18

Graph Neural Networks for Capturing Dependencies in Graph Structured Data

In this chapter, we will introduce a class of deep learning models that operates on graph data, namely, **graph neural networks** (**GNNs**). GNNs have been an area of rapid development in recent years. According to the *State of AI* report from 2021 (`https://www.stateof.ai/2021-report-launch.html`), GNNs have evolved "from niche to one of the hottest fields of AI research."

GNNs have been applied in a variety of areas, including the following:

- Text classification (`https://arxiv.org/abs/1710.10903`)
- Recommender systems (`https://arxiv.org/abs/1704.06803`)
- Traffic forecasting (`https://arxiv.org/abs/1707.01926`)
- Drug discovery (`https://arxiv.org/abs/1806.02473`)

While we can't cover every new idea in this rapidly developing space, we'll provide a basis to understand how GNNs function and how they can be implemented. In addition, we'll introduce the **PyTorch Geometric** library, which provides resources for managing graph data for deep learning as well as implementations of many different kinds of graph layers that you can use in your deep learning models.

The topics that will be covered in this chapter are as follows:

- An introduction to graph data and how it can be represented for use in deep neural networks
- An explanation of graph convolutions, a major building block of common GNNs
- A tutorial showing how to implement GNNs for molecular property prediction using PyTorch Geometric
- An overview of methods at the cutting edge of the GNN field

Introduction to graph data

Broadly speaking, graphs represent a certain way we describe and capture relationships in data. Graphs are a particular kind of data structure that is nonlinear and abstract. And since graphs are abstract objects, a concrete representation needs to be defined so the graphs can be operated on. Furthermore, graphs can be defined to have certain properties that may require different representations. *Figure 18.1* summarizes the common types of graphs, which we will discuss in more detail in the following subsections:

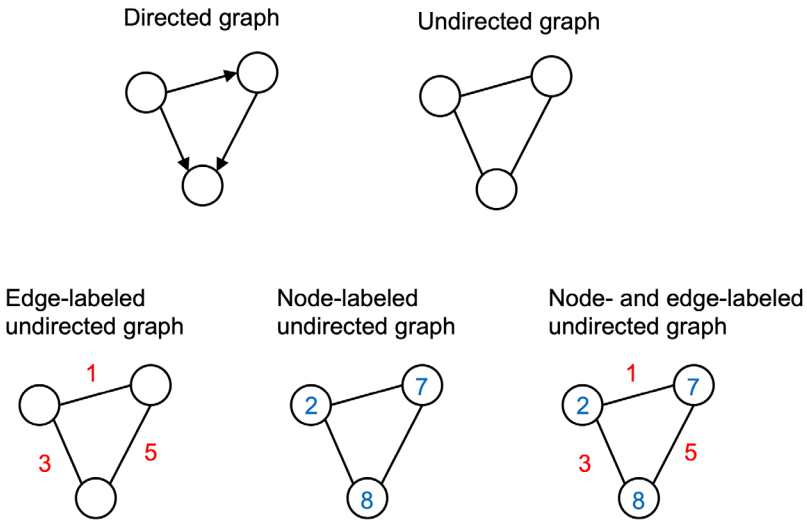

Figure 18.1: Common types of graphs

Undirected graphs

An **undirected graph** consists of **nodes** (in graph theory also often called **vertices**) that are connected via edges where the order of the nodes and their connection does not matter. *Figure 18.2* sketches two typical examples of undirected graphs, a friend graph, and a graph of a chemical molecule consisting of atoms connected through chemical bonds (we will be discussing such molecular graphs in much more detail in later sections):

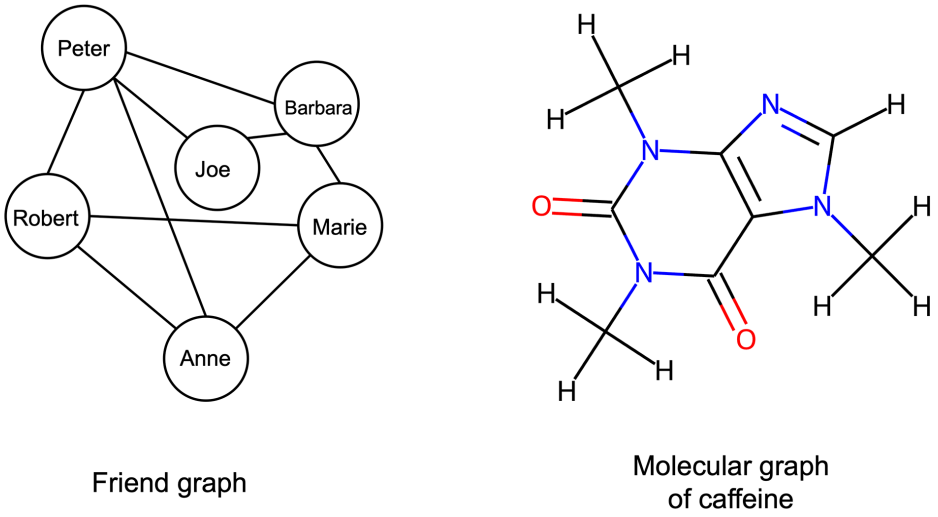

Figure 18.2: Two examples of undirected graphs

Other common examples of data that can be represented as undirected graphs include images, protein-protein interaction networks, and point clouds.

Mathematically, an undirected graph G is a pair (V, E), where V is a set of the graph's nodes, and E is the set of edges making up the paired nodes. The graph can then be encoded as a $|V| \times |V|$ **adjacency matrix** $\boldsymbol{A}$. Each element x_{ij} in matrix $\boldsymbol{A}$ is either a 1 or a 0, with 1 denoting an edge between nodes i and j (vice versa, 0 denotes the absence of an edge). Since the graph is undirected, an additional property of $\boldsymbol{A}$ is that $x_{ij} = x_{ji}$.

Directed graphs

Directed graphs, in contrast to undirected graphs discussed in the previous section, connect nodes via *directed* edges. Mathematically they are defined in the same way as an undirected graph, except that E, the set of edges, is a set of *ordered* pairs. Therefore, element x_{ij} of $\boldsymbol{A}$ does need not equal x_{ji}.

An example of a directed graph is a citation network, where nodes are publications and edges from a node are directed toward the nodes of papers that a given paper cited.

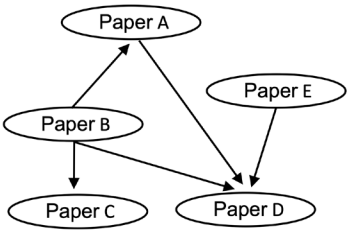

Figure 18.3: An example of a directed graph

Labeled graphs

Many graphs we are interested in working with have additional information associated with each of their nodes and edges. For example, if you consider the caffeine molecule shown earlier, molecules can be represented as graphs where each node is a chemical element (for example, O, C, N, or H atoms) and each edge is the type of bond (for example, single or double bond) between its two nodes. These node and edge features need to be encoded in some capacity. Given graph G, defined by the node set and edge set tuple (V, E), we define a $|V| \times f_V$ node feature matrix $\boldsymbol{X}$, where f_V is the length of the label vector of each node. For edge labels, we define an $|E| \times f_E$ edge feature matrix $\boldsymbol{X}_E$, where f_E is the length of the label vector of each edge.

Molecules are an excellent example of data that can be represented as a **labeled graph**, and we will be working with molecular data throughout the chapter. As such, we will take this opportunity to cover their representation in detail in the next section.

Representing molecules as graphs

As a chemical overview, molecules can be thought of as groups of atoms held together by chemical bonds. There are different atoms corresponding to different chemical elements, for example, common elements include carbon (C), oxygen (O), nitrogen (N), and hydrogen (H). Also, there are different kinds of bonds that form the connection between atoms, for example, single or double bonds.

We can represent a molecule as an undirected graph with a node label matrix, where each row is a one-hot encoding of the associated node's atom type. Additionally, there is an edge label matrix where each row is a one-hot encoding of the associated edge's bond type. To simplify this representation, hydrogen atoms are sometimes made implicit since their location can be inferred with basic chemical rules. Considering the caffeine molecule we saw earlier, an example of a graph representation with implicit hydrogen atoms is shown in *Figure 18.4*:

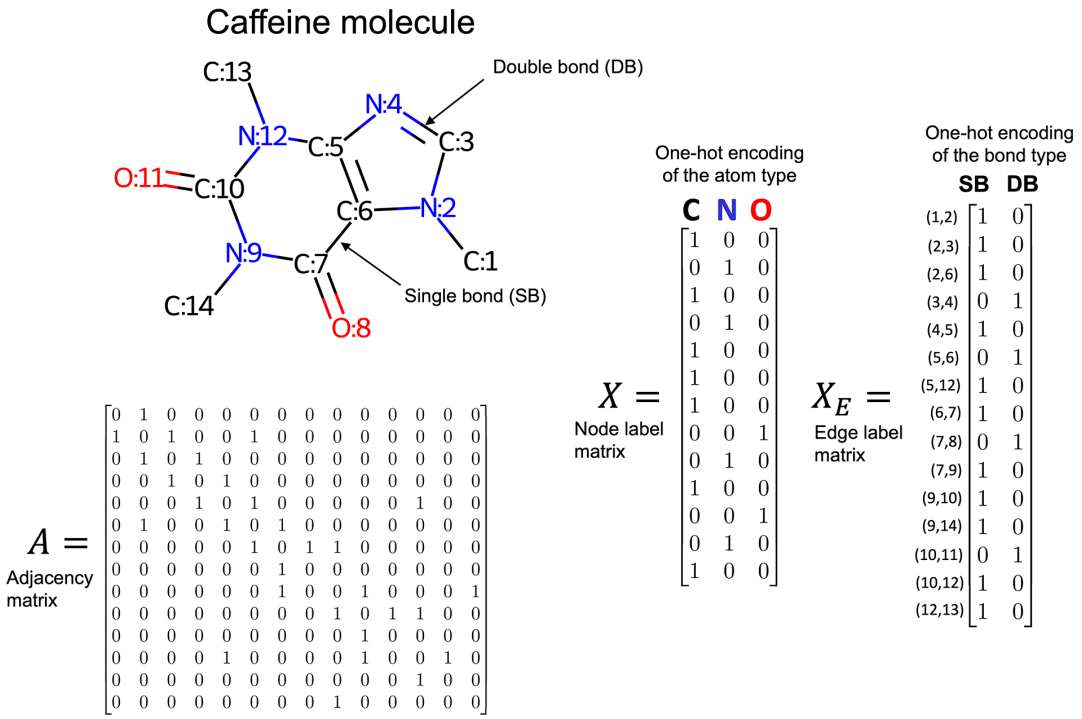

Figure 18.4: Graph representation of a caffeine molecule

Understanding graph convolutions

The previous section showed how graph data can be represented. The next logical step is to discuss what tools we have that can effectively utilize those representations.

In the following subsections, we will introduce graph convolutions, which are the key component for building GNNs. In this section, we'll see why we want to use convolutions on graphs and discuss what attributes we want those convolutions to have. We'll then introduce graph convolutions through an implementation example.

The motivation behind using graph convolutions

To help explain graph convolutions, let's briefly recap how convolutions are utilized in convolutional neural networks (CNNs), which we discussed in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*. In the context of images, we can think of a convolution as the process of sliding a convolutional filter over an image, where, at each step, a weighted sum is computed between the filter and the receptive field (the part of the image it is currently on top of).

As discussed in the CNN chapter, the filter can be viewed as a detector for a specific feature. This approach to feature detection is well-suited for images for several reasons, for instance, the following priors we can place on image data:

1. **Shift-invariance**: We can still recognize a feature in an image regardless of where it is located (for example, after translation). A cat can be recognized as a cat whether it is in the top left, bottom right, or another part of an image.
2. **Locality**: Nearby pixels are closely related.
3. **Hierarchy**: Larger parts of an image can often be broken down into combinations of associated smaller parts. A cat has a head and legs; the head has eyes and a nose; the eyes have pupils and irises.

Interested readers can find a more formal description of these priors, and priors assumed by GNNs, in the 2019 article *Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology*, by *N. Dehmamy*, *A.-L. Barabasi*, and *R. Yu* (`https://arxiv.org/abs/1907.05008`).

Another reason convolutions are well-suited for processing images is that the number of trainable parameters does not depend on the dimensionality of the input. You could train a series of 3×3 convolutional filters on, for example, a 256×256 or a 9×9 image. (However, if the same image is presented in different resolutions, the receptive fields and, therefore, the extracted features will differ. And for higher-resolution images, we may want to choose larger kernels or add additional layers to extract useful features effectively.)

Like images, graphs also have natural priors that justify a convolutional approach. Both kinds of data, images and graphs, share the locality prior. However, how we define locality differs. In images, the prior is on locality in 2D space, while with graphs, it is structural locality. Intuitively, this means that a node that is one edge away is more likely to be related than a node five edges away. For example, in a citation graph, a directly cited publication, which would be one edge away, is more likely to have similar subject matter than a publication with multiple degrees of separation.

A strict prior for graph data is **permutation invariance**, which means that the ordering of the nodes does not affect the output. This is illustrated in *Figure 18.5*, where changing the ordering of a graph's nodes does not change the graph's structure:

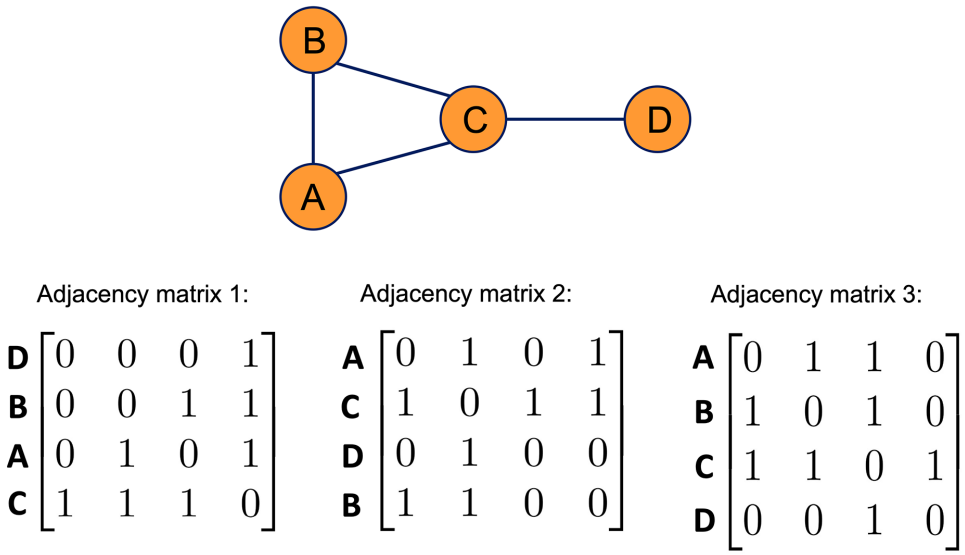

Figure 18.5: Different adjacency matrices representing the same graph

Since the same graph can be represented by multiple adjacency matrices, as illustrated in *Figure 18.5*, consequently, any graph convolution needs to be permutation invariant.

A convolutional approach is also desirable for graphs because it can function with a fixed parameter set for graphs of different sizes. This property is arguably even more important for graphs than images. For instance, there are many image datasets with a fixed resolution where a fully connected approach (for example, using a multilayer perceptron) could be possible, as we have seen in *Chapter 11, Implementing a Multilayer Artificial Neural Network from Scratch*. In contrast, most graph datasets contain graphs of varying sizes.

While image convolutional operators are standardized, there are many different kinds of graph convolutions, and the development of new graph convolutions is a very active area of research. Our focus is on providing general ideas so that readers can rationalize about the GNNs they wish to utilize. To this end, the following subsection will show how to implement a basic graph convolution in PyTorch. Then, in the next section, we will construct a simple GNN in PyTorch from the ground up.

Implementing a basic graph convolution

In this subsection, we will introduce a basic graph convolution function and see what happens when it is applied to a graph. Consider the following graph and its representation:

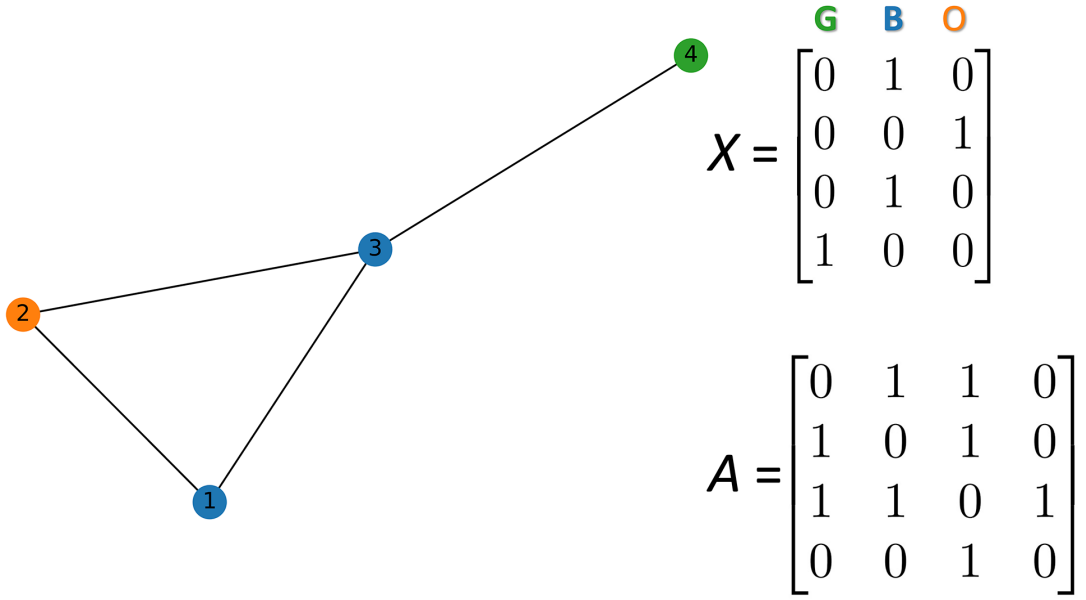

Figure 18.6: A representation of a graph

Figure 18.6 depicts an undirected graph with node labels specified by an $n \times n$ adjacency matrix $\boldsymbol{A}$ and $n \times f_{in}$ node feature matrix $\boldsymbol{X}$, where the only feature is a one-hot representation of each node's color—green (G), blue (B), or orange (O).

One of the most versatile libraries for graph manipulation and visualization is NetworkX, which we will be using to illustrate how to construct graphs from a label matrix $\boldsymbol{X}$ and a node matrix $\boldsymbol{A}$.

Installing NetworkX

NetworkX is a handy Python library for manipulating and visualizing graphs. It can be installed via `pip`:

```
pip install networkx
```

We used version 2.6.2 to create the graph visualizations in this chapter. For more information, please visit the official website at `https://networkx.org`.

Using NetworkX, we can construct the graph shown in *Figure 18.6* as follows:

```
>>> import numpy as np
>>> import networkx as nx
```

```
>>> G = nx.Graph()
... # Hex codes for colors if we draw graph
>>> blue, orange, green = "#1f77b4", "#ff7f0e", "#2ca02c"
>>> G.add_nodes_from([
...     (1, {"color": blue}),
...     (2, {"color": orange}),
...     (3, {"color": blue}),
...     (4, {"color": green})
... ])
>>> G.add_edges_from([(1,2), (2,3), (1,3), (3,4)])
>>> A = np.asarray(nx.adjacency_matrix(G).todense())
>>> print(A)
[[0 1 1 0]
[1 0 1 0]
[1 1 0 1]
[0 0 1 0]]

>>> def build_graph_color_label_representation(G, mapping_dict):
...     one_hot_idxs = np.array([mapping_dict[v] for v in
...         nx.get_node_attributes(G, 'color').values()])
>>>     one_hot_encoding = np.zeros(
...         (one_hot_idxs.size, len(mapping_dict)))
>>>     one_hot_encoding[
...         np.arange(one_hot_idxs.size), one_hot_idxs] = 1
>>>     return one_hot_encoding
>>> X = build_graph_color_label_representation(
...     G, {green: 0, blue: 1, orange: 2})
>>> print(X)
[[0., 1., 0.],
[0., 0., 1.],
[0., 1., 0.],
[1., 0., 0.]]
```

To draw the graph constructed in the preceding code, we can then use the following code:

```
>>> color_map = nx.get_node_attributes(G, 'color').values()
>>> nx.draw(G,with_labels=True, node_color=color_map)
```

In the preceding code example, we first initiated a new `Graph` object from NetworkX. We then added nodes 1 to 4 together with color specifications for visualization. After adding the nodes, we specified their connections (edges). Using the `adjacency_matrix` constructor from NetworkX, we create the adjacency matrix A, and our custom `build_graph_color_label_representation` function creates the node label matrix X from the information we added to the `Graph` object earlier.

With graph convolutions, we can interpret each row of $\boldsymbol{X}$ as being an embedding of the information that is stored at the node corresponding to that row. Graph convolutions update the embeddings at each node based on the embeddings of their neighbors and themselves. For our example implementation, the graph convolution will take the following form:

$$\boldsymbol{x}_i' = \boldsymbol{x}_i \boldsymbol{W}_1 + \sum_{j \in N(i)} \boldsymbol{x}_j \boldsymbol{W}_2 + b$$

Here, $\boldsymbol{x}_i'$ is the updated embedding for node i; $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are $f_{in} \times f_{out}$ matrices of learnable filter weights; and b is a learnable bias vector of length f_{out}.

The two weight matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ can be considered filter banks, where each column is an individual filter. Note that this filter design is most effective when the locality prior on graph data holds. If a value at a node is highly correlated with the value at another node many edges away, a single convolution will not capture that relationship. Stacking convolutions will capture more distant relationships, as illustrated in *Figure 18.7* (we set the bias to zero for simplicity):

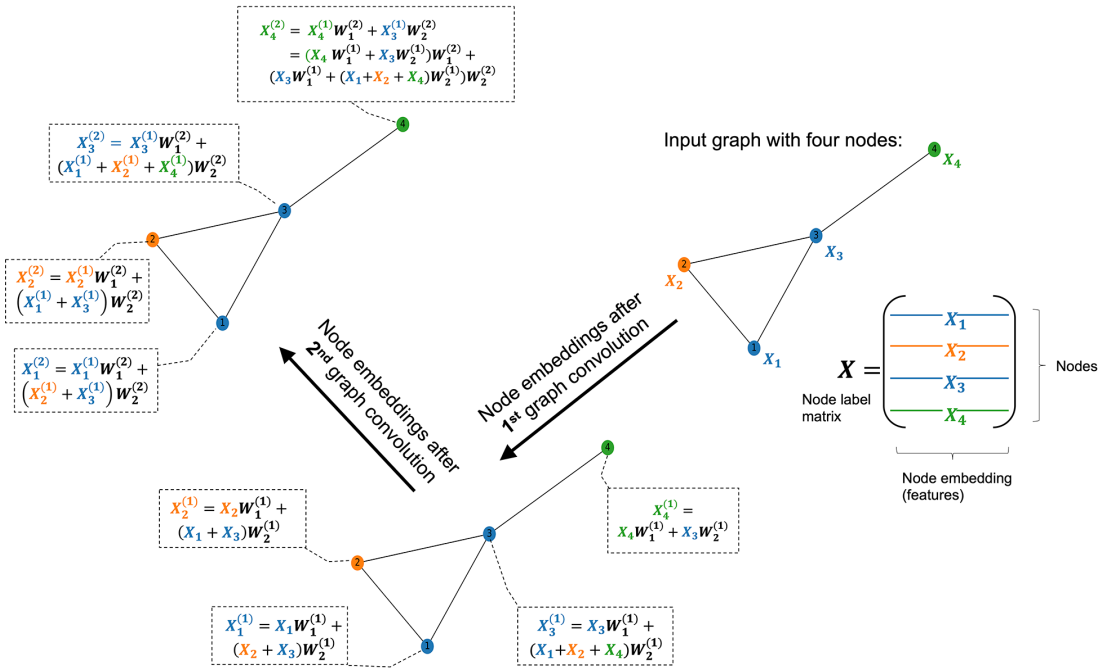

Figure 18.7: Capturing relationships from a graph

The design of the graph convolution illustrated in *Figure 18.7* fits our priors on graph data, but it may not be clear how to implement the sum over neighbors in matrix form. This is where we utilize the adjacency matrix $\boldsymbol{A}$. The matrix form of this convolution is $\boldsymbol{XW}_1 + \boldsymbol{AXW}_2$. Here, the adjacency matrix, consisting of 1s and 0s, acts as a mask to select nodes and compute the desired sums. In NumPy, initializing this layer and computing a forward pass on the previous graph could be written as follows:

```
>>> f_in, f_out = X.shape[1], 6
>>> W_1 = np.random.rand(f_in, f_out)
>>> W_2 = np.random.rand(f_in, f_out)
>>> h = np.dot(X, W_1)+ np.dot(np.dot(A,X), W_2)
```

Computing a forward pass of a graph convolution is that easy.

Ultimately, we want a graph convolutional layer to update the representation of the node information encoded in $\boldsymbol{X}$ by utilizing the structural (connectivity) information provided by $\boldsymbol{A}$. There are many potential ways to do this, and this plays out in the numerous kinds of graph convolutions that have been developed.

To talk about different graph convolutions, generally, it would be nice for them to have a unifying framework. Thankfully, such a framework was presented in *Neural Message Passing for Quantum Chemistry* by *Justin Gilmer* and colleagues, 2017, `https://arxiv.org/abs/1704.01212`.

In this **message-passing** framework, each node in the graph has an associated hidden state $h_i^{(t)}$, where i is the node's index at time step t. The initial value $h_i^{(0)}$ is defined as $\boldsymbol{X}_i$, which is the row of $\boldsymbol{X}$ associated with node i.

Each graph convolution can be split into a message-passing phase and a node update phase. Let $N(i)$ be the neighbors of node i. For undirected graphs, $N(i)$ is the set of nodes that share an edge with node i. For directed graphs, $N(i)$ is the set of nodes that have an edge whose endpoint is node i. The message-passing phase can be formulated as follows:

$$m_i = \sum_{j \in N(i)} M_t\left(h_i^{(t)}, h_j^{(t)}, e_{ij}\right)$$

Here, M_t is a message function. In our example layer, we define this message function as $M_t = h_j^{(t)} W_2$. The node update phase with the update function U_t is $h_i^{(t+1)} = U_t\left(h_i^{(t)}, m_i\right)$. In our example layer, this update is $h_i^{(t+1)} = h_i^{(t)} W_1 + m_i + b$.

Figure 18.8 visualizes the message-passing idea and summarizes the convolution we have implemented:

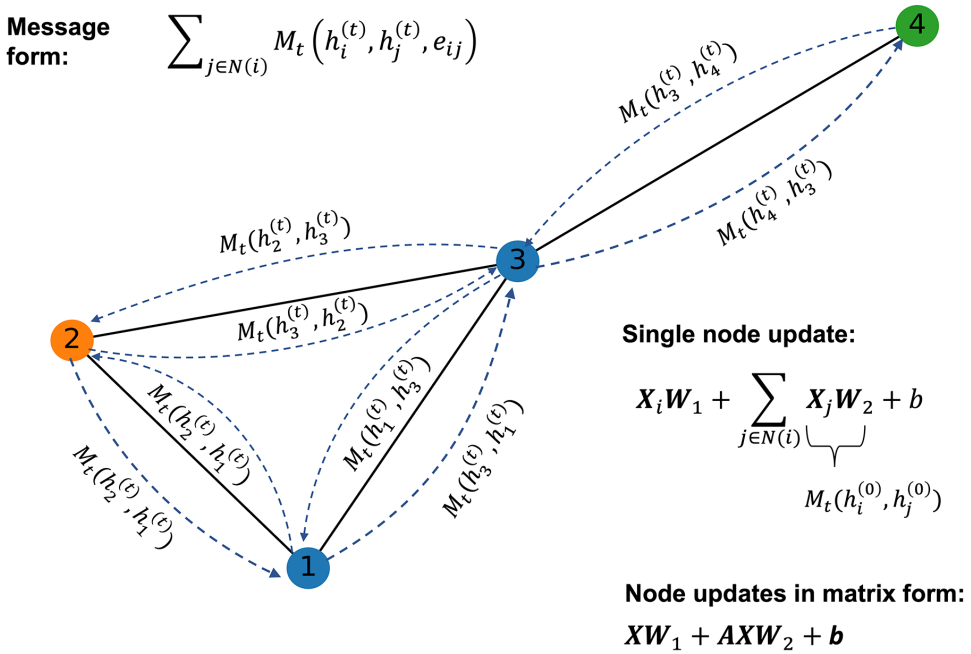

Figure 18.8: The convolutions implemented on the graph and the message form

In the next section, we'll incorporate this graph convolution layer into a GNN model implemented in PyTorch.

Implementing a GNN in PyTorch from scratch

The previous section focused on understanding and implementing a graph convolution operation. In this section, we'll walk you through a basic implementation of a graph neural network to illustrate how to apply these methods to graphs if you start from scratch. If this approach appears complicated, don't worry; GNNs are relatively complex models to implement. Thus, we'll introduce PyTorch Geometric in a later section, which provides tools to ease the implementation of, and the data management for, graph neural networks.

Defining the NodeNetwork model

We will start this section by showing a PyTorch from-scratch implementation of a GNN. We will take a top-down approach, starting with the main neural network model, which we call `NodeNetwork`, and then we will fill in the individual details:

```
import networkx as nx
import torch
from torch.nn.parameter import Parameter
import numpy as np
import math
import torch.nn.functional as F

class NodeNetwork(torch.nn.Module):
    def __init__(self, input_features):
        super().__init__()
        self.conv_1 = BasicGraphConvolutionLayer (
            input_features, 32)
        self.conv_2 = BasicGraphConvolutionLayer(32, 32)
        self.fc_1 = torch.nn.Linear(32, 16)
        self.out_layer = torch.nn.Linear(16, 2)

    def forward(self, X, A, batch_mat):
        x = F.relu(self.conv_1(X, A))
        x = F.relu(self.conv_2(x, A))
        output = global_sum_pool(x, batch_mat)
        output = self.fc_1(output)
        output = self.out_layer(output)
        return F.softmax(output, dim=1)
```

The `NodeNetwork` model we just defined can be summarized as follows:

1. Perform two graph convolutions (`self.conv_1` and `self.conv_2`)
2. Pool all the node embeddings via `global_sum_pool`, which we will define later
3. Run the pooled embeddings through two fully connected layers (`self.fc_1` and `self.out_layer`)
4. Output a class-membership probability via softmax

The structure of the network along with a visualization of what each layer is doing is summarized in *Figure 18.9*:

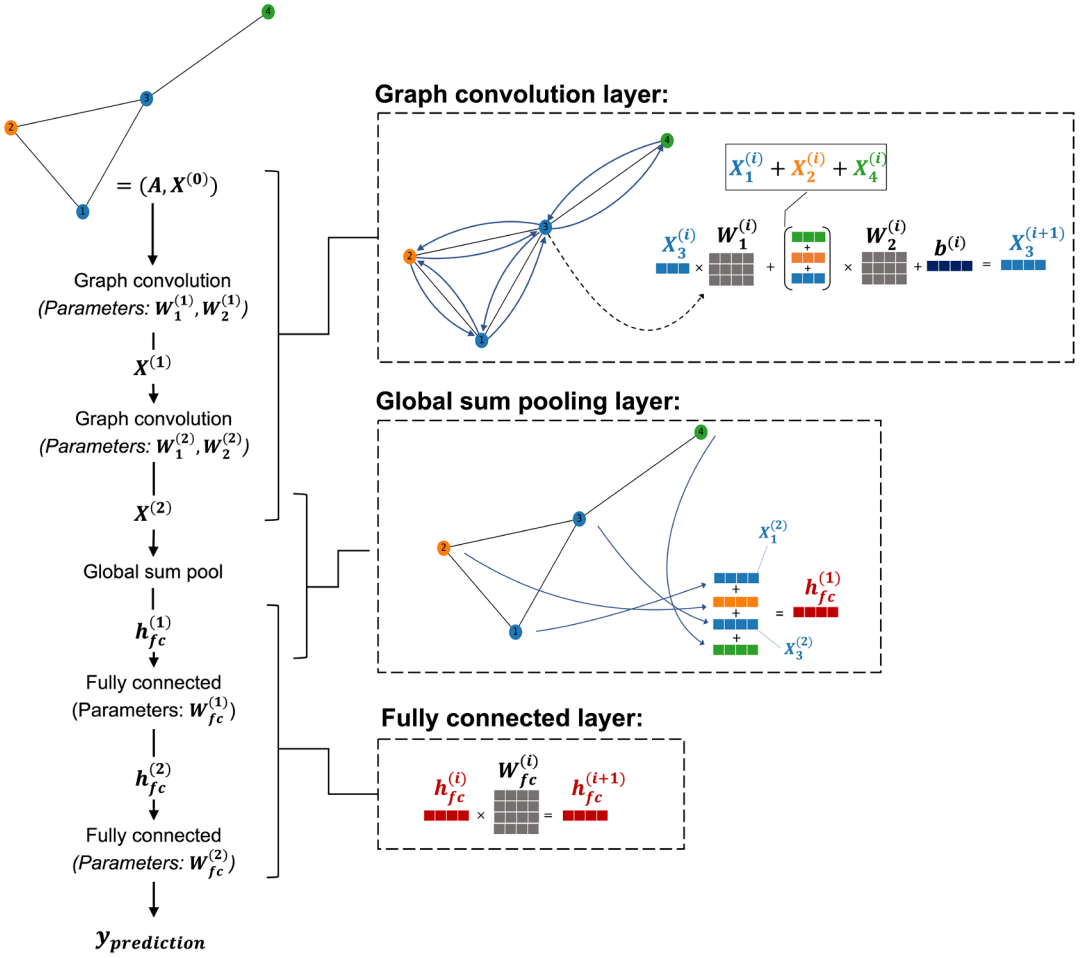

Figure 18.9: A visualization of each neural network layer

The individual aspects, such as the graph convolution layers and global pooling, will be discussed in the next subsections.

Coding the NodeNetwork's graph convolution layer

Now, let's define the graph convolution operation (`BasicGraphConvolutionLayer`) that was used inside the previous `NodeNetwork` class:

```
class BasicGraphConvolutionLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
```

```
        self.out_channels = out_channels
        self.W2 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))
        self.W1 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))

        self.bias = Parameter(torch.zeros(
                out_channels, dtype=torch.float32))
    def forward(self, X, A):
        potential_msgs = torch.mm(X, self.W2)
        propagated_msgs = torch.mm(A, potential_msgs)
        root_update = torch.mm(X, self.W1)
        output = propagated_msgs + root_update + self.bias
        return output
```

As with fully connected layers and image convolutional layers, we add a bias term so that the intercept of the linear combination of the layer outputs (prior to the application of a nonlinearity like ReLU) can vary. The `forward()` method implements the matrix form of the forward pass, which we discussed in the previous subsection, with the addition of a bias term.

To try out the `BasicGraphConvolutionLayer`, let's apply it to the graph and adjacency matrix that we defined in the section *Implementing a basic graph convolution* previously:

```
>>> print('X.shape:', X.shape)
X.shape: (4, 3)

>>> print('A.shape:', A.shape)
A.shape: (4, 4)

>>> basiclayer = BasicGraphConvolutionLayer(3, 8)
>>> out = basiclayer(
...     X=torch.tensor(X, dtype=torch.float32),
...     A=torch.tensor(A, dtype=torch.float32)
... )

>>> print('Output shape:', out.shape)
Output shape: torch.Size([4, 8])
```

Based on the code example above, we can see that our `BasicGraphConvolutionLayer` converted the four-node graph consisting of three features into a representation with eight features.

Adding a global pooling layer to deal with varying graph sizes

Next, we define the `global_sum_pool()` function that was used in the `NodeNetwork` class, where `global_sum_pool()` implements a global pooling layer. Global pooling layers aggregate all of a graph's node embeddings into a fixed-sized output. As shown in *Figure 18.9*, `global_sum_pool()` sums all the node embeddings of a graph. We note that this global pooling is relatively similar to the global average pooling used in CNNs, which is used before the data is run through fully connected layers, as we have seen in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*.

Summing all the node embeddings results in a loss of information, so reshaping the data would be preferable, but since graphs can have different sizes, this is not feasible. Global pooling can be done with any permutation invariant function, for example, `sum`, `max`, and `mean`. Here is the implementation of `global_sum_pool()`:

```
def global_sum_pool(X, batch_mat):
    if batch_mat is None or batch_mat.dim() == 1:
        return torch.sum(X, dim=0).unsqueeze(0)
    else:
        return torch.mm(batch_mat, X)
```

If data is not batched or the batch size is one, this function just sums over the current node embeddings. Otherwise, the embeddings are multiplied with `batch_mat`, which has a structure based on how graph data is batched.

When all data in a dataset has the same dimensionality, batching the data is as straightforward as adding a dimension by stacking the data. (Side note: the function called in the default batching function in PyTorch is literally called `stack`.) Since graph sizes vary, this approach is not feasible with graph data unless padding is used. However, padding can be inefficient in cases where graph sizes can vary substantially. Usually, the better way to deal with varying graph sizes is to treat each batch as a single graph where each graph in the batch is a subgraph that is disconnected from the rest. This is illustrated in *Figure 18.10*:

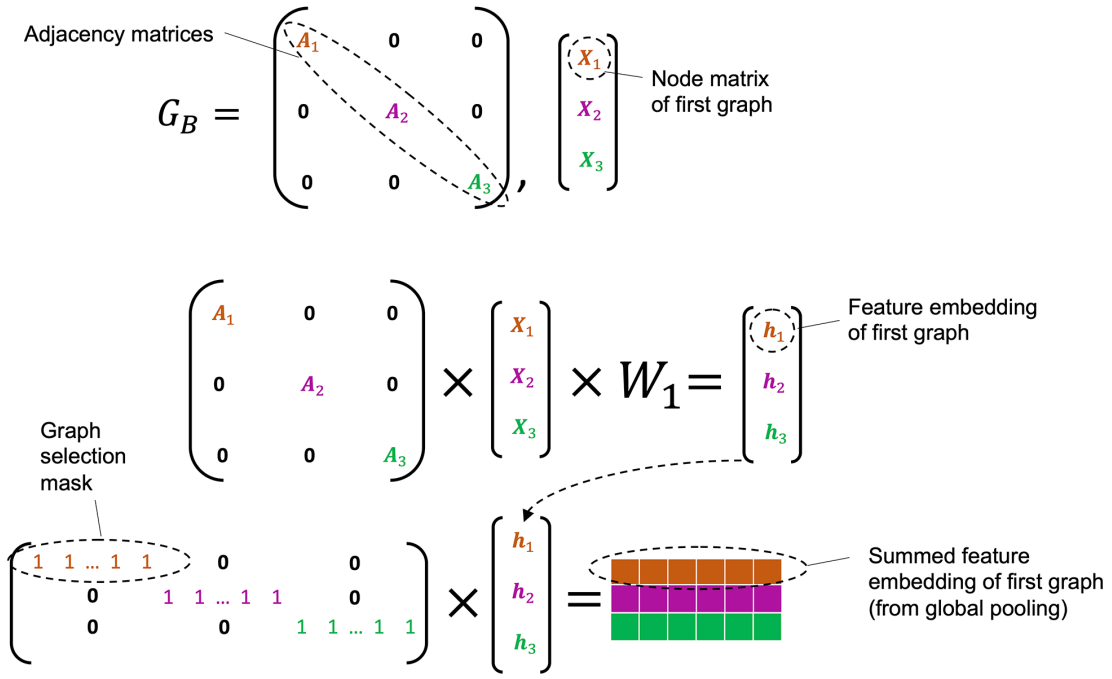

Figure 18.10: How to deal with varying graph sizes

To describe *Figure 18.10* more formally, suppose we are given graphs G_1, ..., G_k of sizes n_1, ..., n_k with f features per node. In addition, we are given the corresponding adjacency matrices $\boldsymbol{A}_1$, ..., $\boldsymbol{A}_k$ and feature matrices $\boldsymbol{X}_1$, ..., $\boldsymbol{X}_k$. Let N be the total number of nodes, $N = \sum_{i=1}^{k} n_i$, $s_1 = 0$, and $s_i = s_{i-1} + n_{i-1}$ for $1 < i \leq k$. As shown in the figure, we define a graph G_B with $N \times N$ adjacency matrix $\boldsymbol{A}_B$ and $N \times f$ feature matrix $\boldsymbol{X}_B$. Using Python index notation, $\boldsymbol{A}_B[s_i{:}s_i + n_i, s_i + n_i] = \boldsymbol{A}_i$, and all other elements of $\boldsymbol{A}_B$ outside these index sets are 0. Additionally, $\boldsymbol{X}_B[s_i{:}s_i + n_i, :] = \boldsymbol{X}_i$.

By design, disconnected nodes will never be in the same receptive field of a graph convolution. As a result, when backpropagating gradients of G_B through graph convolutions, the gradients attached to each graph in the batch will be independent. This means that if we treat a set of graph convolutions as a function f, if $h_B = f(X_B, A_B)$ and $h_i = f(X_i, A_i)$, then $h_B[s_i{:}s_i + n, :] = h_i$. If the sum global pooling extracts the sums of each h_i from h_B as separate vectors, passing that stack of vectors through fully connected layers would keep the gradients of each item in the batch separate throughout the entire backpropagation.

This is the purpose of batch_mat in global_sum_pool()—to serve as a graph selection mask that keeps the graphs in the batch separate. We can generate this mask for graphs of sizes n_1, ..., n_k with the following code:

```
def get_batch_tensor(graph_sizes):
    starts = [sum(graph_sizes[:idx])
              for idx in range(len(graph_sizes))]
    stops = [starts[idx] + graph_sizes[idx]
             for idx in range(len(graph_sizes))]
    tot_len = sum(graph_sizes)
    batch_size = len(graph_sizes)
    batch_mat = torch.zeros([batch_size, tot_len]).float()
    for idx, starts_and_stops in enumerate(zip(starts, stops)):
        start = starts_and_stops[0]
        stop = starts_and_stops[1]
        batch_mat[idx,start:stop] = 1
    return batch_mat
```

Thus, given a batch size, b, batch_mat is a $b \times N$ matrix where batch_mat[i–1, s_i:$s_i + n_i$] = 1 for $1 \leq i \leq k$ and where elements outside these index sets are 0. The following is a collate function for constructing a representation of some G_B and a corresponding batch matrix:

```
# batch is a list of dictionaries each containing
# the representation and label of a graph
def collate_graphs(batch):
    adj_mats = [graph['A'] for graph in batch]
    sizes = [A.size(0) for A in adj_mats]
    tot_size = sum(sizes)
    # create batch matrix
    batch_mat = get_batch_tensor(sizes)
    # combine feature matrices
    feat_mats = torch.cat([graph['X'] for graph in batch], dim=0)
    # combine labels
    labels = torch.cat([graph['y'] for graph in batch], dim=0)
    # combine adjacency matrices
    batch_adj = torch.zeros([tot_size, tot_size], dtype=torch.float32)
    accum = 0
    for adj in adj_mats:
        g_size = adj.shape[0]
        batch_adj[accum:accum+g_size,accum:accum+g_size] = adj
        accum = accum + g_size
    repr_and_label = {'A': batch_adj,
```

```
            'X': feat_mats, 'y': labels,
            'batch': batch_mat}
    return repr_and_label
```

Preparing the DataLoader

In this section, we will see how the code from the previous subsections all comes together. First, we will generate some graphs and put them into a PyTorch `Dataset`. Then, we will use our `collate` function in a `DataLoader` for our GNN.

But before we define the graphs, let's implement a function that builds a dictionary representation that we will use later:

```
def get_graph_dict(G, mapping_dict):
    # Function builds dictionary representation of graph G
    A = torch.from_numpy(
        np.asarray(nx.adjacency_matrix(G).todense())).float()
    # build_graph_color_label_representation()
    # was introduced with the first example graph
    X = torch.from_numpy(
      build_graph_color_label_representation(
               G, mapping_dict)).float()
    # kludge since there is not specific task for this example
    y = torch.tensor([[1,0]]).float()
    return {'A': A, 'X': X, 'y': y, 'batch': None}
```

This function takes a NetworkX graph and returns a dictionary containing its adjacency matrix `A`, its node feature matrix `X`, and a binary label `y`. Since we won't actually be training this model on a real-world task, we just set the labels arbitrarily. Then, `nx.adjacency_matrix()` takes a NetworkX graph and returns a sparse representation that we convert to a dense `np.array` form using `todense()`.

We'll now construct graphs and use the `get_graph_dict` function to convert NetworkX graphs to a format our network can handle:

```
>>> # building 4 graphs to treat as a dataset
>>> blue, orange, green = "#1f77b4", "#ff7f0e","#2ca02c"
>>> mapping_dict= {green:0, blue:1, orange:2}
>>> G1 = nx.Graph()
>>> G1.add_nodes_from([
...     (1,{"color": blue}),
...     (2,{"color": orange}),
...     (3,{"color": blue}),
...     (4,{"color": green})
... ])
```

```
>>> G1.add_edges_from([(1, 2), (2, 3), (1, 3), (3, 4)])
>>> G2 = nx.Graph()
>>> G2.add_nodes_from([
...     (1,{"color": green}),
...     (2,{"color": green}),
...     (3,{"color": orange}),
...     (4,{"color": orange}),
...     (5,{"color": blue})
... ])
>>> G2.add_edges_from([(2, 3),(3, 4),(3, 1),(5, 1)])
>>> G3 = nx.Graph()
>>> G3.add_nodes_from([
...     (1,{"color": orange}),
...     (2,{"color": orange}),
...     (3,{"color": green}),
...     (4,{"color": green}),
...     (5,{"color": blue}),
...     (6,{"color":orange})
... ])
>>> G3.add_edges_from([(2,3), (3,4), (3,1), (5,1), (2,5), (6,1)])
>>> G4 = nx.Graph()
>>> G4.add_nodes_from([
...     (1,{"color": blue}),
...     (2,{"color": blue}),
...     (3,{"color": green})
... ])
>>> G4.add_edges_from([(1, 2), (2, 3)])
>>> graph_list = [get_graph_dict(graph, mapping_dict) for graph in
...     [G1, G2, G3, G4]]
```

The graphs this code generates are visualized in *Figure 18.11*:

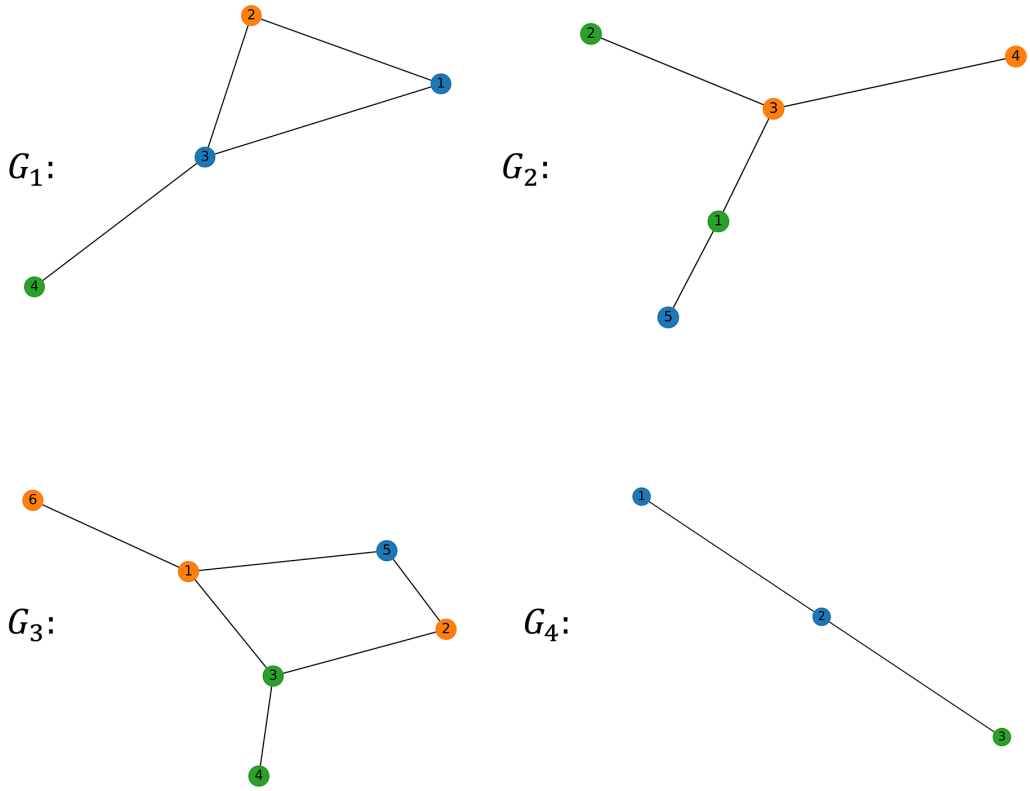

Figure 18.11: Four generated graphs

This code block constructs four NetworkX graphs and stores them in a list. Here, the constructor of `nx.Graph()` initializes an empty graph, and `add_nodes_from()` adds nodes to the empty graph from a list of tuples. The first item in each tuple is the node's name, and the second item is a dictionary of that node's attributes.

The `add_edges_from()` method of a graph takes a list of tuples where each tuple defines an edge between its elements (nodes). Now, we can construct a PyTorch `Dataset` for these graphs:

```
from torch.utils.data import Dataset
class ExampleDataset(Dataset):
    # Simple PyTorch dataset that will use our list of graphs
    def __init__(self, graph_list):
        self.graphs = graph_list
    def __len__(self):
        return len(self.graphs)

    def __getitem__(self,idx):
        mol_rep = self.graphs[idx]
        return mol_rep
```

While using a custom `Dataset` may seem like unnecessary effort, it allows us to exhibit how `collate_graphs()` can be used in a `DataLoader`:

```
>>> from torch.utils.data import DataLoader
>>> dset = ExampleDataset(graph_list)
>>> # Note how we use our custom collate function
>>> loader = DataLoader(
...     dset, batch_size=2, shuffle=False,
...     collate_fn=collate_graphs)
```

Using the NodeNetwork to make predictions

After we have defined all the necessary functions and set up the `DataLoader`, we now initialize a new `NodeNetwork` and apply it to our graph data:

```
>>> node_features = 3
>>> net = NodeNetwork(node_features)

>>> batch_results = []
>>> for b in loader:
...     batch_results.append(
...         net(b['X'], b['A'], b['batch']).detach())
```

Note that for brevity, we didn't include a training loop; however, the GNN model could be trained in a regular fashion by computing the loss between predicted and true class labels, backpropagating the loss via `.backward()`, and updating the model weights via a gradient descent-based optimizer. We leave this as an optional exercise for the reader. In the next section, we will show how to do that with a GNN implementation from PyTorch Geometric, which implements more sophisticated GNN code.

To continue with our previous code, let's now provide a single input graph to the model directly without the `DataLoader`:

```
>>> G1_rep = dset[1]
>>> G1_single = net(
...     G1_rep['X'], G1_rep['A'], G1_rep['batch']).detach()
```

We can now compare the results from applying the GNN to a single graph (`G1_single`) and to the first graph from the `DataLoader` (also the first graph, `G1`, which we guaranteed, since we set `shuffle=False`) to double-check that the batch loader works correctly. As we can see by using `torch.isclose()` (to account for rounding errors), the results are equivalent, as we would have hoped:

```
>>> G1_batch = batch_results[0][1]
>>> torch.all(torch.isclose(G1_single, G1_batch))
tensor(True)
```

Congrats! You now understand how to construct, set up, and run a basic GNN. However, from this introduction, you probably realize that managing and manipulating graph data can be somewhat laborious. Also, we didn't even build a graph convolution that uses edge labels, which would complicate matters further. Thankfully, there is PyTorch Geometric, a package that makes this much easier by providing implementations of many GNN layers. We'll introduce this library with an end-to-end example of implementing and training a more complex GNN on molecule data in the next subsection.

Implementing a GNN using the PyTorch Geometric library

In this section, we will implement a GNN using the PyTorch Geometric library, which simplifies the process of training GNNs. We apply the GNN to QM9, a dataset consisting of small molecules, to predict isotropic polarizability, which is a measure of a molecule's tendency to have its charge distorted by an electric field.

Installing PyTorch Geometric

PyTorch Geometric can be installed via conda or pip. We recommend you visit the official documentation website at `https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html` to select the installation command recommended for your operating system. For this chapter, we used pip to install version 2.0.2 along with its `torch-scatter` and `torch-sparse` dependencies:

```
pip install torch-scatter==2.0.9
pip install torch-sparse==0.6.12
pip install torch-geometric==2.0.2
```

Let's start by loading a dataset of small molecules and look at how PyTorch Geometric stores the data:

```
>>> # For all examples in this section we use the following imports.
>>> # Note that we are using torch_geometric's DataLoader.
>>> import torch
>>> from torch_geometric.datasets import QM9
>>> from torch_geometric.loader import DataLoader
>>> from torch_geometric.nn import NNConv, global_add_pool
>>> import torch.nn.functional as F
>>> import torch.nn as nn
>>> import numpy as np
>>> # let's load the QM9 small molecule dataset
>>> dset = QM9('.')
>>> len(dset)
130831
>>> # Here's how torch geometric wraps data
>>> data = dset[0]
>>> data
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", pos=[5, 3],
x=[5, 11], y=[1, 19], z=[5])
>>> # can access attributes directly
>>> data.z
tensor([6, 1, 1, 1, 1])
>>> # the atomic number of each atom can add attributes
>>> data.new_attribute = torch.tensor([1, 2, 3])
>>> data
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", new_
attribute=[3], pos=[5, 3], x=[5, 11], y=[1, 19], z=[5])
>>> # can move all attributes between devices
>>> device = torch.device(
...     "cuda:0" if torch.cuda.is_available() else "cpu"
... )
>>> data.to(device)
>>> data.new_attribute.is_cuda
True
```

The `Data` object is a convenient, flexible wrapper for graph data. Note that many PyTorch Geometric objects require certain keywords in data objects to process them correctly. Specifically, `x` should contain node features, `edge_attr` should contain edge features, `edge_index` should include an edge list, and `y` should contain labels. The QM9 data contains some additional attributes of note: `pos`, the position of each of the molecules' atoms in a 3D grid, and `z`, the atomic number of each atom in the molecule. The labels in the QM9 are a bunch of physical properties of the molecules, such as dipole moment, free energy, enthalpy, or isotropic polarization. We are going to implement a GNN and train it on QM9 to predict isotropic polarization.

The QM9 dataset

The QM9 dataset contains 133,885 small organic molecules labeled with several geometric, energetic, electronic, and thermodynamic properties. QM9 is a common benchmark dataset for developing methods for predicting chemical structure-property relationships and hybrid quantum mechanic/machine learning methods. More information about the dataset can be found at `http://quantum-machine.org/datasets/`.

The bond types of molecules are important; that is, which atoms are connected via a certain bond type, for example, single or double bonds, matters. Hence, we'll want to use a graph convolution that can utilize edge features. For this, we'll use the `torch_geometric.nn.NNConv` layer. (If you are interested in the implementation details, its source code be found at `https://pytorch-geometric.readthedocs.io/en/latest/_modules/torch_geometric/nn/conv/nn_conv.html#NNConv`.)

This convolution in the `NNConv` layer takes the following form:

$$\boldsymbol{X}_i^{(t)} = \boldsymbol{W}\boldsymbol{X}_i^{(t-1)} + \sum_{j \in N(i)} \boldsymbol{X}_j^{(t-1)} . h_\Theta(e_{i,j})$$

Here, h is a neural network parameterized by a set of weights Θ, and W is a weight matrix for the node labels. This graph convolution is very similar to the one we implemented previously from scratch:

$$\boldsymbol{X}_i^{(t)} = \boldsymbol{W}_1\boldsymbol{X}_i^{(t-1)} + \sum_{j \in N(i)} \boldsymbol{X}_j^{(t-1)}\boldsymbol{W}_2$$

The only real difference is that the W_2 equivalent, the neural network h, is parametrized based on the edge labels, which allows the weights to vary for different edge labels. Via the following code, we implement a GNN utilizing two such graph convolutional layers (`NNConv`):

```
class ExampleNet(torch.nn.Module):
    def __init__(self, num_node_features, num_edge_features):
        super().__init__()
        conv1_net = nn.Sequential(
            nn.Linear(num_edge_features, 32),
            nn.ReLU(),
            nn.Linear(32, num_node_features*32))

        conv2_net = nn.Sequential(
            nn.Linear(num_edge_features, 32),
            nn.ReLU(),
            nn.Linear(32, 32*16))

        self.conv1 = NNConv(num_node_features, 32, conv1_net)
        self.conv2 = NNConv(32,16, conv2_net)
```

```
        self.fc_1 = nn.Linear(16, 32)
        self.out = nn.Linear(32, 1)

    def forward(self, data):
        batch, x, edge_index, edge_attr = (
            data.batch, data.x, data.edge_index, data.edge_attr)
        # First graph conv layer
        x = F.relu(self.conv1(x, edge_index, edge_attr))
        # Second graph conv layer
        x = F.relu(self.conv2(x, edge_index, edge_attr))
        x = global_add_pool(x,batch)
        x = F.relu(self.fc_1(x))
        output = self.out(x)
        return output
```

We'll train this GNN to predict a molecule's isotropic polarizability, a measure of the relative tendency of a molecule's charge distribution to be distorted by an external electric field. We'll split the QM9 dataset into training, validation, and test sets, and use PyTorch Geometric `DataLoader`. Note that these do not require a special collate function, but require a `Data` object with appropriately named attributes.

Next, let's split the dataset:

```
>>> from torch.utils.data import random_split
>>> train_set, valid_set, test_set = random_split(
...     dset,[110000, 10831, 10000])
>>> trainloader = DataLoader(train_set, batch_size=32, shuffle=True)
>>> validloader = DataLoader(valid_set, batch_size=32, shuffle=True)
>>> testloader = DataLoader(test_set, batch_size=32, shuffle=True)
```

The following code will initialize and train a network on a GPU (if available):

```
>>> # initialize a network
>>> qm9_node_feats, qm9_edge_feats = 11, 4
>>> net = ExampleNet(qm9_node_feats, qm9_edge_feats)

>>> # initialize an optimizer with some reasonable parameters
>>> optimizer = torch.optim.Adam(
...     net.parameters(), lr=0.01)
>>> epochs = 4
>>> target_idx = 1 # index position of the polarizability label
>>> device = torch.device("cuda:0" if
...                       torch.cuda.is_available() else "cpu")
>>> net.to(device)
```

The training loop, shown in the following code, follows the familiar pattern we have encountered in previous PyTorch chapters, so we can skip the explanation details. However, one detail that is worth highlighting is that here we are computing the mean squared error (MSE) loss instead of the cross-entropy, since polarizability is a continuous target and not a class label:

```
>>> for total_epochs in range(epochs):
...     epoch_loss = 0
...     total_graphs = 0
...     net.train()
...     for batch in trainloader:
...         batch.to(device)
...         optimizer.zero_grad()
...         output = net(batch)
...         loss = F.mse_loss(
...             output,batch.y[:, target_idx].unsqueeze(1))
...         loss.backward()
...         epoch_loss += loss.item()
...         total_graphs += batch.num_graphs
...         optimizer.step()
...     train_avg_loss = epoch_loss / total_graphs
...     val_loss = 0
...     total_graphs = 0
...     net.eval()
...     for batch in validloader:
...         batch.to(device)
...         output = net(batch)
...         loss = F.mse_loss(
...             output,batch.y[:, target_idx].unsqueeze(1))
...         val_loss += loss.item()
...         total_graphs += batch.num_graphs
...     val_avg_loss = val_loss / total_graphs
...     print(f"Epochs: {total_epochs} | "
...           f"epoch avg. loss: {train_avg_loss:.2f} | "
...           f"validation avg. loss: {val_avg_loss:.2f}")
Epochs: 0 | epoch avg. loss: 0.30 | validation avg. loss: 0.10
Epochs: 1 | epoch avg. loss: 0.12 | validation avg. loss: 0.07
Epochs: 2 | epoch avg. loss: 0.10 | validation avg. loss: 0.05
Epochs: 3 | epoch avg. loss: 0.09 | validation avg. loss: 0.07
```

Over the first four training epochs, both training and validation loss are decreasing. The dataset is large and may take a little while to train on a CPU, so we stop training after four epochs. However, if we train the model further, the loss will continue to improve. You can train the model for additional epochs to see how that changes the performance.

The following code predicts the values on the test data and collects the true labels:

```
>>> net.eval()
>>> predictions = []
>>> real = []
>>> for batch in testloader:
...     output = net(batch.to(device))
...     predictions.append(output.detach().cpu().numpy())
...     real.append(
...             batch.y[:,target_idx] .detach().cpu().numpy())
>>> real = np.concatenate(real)
>>> predictions = np.concatenate(predictions)
```

Now we can make a scatterplot with a subset of the test data. Since the test dataset is relatively large (10,000 molecules), the results can be a bit cluttered, and for simplicity, we only plot the first 500 predictions and targets:

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(real[:500], predictions[:500])
>>> plt.xlabel('Isotropic polarizability')
>>> plt.ylabel('Predicted isotropic polarizability')
```

The resulting figure is shown here:

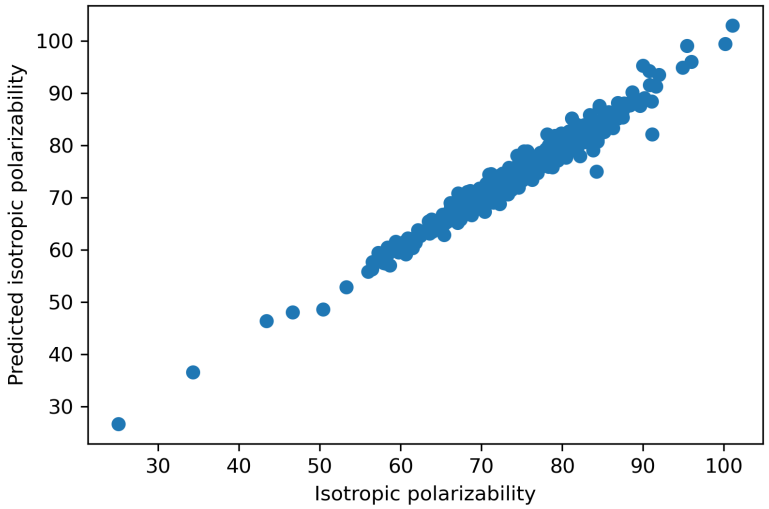

Figure 18.12: Predicted isotropic polarizability plotted against the actual isotropic polarizability

Based on the plot, given that the points lie relatively near the diagonal, our simple GNN appears to have done a decent job with predicting isotropic polarization values, even without hyperparameter tuning.

TorchDrug – A PyTorch-based library for drug discovery

PyTorch Geometric is a comprehensive general-purpose library for working with graphs, including molecules, as you have seen in this section. If you are interested in more in-depth molecule work and drug discovery, we also recommend considering the recently developed TorchDrug library, which offers many convenient utilities for working with molecules. You can find out more about TorchDrug here: `https://torchdrug.ai/`.

Other GNN layers and recent developments

This section will introduce a selection of additional layers that you can utilize in your GNNs, in addition to providing a high-level overview of some recent developments in the field. While we will provide background on the intuition behind these layers and their implementations, these concepts can become a little complicated mathematically speaking, but don't get discouraged. These are optional topics, and it is not necessary to grasp the minutiae of all these implementations. Understanding the general ideas behind the layers will be sufficient to experiment with the PyTorch Geometric implementations that we reference.

The following subsections will introduce spectral graph convolution layers, graph pooling layers, and normalization layers for graphs. Lastly, the final subsection will provide a bird's eye view of some more advanced kinds of graph neural networks.

Spectral graph convolutions

The graph convolutions we have utilized up to this point have all been spatial in nature. This means that they aggregate information based on the topological space associated with the graph, which is just a fancy way of saying that spatial convolutions operate on local neighborhoods of nodes. As a consequence of this, if a GNN that utilizes spatial convolutions needs to capture complex global patterns in graph data, then the network will need to stack multiple spatial convolutions. In situations where these global patterns are important, but network depth needs to be limited, spectral graph convolutions are an alternative kind of convolution to consider.

Spectral graph convolutions operate differently than spatial graph convolutions. Spectral graph convolutions operate by utilizing the graph's spectrum—its set of eigenvalues—by computing the eigendecomposition of a normalized version of the graph's adjacency matrix called the *graph Laplacian*. That last sentence may seem like a doozy, so let's break it down and go over it step by step.

For an undirected graph, the Laplacian matrix of a graph is defined as $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, where $\boldsymbol{A}$ is the adjacency matrix of the graph and $\boldsymbol{D}$ is the degree matrix. A degree matrix is a diagonal matrix where the element on the diagonal in the row with index i is the number of edges in and out of the node associated with the ith row of the adjacency matrix.

$\boldsymbol{L}$ is a real-valued symmetric matrix, and it has been proven that real-valued symmetric matrices can be decomposed as $\boldsymbol{L} = \boldsymbol{Q\Lambda Q^T}$, where Q is an orthogonal matrix whose columns are the eigenvectors of $\boldsymbol{L}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix whose elements are the eigenvalues of $\boldsymbol{L}$. You can think of Q as providing an underlying representation of the graph's structure. Unlike spatial convolutions, which use local neighborhoods of the graph that are defined by $\boldsymbol{A}$, spectral convolutions utilize the alternative representation of the structure from Q to update the node embeddings.

The following example of a spectral convolution utilizes the eigendecomposition of the *symmetric normalized graph Laplacian*, which is defined for a graph as follows:

$$\boldsymbol{L}_{sym} = \boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$$

Here, $\boldsymbol{I}$ is the identity matrix. This is used because the normalization of the graph Laplacian can help stabilize the gradient-based training procedure similar to feature standardization.

Given that $\boldsymbol{Q\Lambda Q^T}$ is the eigendecomposition of $\boldsymbol{L}_{sym}$, the graph convolution is defined as follows:

$$\boldsymbol{X}' = \boldsymbol{Q}(\boldsymbol{Q}^T\boldsymbol{X} \odot \boldsymbol{Q}^T\boldsymbol{W})$$

Here, $\boldsymbol{W}$ is a trainable weight matrix. The inside of the parentheses essentially multiplies $\boldsymbol{X}$ and $\boldsymbol{W}$ by a matrix that encodes structural relationships in the graph. The $\odot$ operator here denotes element-wise multiplication of the inner terms, while the outside Q maps the result back into the original basis. This convolution has a few undesirable properties, since computing a graph's eigendecomposition has a computational complexity of $O(n^3)$. This means that it is slow, and as it is structured, $\boldsymbol{W}$ is dependent on the size of the graph. Consequently, the spectral convolution can only be applied to graphs of the same size. Furthermore, the receptive field of this convolution is the whole graph, and this cannot be tuned in the current formulation. However, various techniques and convolutions have been developed to address these issues.

For example, Bruna and colleagues (`https://arxiv.org/abs/1312.6203`) introduced a smoothing method that addresses the size dependence of $\boldsymbol{W}$ by approximating it with a set of functions, each multiplied by their own scalar parameter, α. That is, given the set of functions f_1, ..., f_n, $\boldsymbol{W} \approx \sum \alpha_i f_i$. The set of functions is such that the dimensionality can be varied. However, since α remains scalar, the convolutions parameter space can be independent of the graph size.

Other spectral convolutions worth mentioning include the Chebyshev graph convolution (`https://arxiv.org/abs/1606.09375`), which can approximate the original spectral convolution at a lower time complexity and can have receptive fields with varying sizes. Kipf and Welling (`https://arxiv.org/abs/1609.02907`) introduce a convolution with properties similar to the Chebyshev convolutions, but with a reduced parameter burden. Implementations of both of these are available in PyTorch Geometric as `torch_geometric.nn.ChebConv` and `torch_geometric.nn.GCNConv` and are reasonable places to start if you want to play around with spectral convolutions.

Pooling

We will briefly discuss some examples of pooling layers that have been developed for graphs. While the downsampling provided by pooling layers has been beneficial in CNN architectures, the benefit of downsampling in GNNs has not been realized as clearly.

Pooling layers for image data (ab)use spatial locality, which graphs do not have. If a clustering of the nodes in a graph is provided, we can define how a graph pooling layer should pool nodes. However, it is unclear how to define optimal clustering, and different clustering approaches may be favored for different contexts. Even after clustering is determined, if nodes are downsampled, it is unclear how the remaining nodes should be connected. While these are still open research questions, we'll look at a few graph pooling layers and point out their approaches to the aforementioned issues.

As with CNNs, there are mean and max pooling layers that can be applied to GNNs. As shown in *Figure 18.13*, given a clustering of nodes, each cluster becomes a node in a new graph:

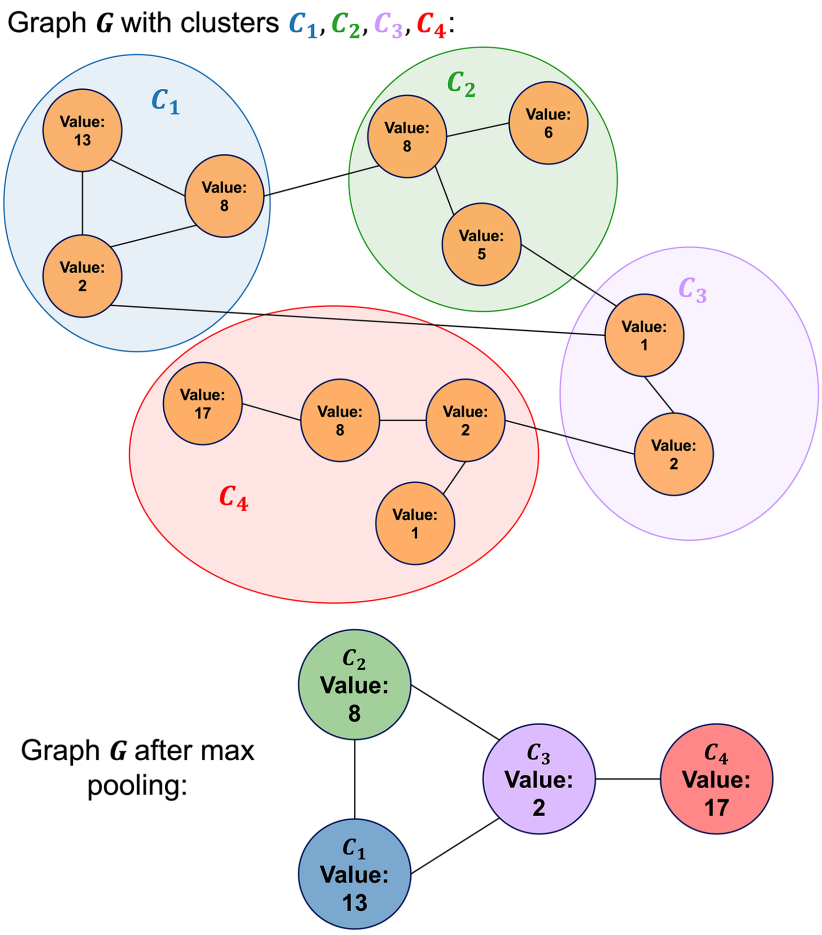

Figure 18.13: Applying max pooling to a graph

Each cluster's embedding is equal to the mean or max of the embeddings of the nodes in the cluster. To address connectivity, the cluster is assigned the union of all edge indices in the cluster. For example, if nodes i, j, k are assigned to cluster c_1, any node, or cluster containing a node, that shared an edge with i, j, or k will share an edge with c_1.

A more complex pooling layer, *DiffPool* (`https://arxiv.org/abs/1806.08804`), tries to address both clustering and downsampling simultaneously. This layer learns a soft cluster assignment matrix $\boldsymbol{S} \in \mathbb{R}^{n \times c}$, which distributes n node embeddings into c clusters. (For a refresher on soft versus hard clustering, refer to the section *Hard versus soft clustering* in *Chapter 10*, *Working with Unlabeled Data – Clustering Analysis*.) With this, $\boldsymbol{X}$ is updated as $\boldsymbol{X}' = \boldsymbol{S}^T\boldsymbol{X}$ and $\boldsymbol{A}$ as $\boldsymbol{A}' = \boldsymbol{S}^T\boldsymbol{A}^T\boldsymbol{S}$. Notably, $\boldsymbol{A}'$ no longer contains discrete values and can instead be viewed as a matrix of edge weightings. Over time, *DiffPool* converges to an almost hard clustering assignment with interpretable structure.

Another pooling method, top-k pooling, drops nodes from the graph instead of aggregating them, which circumvents clustering and connectivity issues. While this seemingly comes with a loss of the information in the dropped nodes, in the context of a network, as long as a convolution occurs before pooling, the network can learn to avoid this. The dropped nodes are selected using a projection score against a learnable vector p. The actual formulation to compute $(\boldsymbol{X}', \boldsymbol{A}')$, as stated in *Towards Sparse Hierarchical Graph Classifiers* (`https://arxiv.org/abs/1811.01287`), is:

$$y = \frac{X_p}{\|p\|}, \quad i = \text{top-k}(\boldsymbol{y}, k), \quad \boldsymbol{X}' = (\boldsymbol{X} \odot \tanh(\boldsymbol{y}))_i, \quad \boldsymbol{A}' = A_{ii}$$

Here, top-k selects the indexes of $\boldsymbol{y}$, with the top k values and the index vector i being used to drop rows of $\boldsymbol{X}$ and $\boldsymbol{A}$. Top-k pooling is implemented in PyTorch Geometric as `torch_geometric.nn.TopKPooling`. Additionally, max and mean pooling are implemented as `torch_geometric.nn.max_pool_x` and `torch_geometric.nn.avg_pool_x`, respectively.

Normalization

Normalization techniques are utilized in many kinds of neural networks to help stabilize and/or speed up the training process. Many approaches, such as batch normalization (discussed in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*), can be readily applied in GNNs with appropriate bookkeeping. In this section, we will briefly describe some of the normalization layers that have been designed specifically for graph data.

As a quick review of normalization, we mean that given a set of feature values x_1, ..., x_n, we update the values with $\frac{x_i - \mu}{\sigma}$, where μ is the mean and σ the standard deviation of the set of values. Typically, most neural network normalization methods take the general form $\gamma \frac{x_i - \mu}{\sigma} + \beta$, where γ and β are learnable parameters, and the difference between methods has to do with the set of features the normalization is applied over.

GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training by *Tianle Cai* and colleagues, 2020 (`https://arxiv.org/abs/2009.03294`), showed that the mean statistic after aggregation in a graph convolution can contain meaningful information, so discarding it completely may not be desirable. To address this, they introduced *GraphNorm*.

Borrowing notation from the original manuscript, let h be the matrix of node embeddings. Let $h_{i,j}$ be the jth feature value of node v_i, where $i = 1, ..., n$, and $j = 1, ..., d$. *GraphNorm* takes the following form:

$$\gamma_j \frac{h_{i,j} - \alpha_j \cdot \mu_j}{\hat{\sigma}_j} + \beta_j$$

Here, $\mu_j = \frac{\sum_{i=1}^{n} h_{i,j}}{n}$ and $\hat{\sigma}_j = \frac{\sum_{i=1}^{n} (h_{i,j} - \alpha_j \mu_j)^2}{n}$. The key addition is the learnable parameter, α, which can control how much of the mean statistic, μ_j, to discard.

Another graph normalization technique is *MsgNorm,* which was described by *Guohao Li* and colleagues in the manuscript *DeeperGCN: All You Need to Train Deeper GCNs* in 2020 (`https://arxiv.org/abs/2006.07739`). *MsgNorm* corresponds to the message-passing formulation of graph convolutions mentioned earlier in the chapter. Using message-passing network nomenclature (defined at the end of the subsection *Implementing a basic graph convolution*), after a graph convolution has summed over M_t and produced m_i but before updating the nodes embedding with U_t, *MsgNorm* normalizes m_i with the following formula:

$$m_i' = s \cdot \|h_i\|_2 \cdot \frac{m_i}{\|m_i\|_2}$$

Here, s is a learnable scaling factor and the intuition behind this approach is to normalize the features of the aggregated messages in a graph convolution. While there is no theory to support this normalization approach, it has worked well in practice.

The normalization layers we've discussed are all implemented and available via PyTorch Geometric as `BatchNorm`, `GroupNorm`, and `MessageNorm`. For more information, please visit the PyTorch Geometric documentation at `https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#normalization-layers`.

Unlike graph pooling layers, which may require an additional clustering setup, graph normalization layers can be more readily plugged into an existing GNN model. Testing a variety of normalization methods during model development and optimization is a reasonable and recommended approach.

Pointers to advanced graph neural network literature

The field of deep learning focused on graphs is developing rapidly, and there are many methods that we can't cover in reasonable detail in this introductory chapter. So, before we conclude this chapter, we want to provide interested readers with a selection of pointers to noteworthy literature for more in-depth studies of this topic.

As you might remember from *Chapter 16, Transformers – Improving Natural Language Processing with Attention Mechanisms*, attention mechanisms can improve the capabilities of models by providing additional contexts. In this regard, a variety of attention methods for GNNs have been developed. Examples of GNNs augmented with attention include *Graph Attention Networks*, by *Petar Veličković* and colleagues, 2017 (`https://arxiv.org/abs/1710.10903`) and *Relational Graph Attention Networks* by *Dan Busbridge* and colleagues, 2019 (`https://arxiv.org/abs/1904.05811`).

Recently, these attention mechanisms have also been utilized in graph transformers proposed by *Seongjun Yun* and colleagues, 2020 (`https://arxiv.org/abs/1911.06455`) and *Heterogeneous Graph Transformer* by *Ziniu Hu* and colleagues, 2020 (`https://arxiv.org/abs/2003.01332`).

Next to the aforementioned graph transformers, other deep generative models have been developed specifically for graphs. There are graph variational autoencoders such as those introduced in *Variational Graph Auto-Encoders* by *Kipf* and *Welling*, 2016 (`https://arxiv.org/abs/1611.07308`), *Constrained Graph Variational Autoencoders for Molecule Design* by Qi Liu and colleagues, 2018 (`https://arxiv.org/abs/1805.09076`), and *GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders* by *Simonovsky* and *Komodakis*, 2018 (`https://arxiv.org/abs/1802.03480`). Another notable graph variational autoencoder that has been applied to molecule generation is the *Junction Tree Variational Autoencoder for Molecular Graph Generation* by *Wengong Jin* and colleagues, 2019 (`https://arxiv.org/abs/1802.04364`).

Some GANs have been designed to generate graph data, though, as of this writing, the performance of GANs on graphs is much less convincing than in the image domain. Examples include *GraphGAN: Graph Representation Learning with Generative Adversarial Nets* by *Hongwei Wang* and colleagues, 2017 (`https://arxiv.org/abs/1711.08267`) and *MolGAN: An Implicit Generative Model for Small Molecular Graphs* by *Cao* and *Kipf*, 2018 (`https://arxiv.org/abs/1805.11973`).

GNNs have also been incorporated into deep reinforcement learning models—you will learn more about reinforcement learning in the next chapter. Examples include *Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation* by *Jiaxuan You* and colleagues, 2018 (`https://arxiv.org/abs/1806.02473`) and a deep Q-network proposed in *Optimization of Molecules via Deep Reinforcement Learning* by *Zhenpeng Zhou* and colleagues, 2018 (`https://arxiv.org/abs/1810.08678`), which utilizes a GNN that was applied to molecule generation tasks.

Lastly, while not technically graph data, 3D point clouds are sometimes represented as such using distance cutoffs to create edges. Applications of graph networks in this space include *Point-GNN: Graph Neural Network for 3D Object Detection in a Point Cloud* by *Weijing Shi* and colleagues, 2020 (`https://arxiv.org/abs/2003.01251`), which detects 3D objects in LiDAR point clouds. In addition, *GAPNet: Graph Attention based Point Neural Network for Exploiting Local Feature of Point Cloud* by *Can Chen* and colleagues, 2019 (`https://arxiv.org/abs/1905.08705`) was designed to detect local features in point cloud data, which had been challenging for other deep architectures.

Summary

As the amount of data we have access to continues to increase, so too will our need to understand interrelations within the data. While this will be done in numerous ways, graphs function as a distilled representation of these relationships, so the amount of graph data available will only increase.

In this chapter, we explained graph neural networks from the ground up by implementing a graph convolution layer and a GNN from scratch. We saw that implementing GNNs, due to the nature of graph data, is actually quite complex. Thus, to apply GNNs to a real-world example, such as predicting molecular polarization, we learned how to utilize the PyTorch Geometric library, which provides implementations of many of the building blocks we need. Lastly, we went over some of the notable literature for diving into the GNN literature more deeply.

Hopefully, this chapter provided an introduction to how deep learning can be leveraged to learn on graphs. Methods in this space are currently a hot area of research, and many of the ones we have mentioned were published in the last couple of years. With this text as a starting point, maybe the next advancement in the space can be made by you.

In the next chapter, we will look at reinforcement learning, which is a completely different category of machine learning compared to what we have covered so far in this book.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`