learning algorithms [80, 81, 82, 83] on these heterogeneous graphs for learning embeddings of different types of entities.

9.2.2 Applications of GNN in Different Domains

GNN methods have also benefited many other fields exposed to structured data, such as computer vision and natural language processing.

GNN in computer vision

In computer vision, researchers use graph neural networks to build the prediction models on various data modalities, such as 2D natural images [84], videos [85], 3D graphics data [86], vision + language [87], as well as medical images [88]. Recent survey papers [89, 90] provide a holistic view of GNN applications in vision.

In 2D images, the graph could be constructed from various perspectives, such as (i) label space [84], where each node represents a label (using word embedding as initial features) and their inter-dependencies as edges; (ii) coordinate space graph [91]; (iii) region graphs [92] considering spatial and temporal similarities; (iv) patch graphs [93] from the original image by splitting it into patches (such as 4-by-4) and then connect the patches into graphs via content similarity or local similarity.

Similarly, for 3D videos, various types of graphs have been constructed, such as (i) [85] connects all objects in the frames as a space-time region graph while the edges are based on spatial-temporal relations in consecutive frames; (ii) [94] connects joints in the same frame according to the natural connectivity in the human skeleton as a graph; (iii) [95] uses object detection graph to encode its temporal structure and predicts whether an edge is active or inactive to obtain the connected edges.

In medical imaging, different types of graphs are built that could improve analysis of human brain activities, such as (i) [96] divides the brain images into regions while the inter-region connections can be viewed as graph edges; (ii) [88] connects the imaging features of different subjects as vertices in the graph for accessing brain analysis in population-level; (iii) [97] uses Transformer to process the dynamic brain graph constructed from spatial-temporal attention.

GNN in natural language processing (NLP)

In natural language processing (NLP), research on graph neural networks could be decomposed into two parts, as summarized in this survey paper [98].

First, many methods have been proposed for graph construction, including constructing static graphs [99] (such as dependency graphs [100], AMR graphs [101], similarity graphs, co-occurrence graphs, co-reference graphs [102], constituency graphs), dynamic graphs [103] (such as node embedding based similarity graphs, attention-based similarity graphs [36], cosine similarity graphs, structured-aware similarity graphs

[104]), or hybrid graphs for the application of GNNs, and the nodes in the graphs could be homogeneous [105] (such as molecular graphs with atom as nodes), heterogeneous [82] (such as disease-drug bipartite graphs) or multi-relational [106] (EHR knowledge graphs contain patients, drugs, diagnosis codes, etc).

Second, with these diverse graphs, GNN models have benefited various applications in many different problem settings, such as graph-to-sequence [107], graph-to-tree [108], graph-to-graph translations [109], and various NLP tasks, such as natural language generation [104], questions answering [110], information extraction [111], knowledge graph reasoning [112], etc.

9.3 Graph Neural Networks for Drug Property Prediction

The structural composition of drug molecules plays a pivotal role in determining their properties, as specific molecular arrangements are intricately linked to distinct characteristics. For example, particular molecular structures may bolster efficacy, while others might influence factors such as toxicity or bioavailability.

GNNs are good at leveraging the intricate representations inherent in molecular graph structures, showcasing promise in predicting diverse drug properties, as evidenced by notable studies [113, 114, 115]. Typically, leveraging GNNs for predicting the properties of drug molecules involves a two-stage procedure:

- **Node Embedding**: Utilizing multiple graph convolutional or attention layers, GNNs generate node embeddings that encapsulate the structural characteristics of the molecule. These embeddings are designed to capture hierarchical representations by discerning the molecular context at various scales.
- **Property Prediction**: Post-generation of node embeddings, prior studies typically implement a pooling layer to aggregate these embeddings into the overall molecular graph embedding. Subsequently, a final one or two-layer neural network is employed on top of the graph embedding to forecast whether the molecule exhibits specific properties. This prediction task commonly assumes a binary classification framework, aiming to discern the presence or absence of specific properties within the molecule.

This section will show a concrete pipeline of applying GNNs on molecular structures presented as SMILES strings to predict their molecule properties. The dataset is about SARS coronavirus 3C-like protease, and we will build a model to predict whether the molecule is active to 3C-like protease.

9.3.1 Introduction of AID1706 SARS CoV 3CL Dataset

SARS-CoV identified in 2003 causes severe acute respiratory syndrome (SARS). The virus encodes a polypeptide that is processed by two main proteases, one being the 3C-like protease (3CLpro). This protease is essential for viral gene expression and is a potential target for inhibitors that could inhibit SARS-CoV replication. Studies have

shown that 3CLpro is essential for the viral life cycle, making it a critical pathogenic component of SARS-CoV. The AID1706 SARS Cov 3CL dataset[6] records tens of thousands of molecules that could potentially be active to 3CLpro.

We load this dataset from DeepPurpose package[7], which contains 26640 molecules represented as SMILES strings and each is associated with an "active or not" binary label. For convenience, we only use the first 1000 SMILES to build the training and test datasets.

```
from DeepPurpose.dataset import *

# load AID1706 Assay Data
X_drugs, _, y = load_AID1706_SARS_CoV_3CL()

print (len(X_drugs), len(y))
"""
26640, 26640
"""

# we use the first 1000 molecules for this demo
X_drugs = X_drugs[:1000]
y = y[:1000]
```

Let us dive deeper and show what the dataset looks like:

```
# We look at the first 10 SMILES strings
print (X_drugs[:10])
"""
['CC1=C(SC(=N1)NC(=O)COC2=CC=CC=C2OC)C' 'CC1=CC=C(C=C1)C(=O)NCCCN2CCOCC2'
 'CSC1=CC=C(C=C1)C(=O)NC2CCSC3=CC=CC=C23'
 'CCOC(=O)N1CCC(CC1)N2CC34C=CC(O3)C(C4C2=O)C(=O)NC5=CC=C(C=C5)C'
 'CC1=CC(=NN1C(=O)C2=CC(=CC=C2)[N+](=O)[O-])[N+](=O)[O-])C'
 'CC1=CC=C(C=C1)C(=O)CSC2=NN=C(N2CC3=CC=CO3)CNC4=C(C=C(C=C4)C)C'
 'CC(C1=CC(=C(C=C1)Cl)Cl)NC(=O)CCl'
 'CCOC(=O)CN1CC23C=CC(O2)C(C3C1=O)C(=O)NC4=CC5=C(C=C4)OCO5'
 'COC(=O)C1=CC=C(C=C1)COC(=O)C2=CC(=C(N=C2)Cl)Cl'
 'C1=CC=C2C(=C1)C=C(C(=O)O2)C3=C(C=C(C=C3)NC(=O)CC4=CC=C(C=C4)Cl)Cl]
"""

# We look at the first 10 labels
print (y[:10])
"""
[0 0 0 0 0 1 1 0 0 0]
"""
```

9.3.2 Dataset Split and Processing

Given the clean SMILES strings and their labels, we would like to process each SMILES string into a molecule graph (get the atoms as nodes atom bonds as edges to extract the adjacency graph) and then learn the graph representations to predict the final binary label.

[6] https://pubchem.ncbi.nlm.nih.gov/bioassay/1706
[7] https://github.com/kexinhuang12345/DeepPurpose

First, we combine all SMILES and labels to be a `pandas.DataFrame` and split the data into training and test portions by 80% and 20%.

```
# split the dataset into 80%: 20%

data = pd.DataFrame(np.stack([X_drugs, y]).T, columns = ["SMILES", "Label"])
train, test = data.iloc[:int(len(data)*0.8)], data.iloc[int(len(data)*0.8):]

print (train)

"""
SMILES Label
0                  CC1=C(SC(=N1)NC(=O)COC2=CC=CC=C2OC)C      0
1                       CC1=CC=C(C=C1)C(=O)NCCCN2CCOCC2      0
2                CSC1=CC=C(C=C1)C(=O)NC2CCSC3=CC=CC=C23      0
3     CCOC(=O)N1CCC(CC1)N2CC34C=CC(O3)C(C4C2=O)C(=O)...      0
4     CC1=CC(=NN1C(=O)C2=CC(=CC(=C2)[N+](=O)[O-])[N+...      0
..                                                  ...    ...
795   COC1=CC=CC=C1N(CC(=O)NC2=CC(=C(C=C2)Cl)C(=O)OC...      1
796                        COC1=C(C=C(C=C1Cl)C(=O)NN)Cl      1
797   C1CC(C2=CC=CC=C2C1)NC(=O)CCC(=O)N3CCN(CC3)S(=O...      0
798   CC(=O)NC1=CC=C(C=C1)N(C(C2=CC=C(C=C2)OC)C(=O)N...      1
799   CC(C)N(CC1=CC=CC=C1)CC(COC2=CC=CC3=C2C(=CN3)CC...      0

[800 rows x 2 columns]
"""
```

To process the SMILES strings, we use the rdkit package[8]. The main function below is the `create_dataset` function, which takes the training or test datasets in and loops over the SMILES strings. Within the for-loop, `smiles` is the SMILES string of the current molecule, and `property` is the label.

```
def create_dataset(data_in, radius=2):
    dataset = []

    for smiles, property in data_in.values:
        try:
            """Create each data with the above defined functions."""
            mol = Chem.AddHs(Chem.MolFromSmiles(smiles))
            atoms = create_atoms(mol, atom_dict)
            molecular_size = len(atoms)
            i_jbond_dict = create_ijbonddict(mol, bond_dict)
            fingerprints = extract_fingerprints(radius, atoms, i_jbond_dict,
                                                fingerprint_dict, edge_dict)
            adjacency = Chem.GetAdjacencyMatrix(mol)

            """Transform the above each data of numpy
            to pytorch tensor on a device (i.e., CPU).
            """
            fingerprints = torch.LongTensor(fingerprints)
            adjacency = torch.FloatTensor(adjacency)

            dataset.append((fingerprints, adjacency, molecular_size, \
                                                    int(property)))
```

[8] https://www.rdkit.org/

```
        except:
            pass

    return dataset
```

We use `Chem.AddHs` and `Chem.MolFromSmiles` to transform the SMILES string to molecule format. Then, we use the `create_atoms`, `create_ijbonddict`, and `extract_fingerprints` functions to extract the nodes, edges, and initial node features (we use molecule fingerprints) for each molecule graph. The adjacency matrix is created by the `Chem.GetAdjacencyMatrix` function. Finally, the fingerprints as features, adjacency matrix, molecule size, and the final label are combined as a tuple to be appended into `dataset` list.

The try-except term here will prevent some erroneous SMILES strings that cause issues for `Chem` module. All the auxiliary functions (`create_atoms`, `create_ijbonddict`, and `extract_fingerprints`) are defined below.

```
from collections import defaultdict
import numpy as np
from rdkit import Chem
import torch


def create_atoms(mol, atom_dict):
    """Transform the atom types in a molecule (e.g., H, C, and O)
    into the indices (e.g., H=0, C=1, and O=2).
    Note that each atom index considers the aromaticity.
    """
    atoms = [a.GetSymbol() for a in mol.GetAtoms()]
    for a in mol.GetAromaticAtoms():
        i = a.GetIdx()
        atoms[i] = (atoms[i], 'aromatic')
    atoms = [atom_dict[a] for a in atoms]
    return np.array(atoms)


def create_ijbonddict(mol, bond_dict):
    """Create a dictionary, in which each key is a node ID
    and each value is the tuples of its neighboring node
    and chemical bond (e.g., single and double) IDs.
    """
    i_jbond_dict = defaultdict(lambda: [])
    for b in mol.GetBonds():
        i, j = b.GetBeginAtomIdx(), b.GetEndAtomIdx()
        bond = bond_dict[str(b.GetBondType())]
        i_jbond_dict[i].append((j, bond))
        i_jbond_dict[j].append((i, bond))
    return i_jbond_dict


def extract_fingerprints(radius, atoms, i_jbond_dict,
                         fingerprint_dict, edge_dict):
    """Extract the fingerprints from a molecular graph
    based on Weisfeiler-Lehman algorithm.
    """
```

```
    if (len(atoms) == 1) or (radius == 0):
        nodes = [fingerprint_dict[a] for a in atoms]

    else:
        nodes = atoms
        i_jedge_dict = i_jbond_dict

        for _ in range(radius):

            """Update each node ID considering its neighboring nodes and edges.
            The updated node IDs are the fingerprint IDs.
            """
            nodes_ = []
            for i, j_edge in i_jedge_dict.items():
                neighbors = [(nodes[j], edge) for j, edge in j_edge]
                fingerprint = (nodes[i], tuple(sorted(neighbors)))
                nodes_.append(fingerprint_dict[fingerprint])

            """Also update each edge ID considering
            its two nodes on both sides.
            """
            i_jedge_dict_ = defaultdict(lambda: [])
            for i, j_edge in i_jedge_dict.items():
                for j, edge in j_edge:
                    both_side = tuple(sorted((nodes[i], nodes[j])))
                    edge = edge_dict[(both_side, edge)]
                    i_jedge_dict_[i].append((j, edge))

            nodes = nodes_
            i_jedge_dict = i_jedge_dict_

    return np.array(nodes)
```

Finally, we could use this `create_dataset` function to process the training and test sets. The `atom_dict`, `bond_dict`, `fingerprint_dict`, and `edge_dict` are different dictionaries that are shared in training and test sets, while only the `fingerprint_dict` is used to calculate the number of fingerprint, which will be used to initialize the learnable tables for initial atom features.

```
"""Initialize x_dict, in which each key is a symbol type
(e.g., atom and chemical bond) and each value is its index.
"""
atom_dict = defaultdict(lambda: len(atom_dict))
bond_dict = defaultdict(lambda: len(bond_dict))
fingerprint_dict = defaultdict(lambda: len(fingerprint_dict))
edge_dict = defaultdict(lambda: len(edge_dict))

dataset_train = create_dataset(train[["SMILES", "Label"]])
dataset_test = create_dataset(test[["SMILES", "Label"]])

N_fingerprints = len(fingerprint_dict)
```

To this end, let us look at the summary of the first two molecules in the training set, following the example below, which includes:

- The atom index of the molecule (in tensor form), which will be used to index the learnable atom embeddings in the later molecule graph neural network class.
- Adjacency matrix of the molecular graph (in the tensor form).
- Molecule size (such as 36).
- Molecule property label (such as 0).

```
# look at the first 2 data points in training
# they are at (fingerprints, adjacency, molecular_size, property) structure
print (dataset_train[:2])

"""
[(tensor([17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 30, 29, 28,
31, 32, 33, 34, 34, 34, 35, 36, 36, 37, 37, 37, 37, 38, 38, 38, 34, 34, 34]),
tensor([[0., 1., 0.,  ..., 0., 0., 0.],
        [1., 0., 1.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]]), 36, 0),

(tensor([46, 47, 48, 48, 49, 48, 48, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
58, 57, 34, 34, 34, 37, 37, 37, 37, 60, 61, 61, 62, 62, 61, 61, 61, 61, 36,
        36, 36, 36, 61, 61]),
tensor([[0., 1., 0.,  ..., 0., 0., 0.],
        [1., 0., 1.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]]), 41, 0)]
"""
```

9.3.3 Graph neural networks for molecules

Next, we define our graph neural networks (GNN) for the molecules, which are represented as graphs, with atoms as nodes and chemical bonds as edges. We assume the atom set (node set) as $\mathcal{N}$, while $|\mathcal{N}| = N$ is the number of nodes in the molecular graph. We assume the bond set (edge set) as $\mathcal{E}$, and any two connected atoms $u, v \in \mathcal{N}$ form a bond $(u, v) \in \mathcal{E}$. Each atom and bond in the molecule graph possesses associated features, such as atom type, solubility, bond type, electronegativity, or other physicochemical properties.

Assuming the initial node embeddings are represented as $\mathbf{H}^{(0)} \in \mathbb{R}^{N\times d}$, we adopt a simple variant of the GCN model by utilizing the following update function,

$$\mathbf{Z}^{(t)} = \mathrm{ReLU}(\mathbf{H}^{(t-1)}W^{(t)}), \tag{9.15}$$

$$\mathbf{H}^{(t)} = (\mathbf{A} + \mathbf{I})\mathbf{Z}^{(t)}, \tag{9.16}$$

where we use non-linear projection first to transform the last layer hidden embedding, and then apply an un-normalized adjacency with self-loop (normalization could prevent

potential numerical issues while un-normalized adjacency works fine here) to aggregate the neighborhood information to be the current hidden node embedding. The output graph embedding is calculated by summing over all the node embeddings and then we apply a final linear layer to obtain the predicted probability of molecule property.

In the implementation, the `self.forward()` function of the model takes two steps: (i) use gnn with molecule adjacency matrix to get molecular graph embeddings; (ii) apply a property prediction layer on top of the graph embedding to predict the binary property label. The `self.gnn()` part is the core of the `MoleculeGNN` class, which implements batch processing for molecule graphs with different sizes using padding.

```
class MolecularGNN(nn.Module):
    """
    based on https://github.com/masashitsubaki/molecularGNN_smiles
    """
    def __init__(self, N_fingerprints, dim, layer_gnn_hidden):
        super(MolecularGNN, self).__init__()
        # learnable atom initial features
        self.embed_fingerprint = nn.Embedding(N_fingerprints, dim)
        # gnn layers (will be used together with the adj in self.gnn)
        self.W_fingerprint = nn.ModuleList([nn.Linear(dim, dim)
                                            for _ in range(layer_gnn_hidden)])
        # final prediction layers
        self.W_property = nn.Linear(dim, 1)

    def pad(self, matrices, pad_value):
        """Pad the list of matrices
        with a pad_value (e.g., 0) for batch processing.
        For example, given a list of matrices [A, B, C],
        we obtain a new matrix [A00, 0B0, 00C],
        where 0 is the zero (i.e., pad value) matrix.
        """
        shapes = [m.shape for m in matrices]
        M, N = sum([s[0] for s in shapes]), sum([s[1] for s in shapes])
        zeros = torch.FloatTensor(np.zeros((M, N)))
        pad_matrices = pad_value + zeros
        i, j = 0, 0
        for k, matrix in enumerate(matrices):
            m, n = shapes[k]
            pad_matrices[i:i+m, j:j+n] = matrix
            i += m
            j += n
        return pad_matrices

    def update(self, matrix, vectors, layer):
        hidden_vectors = torch.relu(self.W_fingerprint[layer](vectors))
        return hidden_vectors + torch.matmul(matrix, hidden_vectors)

    def sum(self, vectors, axis):
        sum_vectors = [torch.sum(v, 0) for v in torch.split(vectors, axis)]
        return torch.stack(sum_vectors)

    def mean(self, vectors, axis):
        mean_vectors = [torch.mean(v, 0) for v in torch.split(vectors, axis)]
        return torch.stack(mean_vectors)
```

```
    def gnn(self, inputs):
        """Cat or pad each input data for batch processing."""
        fingerprints, adjacencies, molecular_sizes = inputs
        fingerprints = torch.cat(fingerprints)
        adjacencies = self.pad(adjacencies, 0)

        """GNN layer (update the fingerprint vectors)."""
        fingerprint_vectors = self.embed_fingerprint(fingerprints)
        for l in range(len(self.W_fingerprint)):
            hs = self.update(adjacencies, fingerprint_vectors, l)

        """Molecular vector by sum or mean of the fingerprint vectors."""
        molecular_vectors = self.sum(fingerprint_vectors, molecular_sizes)
        # molecular_vectors = self.mean(fingerprint_vectors, molecular_sizes)
        return molecular_vectors

    def forward(self, data_batch):
        molecular_vectors = self.gnn(data_batch)
        predicted_scores = self.W_property(molecular_vectors)
        return predicted_scores

model = MolecularGNN(N_fingerprints, 128, 2)
```

9.3.4 GNN training and evaluation

We then call the binary cross entropy loss to calculate the loss function (we use `torch.nn.BCEWithLogitsLoss()` since it includes a Sigmoid function on top of our output, which is more numerically stable than using a plain `torch.nn.Sigmoid` followed by a `torch.nn.BCELoss` [9]) and use AdamW optimizer to calculate the gradients.

The model is trained for 15 epochs, while after each epoch, the AUROC score will be computed based on the test set. The training procedures are similar to the previous PyTorch model training pipeline. Essentially, we find that the loss function is going down quickly and the AUROC gradually climbs up to almost a perfect score.

```
from sklearn.metrics import roc_auc_score

batch_size = 32
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001)

for epoch in range(15):
    """ model training """
    np.random.shuffle(dataset_train)
    train_loss = 0

    model.train()
    for i in range(0, len(dataset_train), batch_size):
```

[9] This is because by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

```
        data_batch = list(zip(*dataset_train[i: i+batch_size]))

        # feed features into the model
        pred = model.forward(data_batch[:3]).squeeze(-1)

        # use gt property as labels
        label = torch.FloatTensor(data_batch[-1])
        loss = criterion(pred, label)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    """ model evaluation """
    model.eval()

    predicted, groundtruth = [], []
    with torch.no_grad():
        for i in range(0, len(dataset_test), batch_size):
            data_batch = list(zip(*dataset_test[i: i+batch_size]))
            # feed features into the model
            pred = model.forward(data_batch[:3]).squeeze(-1).numpy().tolist()
            label = data_batch[-1]

            predicted += pred
            groundtruth += label

    print (f"--- epoch: {epoch} ---, train loss: {train_loss}, \
                    test AUROC: {roc_auc_score(groundtruth, predicted)}")

"""
--- epoch: 0 ---, train loss: 39.3298280239, test AUROC: 0.5515151515151515
--- epoch: 1 ---, train loss: 24.1569204330, test AUROC: 0.6504513140027158
--- epoch: 2 ---, train loss: 18.3505403399, test AUROC: 0.7955907021327583
--- epoch: 3 ---, train loss: 15.5090635120, test AUROC: 0.8324640937174036
--- epoch: 4 ---, train loss: 12.8106079697, test AUROC: 0.8664265706282513
--- epoch: 5 ---, train loss: 11.1026673614, test AUROC: 0.9002656363197294
--- epoch: 6 ---, train loss: 9.69138415157, test AUROC: 0.9347860791826309
--- epoch: 7 ---, train loss: 8.68385870754, test AUROC: 0.9528820856254485
--- epoch: 8 ---, train loss: 7.63521204888, test AUROC: 0.9483418367346939
--- epoch: 9 ---, train loss: 6.85020373761, test AUROC: 0.969544766004943
--- epoch: 10 ---, train loss: 6.0492331385, test AUROC: 0.9728867623604466
--- epoch: 11 ---, train loss: 5.4066562131, test AUROC: 0.989516129032258
--- epoch: 12 ---, train loss: 4.9491942748, test AUROC: 0.9864766964501328
--- epoch: 13 ---, train loss: 4.4005933329, test AUROC: 0.9948178266762338
--- epoch: 14 ---, train loss: 3.9376147910, test AUROC: 0.9946519795657727
"""
```

If readers want to practice by their own, a complete notebook can be found in this public repository[10].

[10] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap7-GNN/notebook

9.4 Clinical Predictive Modeling with GNNs

In this section, we are going to revisit the chest X-ray image classification challenge using Graph Neural Networks (GNNs). In previous sections, Convolutional Neural Network (CNN) models were employed for chest X-ray image classification, treating each patient's image data as an independent sample. However, GNNs offer a novel approach that capitalizes on leveraging demographic similarities to potentially enhance predictions. This novel approach operates under the premise that patients with similar demographic features may exhibit similar labels in their X-ray images.

The approach involves constructing a comprehensive graph where *each node represents a single patient* (we assume each patient only has a single x-ray image) Crucially, the connection between nodes in this graph is established based on the similarity of patient demographics. Consequently, during training, the embeddings of neighboring nodes play a pivotal role in updating the embedding of the current node. To manage instances where numerous images are connected to a single image, a neighborhood sampling strategy, akin to the one proposed in GraphSAGE by Hamilton et al. [77], is employed. This strategy aids in managing and updating node embeddings effectively in a scalable way, optimizing the learning process within the GNN framework for enhanced chest X-ray image classification.

We will use pyhealth modules to implement the whole pipeline.

Graph structures to improve the prediction accuracy

Comparing Graph Neural Networks (GNNs) with straightforward Deep Neural Networks (DNNs) reveals a distinctive approach. GNN-based models incorporate external information to construct a graph structure encompassing all data samples. Subsequently, leveraging graph convolutions, these models borrow information from neighboring samples within the graph. This methodology hinges on the assumption that samples connected within the graph should exhibit similarities in their respective labels, thereby enhancing the model's predictive capabilities.

This approach is not limited to chest X-ray image classification; it is applicable across various healthcare domains. For instance, in spatio-temporal disease prediction, geographical data serves as the foundation for constructing the graph. The underlying assumption here is that diseases might spread more easily among nearby geographic locations. By creating a graph based on spatial proximity, GNNs can capitalize on this geographical information to improve disease prediction models. This strategy aligns with the notion that regions with geographic proximity might exhibit similarities in disease prevalence or transmission patterns, which GNNs can effectively leverage to enhance predictive accuracy.

9.4.1 Step 1: data loading

The whole pipeline starts with data loading, for which we use the `COVID19CXRDataset` API similar in Section 6.3. This API only requires the root

path to the chest X-ray as the argument, and everything else is handled by the API already.

```
from pyhealth.datasets import COVID19CXRDataset

root = "/srv/local/data/COVID-19_Radiography_Dataset"
base_dataset = COVID19CXRDataset(root)
```

9.4.2 Step 2: define machine learning task

As usual, we apply the `set_task()` function for processing the dataset into a listed dictionary structure, where the information of each sample is formatted as a dictionary. We can see that this dataset itself has a default task which is to classify the chest Xray images into different categories. We then directly use the default task for processing.

```
base_dataset.default_task
# COVID19CXRClassification(task_name='COVID19CXRClassification', \
    input_schema={'path': 'image'}, output_schema={'label': 'label'})

sample_dataset = base_dataset.set_task()
```

Similar to Section 6.3, we add one more sample transformation step to further clean up the image data, which involves enforcing channel consistency, resizing and normalizing images. The transformation is made possible with the featurizer design in the module. We do not show the data transformation for other pipelines for simplicity.

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.Lambda(lambda x: x if x.shape[0] == 3 else x.repeat(3, 1, 1)),
    transforms.Resize((224, 224)),
    transforms.Normalize(mean=[0.5862785803043838], std=[0.27950088968644304])
])

def encode(sample):
    sample["path"] = transform(sample["path"])
    return sample

sample_dataset.set_transform(encode)
```

After data transformation, as usual, we split the data into training, validation, and test by 70% : 10% : 20%. This time, we leverage another data splitter, which is split by sample, meaning that we do not care whether the data from the same patients are in the same datasets (either training, validation, or test).

```
from pyhealth.datasets import split_by_sample

# Get Index of train, valid, test set
train_index, val_index, test_index = split_by_sample(
    dataset=sample_dataset,
    ratios=[0.7, 0.1, 0.2],
    get_index = True
)
```

9.4.3 Step 3: initialize ML model

PyHealth provides a set of graph neural network models combined with advanced CNN backbones in the `Graph_TorchvisionModel` API. In this example, we initialize the resnet18 backbone while there are other choices, such as *resnet*34, *resnet*50, *densenet*121.

```
from pyhealth.models import Graph_TorchvisionModel

model = Graph_TorchvisionModel(
        dataset=sample_dataset,
        feature_keys=["path"],
        label_key="label",
        mode="multiclass",
        model_name="resnet18",
        model_config={},
        gnn_config={"input_dim": 256, "hidden_dim": 128},
    )
```

A new procedure in this pipeline is that we need to build an image graph structure. The demographic information of each patient is already stored in `sample_dataset` and our initialized model will build a graph based on the information, so we input the `sample_dataset` as an argument.

Next, we leverage the neighborhood sampler to simplify the graph and sparsify the connections by the GraphSAGE model [77]. After doing so, the training, validation, and test sets are prepared for training and evaluation.

```
from pyhealth.sampler import NeighborSampler

graph = model.build_graph(sample_dataset, random = True)

# Define Sampler as Dataloader
train_dataloader = NeighborSampler(sample_dataset, graph["edge_index"],
                                   node_idx=train_index, sizes=[15, 10],
                                   batch_size=64, shuffle=True,
                                   num_workers=12)

# We sample all edges connected to target node for validation and test (Sizes =
    [-1, -1])
valid_dataloader = NeighborSampler(sample_dataset, graph["edge_index"],
                                   node_idx=val_index, sizes=[-1, -1],
                                   batch_size=64, shuffle=False,
                                   num_workers=12)
test_dataloader = NeighborSampler(sample_dataset, graph["edge_index"], node_idx
                                   =test_index, sizes=[-1, -1], batch_size
                                   =64, shuffle=False, num_workers=12)
```

9.4.4 Step 4&5: model training and inference

As usual, we use the typical `pyhealth.trainer` to train the model and use "accuracy" as the monitoring metric. We then evaluate the model on the test set.

```
from pyhealth.trainer import Trainer

resnet_trainer = Trainer(model=model, device="cpu")
resnet_trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=valid_dataloader,
    epochs=10,
    monitor="accuracy",
)
```

After training the model for 10 epochs, we could evaluate the model performance on the test set.

```
print(resnet_trainer.evaluate(test_dataloader))
"""
{'accuracy': 0.4786590097780537, 'f1_macro': 0.1618557783142647, 'f1_micro': 0.
                                    4786590097780537, 'loss': 1.
                                    256981566770753}
"""
```

A complete Jupyter notebook of this example can be found in the repository [11].

[11] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap7-GNN/notebook

9.5 Takeaways

- **Graph Data:** Many real-world data are irregular and are represented as graphs (compared to images and time-series which are regular grid-like data). Specifically, graph is a data structure that consists of a set of nodes and a set of edges connecting these nodes.
- **Common Graph Neural Networks (GNN):** Graph convolutional networks (GCN), graph attention network (GAT), message passing neural networks (MPNN) are common GNN models. They differ in that GCN treats all edges equally, GAT learns the edge weights by attention modules, and MPNN allows flexible aggregation mechanism.
- **Advancements in GNN Training:** Some key components of modern GNN training tricks are practically useful, such as neighborhood sampling, distributed graph partition and training, heterogeneous graph modeling.
- **GNN Applications in CV and NLP:** In the domains of computer vision and natural language processing, researchers basically construct different types of graph structures from data and then apply the GNN model to learn node or graph embeddings.
- **Molecule Property Prediction Example:** GNN is powerful in learning molecule graph structures by treating atom as nodes and bonds as edges for predicting the molecular structure properties.
- **PyHealth GNN code examples:** GNN could be applied on similarity graphs constructed by patient demongraphics, which could further enhance the predictive model, such as on ChestXray image classification.

This chapter presents the Transformer architecture, pre-training strategies employed by models like BERT and diverse applications in healthcare enabled by this breakthrough technique. The key components and innovations of Transformers are the foundation for many AI applications including large language models.

Questions

- What are the key elements of a graph? What is the adjacency matrix of a graph? What are the degree matrix of a graph?
- Comparing two different ways of normalizing the adjacency graph?
- In the section, we discuss the undirected graph. However, the concepts of adjacency matrix, degree matrix, and normalized adjacency graph can be generalized to directed graph as well. Could you specify these concepts for the graph shown in Figure 9.2.
- What are graph neural networks? What are the key differences between GNNs and DNNs? Could you explain the difference between GCN and multi-layer perceptron (both use ReLU as activation)?

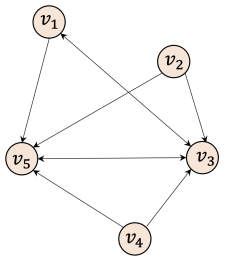

Figure 9.2 A directed graph

- Could you elaborate on the difference between GCN and GAT? Why MPNN is a general form? Could you use MPNN equation (i.e., Message and Update) function to formulate the GCN and GAT networks?
- Why is GNN suitable for molecule property prediction? Could you choose your own favorite molecule graph and your favorite GNN models and explain how to use this GNN model to model the molecule graph structure?
- In Section **??**, we connect the graph by patient demographics features, could you connect a new Xray image graph by image pixel-level similarity and re-implement the whole pipeline again. Show the results and explain why two different graphs lead to different final prediction results.