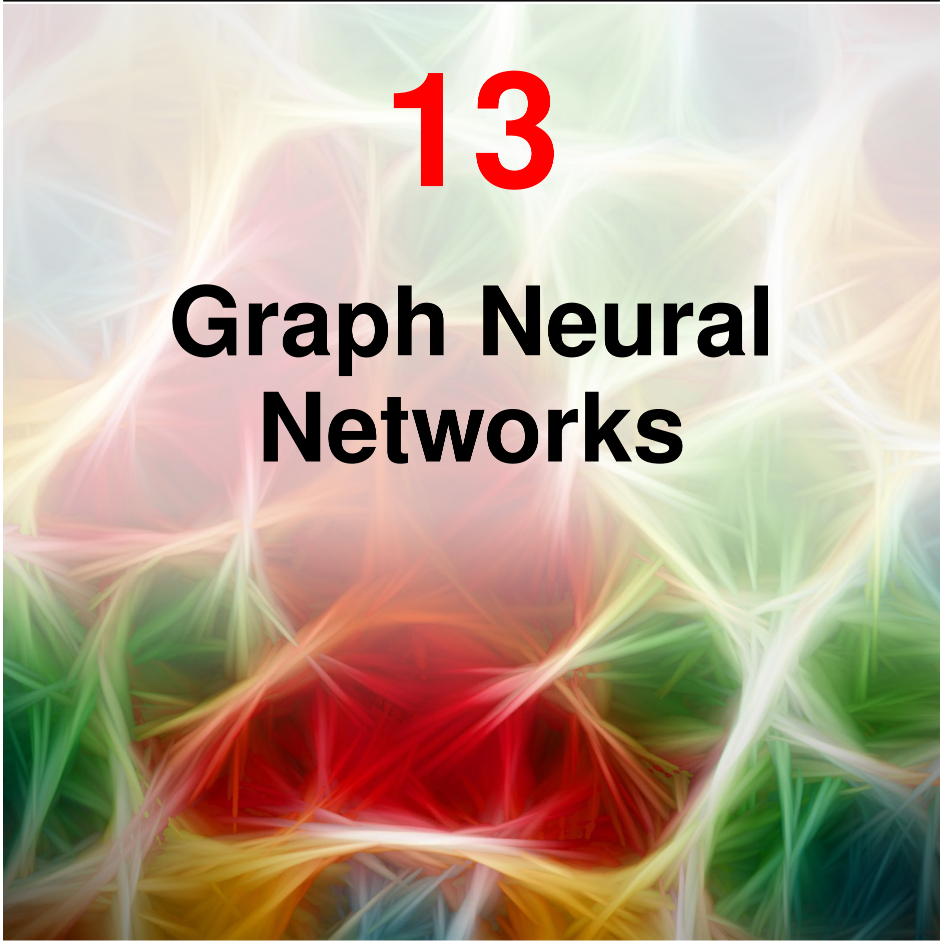


13 Graph Neural Networks

In previous chapters we have encountered *structured data* in the form of sequences and images, corresponding to one-dimensional and two-dimensional arrays of variables respectively. More generally, there are many types of structured data that are best described by a *graph* as illustrated in Figure 13.1. In general a graph consists of a set of objects, known as *nodes*, connected by *edges*. Both the nodes and the edges can have data associated with them. For example, in a molecule the nodes and edges are associated with discrete variables corresponding to the types of atom (carbon, nitrogen, hydrogen, etc.) and the types of bonds (single bond, double bond, etc.). For a rail network, each railway line might be associated with a continuous variable given by the average journey time between two cities. Here we are assuming that the edges are symmetrical, for example that the journey time from London to Cambridge is the same as the journey time from Cambridge to London. Such edges are depicted by undirected links between the nodes. For the worldwide web the edges are directed

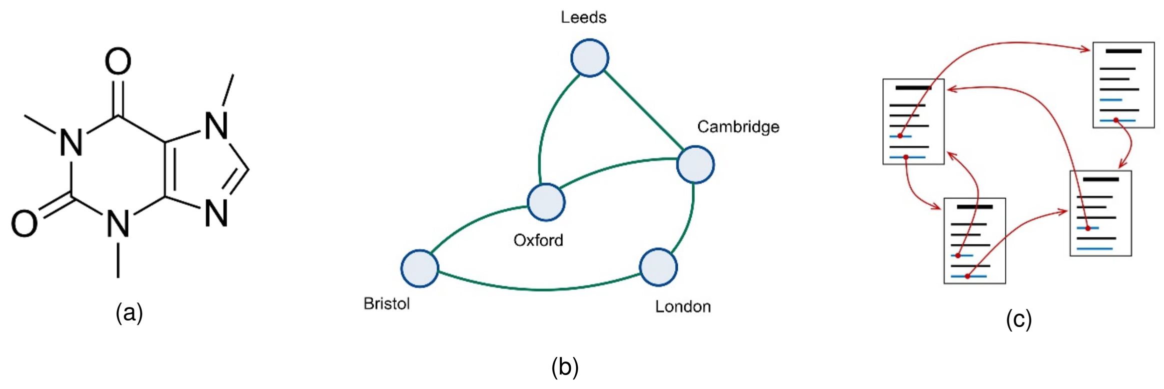


Figure 13.1 Three examples of graph-structured data: (a) the caffeine molecule consisting of atoms connected by chemical bonds, (b) a rail network consisting of cities connected by railway lines, and (c) the worldwide web consisting of pages connected by hyperlinks.

since if there is a hyperlink on page A that points to page B there is not necessarily a hyperlink on page B pointing back to page A.

Other examples of graph-structured data include a protein interaction network, in which the nodes are proteins and the edges express how strongly pairs of proteins interact, an electrical circuit where the nodes are components and the edges are conductors, or a social network where the nodes are people and the edges are 'friendships'. More complex graphical structures are also possible, for example the knowledge graph inside a company comprises multiple different kinds of nodes such as people, documents, and meetings, along with multiple kinds of edges capturing different properties such as a person being present at a meeting or a document referencing another document.

In this chapter we explore how to apply deep learning to graph-structured data. We have already encountered an example of structured data when we discussed images, in which the individual elements of an image data vector $\mathbf{x}$ correspond to pixels on a regular grid. An image is therefore a special instance of graph-structured data in which the nodes are the pixels and the edges describe which pixels are adjacent. Convolutional neural networks (CNNs) take this structure into account, incorporating prior knowledge of the relative positions of the pixels, together with the equivariance of properties such as segmentation and the invariance of properties such as classification. We will use CNNs for images as a source of inspiration to construct more general approaches to deep learning for graphical data known as *graph neural networks* (Zhou *et al.*, 2018; Wu *et al.*, 2019; Hamilton, 2020; Veličković, 2023). We will see that a key consideration when applying deep learning to graph-structured data is to ensure either equivariance or invariance with respect to a reordering of the nodes in the graph.

Chapter 10

13.1. Machine Learning on Graphs

There are many kinds of applications that we might wish to address using graph-structured data, and we can group these broadly according to whether the goal is to predict properties of nodes, of edges, or of the whole graph. An example of node prediction would be to classify documents according to their topic based on the hyperlinks and citations between the documents.

Regarding edges we might, for example, know some of the interactions in a protein network and would like to predict the presence of any additional ones. Such tasks are called *edge prediction* or *graph completion* tasks. There are also tasks where the edges are known in advance and the goal is to discover clusters or 'communities' within the graph.

Finally, we may wish to predict properties that relate to the graph as a whole. For example, we might wish to predict whether a particular molecule is soluble in water. Here instead of being given a single graph we will have a data set of different graphs, which we can view as being drawn from some common distribution, in other words we assume that the graphs themselves are *independent and identically distributed*. Such tasks can be considered as graph regression or graph classification tasks.

For the molecule solubility classification example, we might be given a labelled training set of molecules, along with a test set of new molecules whose solubility needs to be predicted. This is a standard example of an *inductive* task of the kind we have seen many times in previous chapters. However, some graph prediction examples are *transductive* in which we are given the structure of the entire graph along with labels for some of the nodes and the goal is to predict the labels of the remaining nodes. An example would be a large social network in which our goal is to classify each node as either a real person or an automated bot. Here a small number of nodes might be manually labelled, but it would be prohibitive to investigate every node individually in a large and ever-changing social network. During training, we therefore have access to the whole graph along with labels for a subset of the nodes, and we wish to predict the labels for the remaining nodes. This can be viewed as a form of semi-supervised learning.

As well as solving prediction tasks directly, we can also use deep learning on graphs to discover useful internal representations that can subsequently facilitate a range of downstream tasks. This is known as *graph representation learning*. For example we could seek to build a *foundation model* for molecules by training a deep learning system on a large corpus of molecular structures. The goal is that once trained, such a foundation model can be fine-tuned to specific tasks by using a small, labelled data set.

Graph neural networks define an *embedding vector* for each of the nodes, usually initialized with the observed node properties, which are then transformed through a series of learnable layers to create a learned representation. This is analogous to the way word embeddings, or tokens, are processed through a series of layers in the transformer to give a representation that better captures the meaning of the words in the context of the rest of the text. Graph neural networks can also use learned embeddings associated with the edges and with the graph as a whole.

Chapter 12

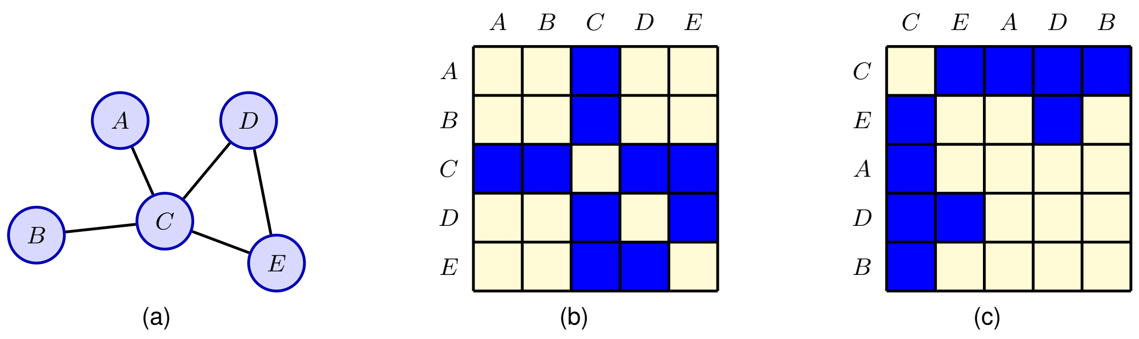


Figure 13.2 An example of an adjacency matrix showing (a) an example of a graph with five nodes, (b) the associated adjacency matrix for a particular choice of node order, and (c) the adjacency matrix corresponding to a different choice for the node order.

13.1.1 Graph properties

In this chapter we will focus on *simple graphs* where there is at most one edge between any pair of nodes, where the edges are undirected, and where there are no self-edges that connect a node to itself. This suffices to introduce the key concepts of graph neural networks, and it also encompasses a wide range of practical applications. These concepts can then be applied to more complex graphical structures.

We begin by introducing some notation associated with graphs and by defining some important properties. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of *nodes* or *vertices*, denoted by $\mathcal{V}$, along with a set of *edges* or *links*, denoted by $\mathcal{E}$. We index the nodes by $n = 1, \ldots N$, and we write the edge from node n to node m as (n, m). If two nodes are linked by an edge they are called *neighbours*, and the set of all neighbours of node n is denoted by $\mathcal{N}(n)$.

In addition to the graph structure, we usually also have observed data associated with the nodes. For each node n we can represent the corresponding node variables as a D-dimensional column vector $\mathbf{x}_n$ and we can group these into a data matrix $\mathbf{X}$ of dimensionality $N \times D$ in which row n is given by $\mathbf{x}_n^{\mathrm{T}}$. There may also be data variables associated with the edges in the graph, although to start with we will focus just on node variables.

Section 13.3.2

13.1.2 Adjacency matrix

A convenient way to specify the edges in a graph is to use an *adjacency matrix* denoted by $\mathbf{A}$. To define the adjacency matrix we first have to choose an ordering for the nodes. If there are N nodes in the graph, we can index them using $n = 1, \ldots, N$. The adjacency matrix has dimensions $N \times N$ and contains a 1 in every location n, m for which there is an edge going from node n to node m, with all other entries being 0. For graphs with undirected edges, the adjacency matrix will be symmetric since the presence of an edge from node n to node m implies that there is also an edge from node m to node n, and therefore $A_{mn} = A_{nm}$ for all n and m. An example of an adjacency matrix is shown in Figure 13.2.

Since the adjacency matrix defines the structure of a graph, we could consider using it directly as the input to a neural network. To do this we could 'flatten' the matrix, for example by concatenating the columns into one long column vector. However, a major problem with this approach is that the adjacency matrix depends on the arbitrary choice of node ordering, as seen in Figure 13.2. Suppose for instance that we want to predict the solubility of a molecule. This clearly should not depend on the ordering assigned to the nodes when writing down an adjacency matrix. Because the number of permutations increases factorially with the number of nodes, it is impractical to try to learn permutation invariance by using large data sets or by data augmentation. Instead, we should treat this invariance property as an inductive bias when constructing a network architecture.

13.1.3 Permutation equivariance

We can express node label permutation mathematically by introducing the concept of a *permutation matrix* $\mathbf{P}$, which has the same size as the adjacency matrix and which specifies a particular permutation of a node ordering. It contains a single 1 in each row and a single 1 in each column, with 0 in all the other elements, such that a 1 in position n, m indicates that node n will be relabelled as node m after the permutation. Consider, for example, the permutation from $(A, B, C, D, E) \rightarrow (C, E, A, D, B)$ corresponding to the two choices of node ordering in Figure 13.2. The corresponding permutation matrix takes the form

Exercise 13.1

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{13.1}$$

We can define the permutation matrix more formally as follows. First we introduce the *standard unit vector* $\mathbf{u}_n$, for $n = 1, \ldots, N$. This is a column vector in which all elements are 0 except element n, which equals 1. In this notation the identity matrix is given by

$$\mathbf{I} = \begin{pmatrix} \mathbf{u}_1^{\mathrm{T}} \\ \mathbf{u}_2^{\mathrm{T}} \\ \cdots \\ \mathbf{u}_N^{\mathrm{T}} \end{pmatrix}. \tag{13.2}$$

We can now introduce a permutation function $\pi(\cdot)$ that maps n to $m = \pi(n)$. The associated permutation matrix is given by

$$\mathbf{P} = \begin{pmatrix} \mathbf{u}_{\pi(1)}^{\mathrm{T}} \\ \mathbf{u}_{\pi(2)}^{\mathrm{T}} \\ \cdots \\ \mathbf{u}_{\pi(N)}^{\mathrm{T}} \end{pmatrix}. \tag{13.3}$$

When we reorder the labelling on the nodes of a graph, the effect on the corresponding node data matrix $\mathbf{X}$ is to permute the rows according to $\pi(\cdot)$, which can be achieved by pre-multiplication by $\mathbf{P}$ to give

Exercise 13.4

$$\widetilde{\mathbf{X}} = \mathbf{PX}. \tag{13.4}$$

For the adjacency matrix, both the rows and the columns become permuted. Again the rows can be permuted using pre-multiplication by $\mathbf{P}$ whereas the columns are permuted using post-multiplication by $\mathbf{P}^{\mathrm{T}}$, giving a new adjacency matrix:

Exercise 13.5

$$\widetilde{\mathbf{A}} = \mathbf{PAP}^{\mathrm{T}}. \tag{13.5}$$

When applying deep learning to graph-structured data, we will need to represent the graph structure in numerical form so that it can be fed into a neural network, which requires that we assign an ordering to the nodes. However, the specific ordering we choose is arbitrary and so it will be important to ensure that any global property of the graph does not depend on this ordering. In other words, the network predictions must be *invariant* to node label reordering, so that

$$y(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) = y(\mathbf{X}, \mathbf{A}) \qquad \text{Invariance} \tag{13.6}$$

where $y(\cdot, \cdot)$ is the output of the network.

We may also want to make predictions that relate to individual nodes. In this case, if we reorder the node labelling then the corresponding predictions should show the same reordering so that a given prediction is always associated with the same node irrespective of the choice of order. In other words, node predictions should be *equivariant* with respect to node label reordering. This can be expressed as

$$\mathbf{y}(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) = \mathbf{P}\mathbf{y}(\mathbf{X}, \mathbf{A}) \qquad \text{Equivariance} \tag{13.7}$$

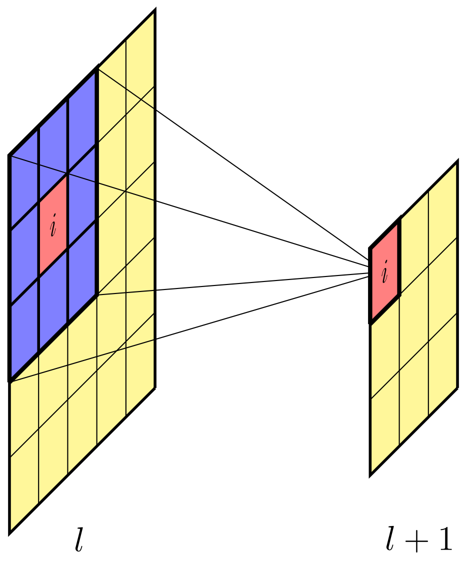
where $\mathbf{y}(\cdot, \cdot)$ is a vector of network outputs, with one element per node.

13.2. Neural Message-Passing

Ensuring invariance or equivariance under node label permutations is a key design consideration when we apply deep neural networks to graph-structured data. Another consideration is that we want to exploit the representational capabilities of deep neural networks and so we retain the concept of a 'layer' as a computational transformation that can be applied repeatedly. If each layer of the network is equivariant under node reordering then multiple layers applied in succession will also exhibit equivariance, while allowing each layer of the network to be informed by the graph structure.

For networks whose outputs represent node-level predictions, the whole network will be equivariant as required. If the network is being used to predict a graph-level property then a final layer can be included that is invariant to permutations of its inputs. We also want to ensure that each layer is a highly flexible nonlinear function and is differentiable with respect to its parameters so that it can be trained by stochastic gradient descent using gradients obtained by automatic differentiation.

Graphs come in various sizes. For example different molecules can have different numbers of atoms, so a fixed-length representation as used for standard neural



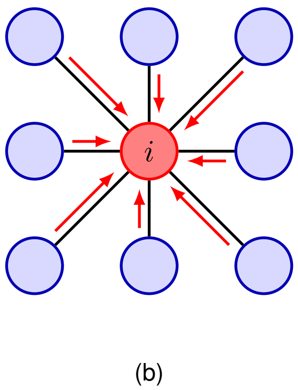


Figure 13.3 A convolutional filter for images can be represented as a graph-structured computation. (a) A filter computed by node i in layer $l+1$ of a deep convolutional network is a function of the activation values in layer l over a local patch of pixels. (b) The same computation structure expressed as a graph showing 'messages' flowing into node i from its neighbours.

networks is unsuitable. A further requirement is therefore that the network should be able to handle variable-length inputs, as we saw with transformer networks. Some graphs can be very large, for example a social network with many millions of participants, and so we also want to construct models that scale well. Not surprisingly, parameter sharing will play an important role, both to allow the invariance and equivariance properties to be built into the network architecture but also to facilitate scaling to large graphs.

Chapter 12

13.2.1 Convolutional filters

To develop a framework that meets all of these requirements, we can seek inspiration from image processing using convolutional neural networks. First note that an image can be viewed as a specific instance of graph-structured data, in which the nodes are the pixels and the edges represent pairs of pixels that are adjacent in the image, where adjacency includes nodes that are diagonally adjacent as well as those that are horizontally or vertically adjacent.

Chapter 10

In a convolutional network, we make successive transformations of the image domain such that a pixel at a particular layer computes a function of states of pixels in the previous layer through a local function called a *filter*. Consider a convolutional layer using 3×3 filters, as illustrated in Figure 13.3(a). The computation performed

Section 10.2

by a single filter at a single pixel in layer $l+1$ can be expressed as

$$z_i^{(l+1)} = f\left(\sum_j w_j z_j^{(l)} + b\right) \tag{13.8}$$

where $f(\cdot)$ is a differentiable nonlinear activation function such as ReLU, and the sum over j is taken over all nine pixels in a small patch in layer l. The same function is applied across multiple patches in the image, so that the weights w_j and bias b are shared across the patches (and therefore do not carry the index i).

As it stands, (13.8) is not equivariant under reordering of the nodes in layer l because the weight vector, with elements w_j, is not invariant under permutation of its elements. However, we can achieve equivariance with some simple modifications as follows. We first view the filter as a graph, as shown in Figure 13.3(b), and separate out the contribution from node i. The other eight 8 nodes are its neighbours $\mathcal{N}(i)$. We then assume that a single weight parameter w_{neigh} is shared across the neighbours so that

$$z_i^{(l+1)} = f\left(w_{\text{neigh}} \sum_{j\in\mathcal{N}(i)} z_j^{(l)} + w_{\text{self}} z_i^{(l)} + b\right) \tag{13.9}$$

where node i has its own weight parameter w_{self}.

We can interpret (13.9) as updating a local representation z_i at node i by gathering information from the neighbouring nodes by passing *messages* from the neighbouring nodes into node i. In this case the messages are simply the activations of the other nodes. These messages are then combined with information from node i, and the result is transformed using a nonlinear function. The information from the neighbouring nodes is *aggregated* through a simple summation in (13.9), and this is clearly invariant to any permutation of the labels associated with those nodes. Furthermore, the operation (13.9) is applied synchronously to every node in a graph, and so if the nodes are permuted then the resulting computations will be unchanged but their ordering will be likewise permuted, and hence, this calculation is equivariant under node reordering. Note that this depends on the parameters w_{neigh}, w_{self}, and b being shared across all nodes.

13.2.2 Graph convolutional networks

We now use the convolution example as a template to construct deep neural networks for graph-structured data. Our goal is to define a flexible, nonlinear transformation of the node embeddings that is differentiable with respect to a set of weight and bias parameters and which maps the variables in layer l into corresponding variables in layer $l+1$. For each node n in the graph and for each layer l in the network, we introduce a D-dimensional column vector $\mathbf{h}_n^{(l)}$ of node-embedding variables, where $n = 1, \ldots, N$ and $l = 1, \ldots, L$.

We see that the transformation given by (13.9) first gathers and combines information from neighbouring nodes and then updates the node as a function of the

Algorithm 13.1: Simple message-passing neural network

```
Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
       Initial node embeddings $\{\mathbf{h}_n^{(0)} = \mathbf{x}_n\}$
       $\mathrm{Aggregate}(\cdot)$ function
       $\mathrm{Update}(\cdot, \cdot)$ function
Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$
// Iterative message-passing
for $l \in \{0, \ldots, L-1\}$ do
    $\mathbf{z}_n^{(l)} \leftarrow \mathrm{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right)$
    $\mathbf{h}_n^{(l+1)} \leftarrow \mathrm{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right)$
end for
return $\{\mathbf{h}_n^{(L)}\}$
```

current embedding of the node and the incoming messages. We can therefore view each layer of processing as having two successive stages. The first is the *aggregation* stage in which, for each node n, messages are passed to that node from its neighbours and combined to form a new vector $\mathbf{z}_n^{(l)}$ in a way that is permutation invariant. This is followed by an *update* step in which the aggregated information from neighbouring nodes is combined with local information from the node itself and used to calculate a revised embedding vector for that node.

Consider a specific node n in the graph. We first aggregate the node vectors from all the neighbours of node n:

$$\mathbf{z}_n^{(l)} = \mathrm{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right). \tag{13.10}$$

The form of this aggregation function is very flexible if it is well defined for a variable number of neighbouring nodes and does not depend on the ordering of those nodes. It can potentially contain learnable parameters as long as it is a differentiable function with respect to those parameters to facilitate gradient descent training.

We then use another operation to update the embedding vector at node n:

$$\mathbf{h}_n^{(l+1)} = \mathrm{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right). \tag{13.11}$$

Again, this can be a differentiable function of a set of learnable parameters. Application of the Aggregate operation followed by the Update operation in parallel for every node in the graph represents one layer of the network. The node embeddings are typically initialized using observed node data so that $\mathbf{h}_n^{(0)} = \mathbf{x}_n$. Note that each layer generally has its own independent parameters, although the parameters can also be shared across layers. This framework is called a *message-passing neural network* (Gilmer *et al.*, 2017) and is summarized in Algorithm 13.1.

13.2.3 Aggregation operators

There are many possible forms for the Aggregate function, but it must depend only on the set of inputs and not on their ordering. It must also be a differentiable function of any learnable parameters. The simplest such aggregation function, following from (13.9), is summation:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)}. \tag{13.12}$$

A simple summation is clearly independent of the ordering of the neighbouring nodes and is also well defined no matter how many nodes are in the neighbourhood set. Note that this has no learnable parameters.

A summation gives a stronger influence over nodes that have many neighbours compared to those with few neighbours, and this can lead to numerical issues, particularly in applications such as social networks where the size of the neighbourhood set can vary by several orders of magnitude. A variation of this approach is to define the Aggregation operation to be the average of the neighbouring embedding vectors so that

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \frac{1}{|\mathcal{N}(n)|} \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)} \tag{13.13}$$

where $|\mathcal{N}(n)|$ denotes the number of nodes in the neighbourhood set $\mathcal{N}(n)$. However, this normalization also discards information about the network structure and is provably less powerful than a simple summation (Hamilton, 2020), and so the choice of whether to use it depends on the relative importance of node features compared to graph structure.

Another variation of this approach (Kipf and Welling, 2016) takes account of the number of neighbours for each of the neighbouring nodes:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \sum_{m \in \mathcal{N}(n)} \frac{\mathbf{h}_m^{(l)}}{\sqrt{|\mathcal{N}(n)|\,|\mathcal{N}(m)|}}. \tag{13.14}$$

Yet another possibility is to take the element-wise maximum (or minimum) of the neighbouring embedding vectors, which also satisfies the desired properties of being well defined for a variable number of neighbours and of being independent of their order.

Since each node in a given layer of the network is updated by aggregating information from its neighbours in the previous layer, this defines a *receptive field* analogous to the receptive fields of filters used in CNNs. *Chapter 10* As information is processed through successive layers, the updates to a given node depend on a steadily increasing fraction of other nodes in earlier layers until the effective receptive field potentially spans the whole graph as illustrated in Figure 13.4. However, large, sparse graphs may require an excessive number of layers before each output is influenced by every input. Some architectures therefore introduce an additional 'super-node'

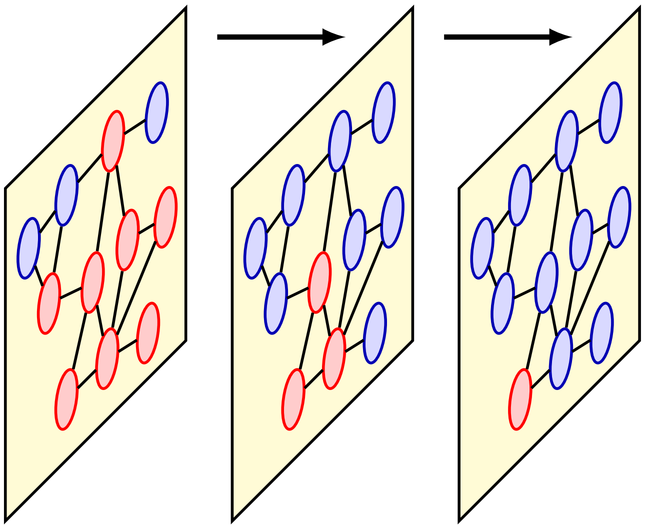


Figure 13.4 Schematic illustration of information flow through successive layers of a graph neural network. In the third layer a single node is highlighted in red. It receives information from its two neighbours in the previous layer and those in turn receive information from their neighbours in the first layer. As with convolutional neural networks for images, we see that the effective receptive field, corresponding to the number of nodes shown in red, grows with the number of processing layers.

that connects directly to every node in the original graph to ensure fast propagation of information.

The aggregation operators discussed so far have no learnable parameters. We can introduce such parameters if we first transform each of the embedding vectors from neighbouring nodes using a multilayer neural network, denoted by $\mathrm{MLP}_{\boldsymbol{\phi}}$, before combining their outputs, where MLP denotes 'multilayer perceptron' and $\boldsymbol{\phi}$ represents the parameters of the network. So long as the network has a structure and parameter values that are shared across nodes then this aggregation operator again be permutation invariant. We can also transform the combined vector with another neural network $\mathrm{MLP}_{\boldsymbol{\theta}}$, with parameters $\boldsymbol{\theta}$, to give an overall aggregation operator:

$$\mathrm{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right) = \mathrm{MLP}_{\boldsymbol{\theta}}\left(\sum_{m\in\mathcal{N}(n)} \mathrm{MLP}_{\boldsymbol{\phi}}(\mathbf{h}_m^{(l)})\right) \tag{13.15}$$

in which $\mathrm{MLP}_{\boldsymbol{\phi}}$ and $\mathrm{MLP}_{\boldsymbol{\theta}}$ are shared across layer l. Due to the flexibility of MLPs, the transformation defined by (13.15) represents a universal approximator for any permutation-invariant function that maps a set of embeddings to a single embedding (Zaheer *et al.*, 2017). Note that the summation can be replaced by other invariant functions such as averages or an element-wise maximum or minimum.

A special case of graph neural networks arises if we consider a graph having no edges, which corresponds simply to an unstructured set of nodes. In this case if we use (13.15) for each vector $\mathbf{h}_n^{(l)}$ in the set, in which the summation is taken over all other vectors except $\mathbf{h}_n^{(l)}$, then we have a general framework for learning functions over unstructured sets of variables known as *deep sets*.

13.2.4 Update operators

Having chosen a suitable Aggregate operator, we similarly need to decide on the form of the Update operator. By analogy with (13.9) for the CNN, a simple form for this operator would be

$$\mathrm{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right) = f\left(\mathbf{W}_{\mathrm{self}}\mathbf{h}_n^{(l)} + \mathbf{W}_{\mathrm{neigh}}\mathbf{z}_n^{(l)} + \mathbf{b}\right) \tag{13.16}$$

where $f(\cdot)$ is a nonlinear activation function such as ReLU applied element-wise to its vector argument, and where $\mathbf{W}_{\mathrm{self}}$, $\mathbf{W}_{\mathrm{neigh}}$, and $\mathbf{b}$ are the learnable weights and biases and $\mathbf{z}_n^{(l)}$ is defined by the Aggregate operator (13.10).

If we choose a simple summation (13.12) as the aggregation function and if we also share the same weight matrix between nodes and their neighbours so that $\mathbf{W}_{\mathrm{self}} = \mathbf{W}_{\mathrm{neigh}}$, we obtain a particularly simple form of Update operator given by

$$\mathbf{h}_n^{(l+1)} = \mathrm{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right) = f\left(\mathbf{W}_{\mathrm{neigh}} \sum_{m \in \mathcal{N}(n), n} \mathbf{h}_m^{(l)} + \mathbf{b}\right). \tag{13.17}$$

The message-passing algorithm is typically initialized by setting $\mathbf{h}_n^{(0)} = \mathbf{x}_n$. Sometimes, however, we may want to have an internal representation vector for each node that has a higher, or lower, dimensionality than that of $\mathbf{x}_n$. Such a representation can be initialized by padding the node vectors $\mathbf{x}_n$ with additional zeros (to achieve a higher dimensionality) or simply by transforming the node vectors using a learnable linear transformation to a space of the desired number of dimensions. An alternative form of initialization, particularly when there are no data variables associated with the nodes, is to use a one-hot vector that labels the degree of each node (i.e., the number of neighbours).

Overall, we can represent a graph neural network as a sequence of layers that successively transform the node embeddings. If we group these embeddings into a matrix $\mathbf{H}$ whose nth row is the vector $\mathbf{h}_n^{\mathrm{T}}$, which is initialized to the data matrix $\mathbf{X}$, then we can write the successive transformations in the form

$$\begin{aligned}
\mathbf{H}^{(1)} &= \mathbf{F}\left(\mathbf{X}, \mathbf{A}, \mathbf{W}^{(1)}\right) \\
\mathbf{H}^{(2)} &= \mathbf{F}\left(\mathbf{H}^{(1)}, \mathbf{A}, \mathbf{W}^{(2)}\right) \\
\vdots &= \vdots \\
\mathbf{H}^{(L)} &= \mathbf{F}\left(\mathbf{H}^{(L-1)}, \mathbf{A}, \mathbf{W}^{(L)}\right)
\end{aligned} \tag{13.18}$$

where $\mathbf{A}$ is the adjacency matrix, and $\mathbf{W}^{(l)}$ represents the complete set of weight and biases in layer l of the network. Under a node reordering defined by a permutation matrix $\mathbf{P}$, the transformation of the node embeddings computed by layer l is equivariant:

$$\mathbf{P}\mathbf{H}^{(l)} = \mathbf{F}\left(\mathbf{P}\mathbf{H}^{(l-1)}, \mathbf{P}\mathbf{A}\mathbf{P}^{\mathrm{T}}, \mathbf{W}^{(l)}\right). \tag{13.19}$$

Exercise 13.7

As a consequence, the complete network computes an equivariant transformation.

13.2.5 Node classification

A graph neural network can be viewed as a series of layers each of which transforms a set of node-embedding vectors $\{\mathbf{h}_n^{(l)}\}$ into a new set $\{\mathbf{h}_n^{(l+1)}\}$ of the same size and dimensionality. After the final convolutional layer of the network, we need to obtain predictions so that we can define a cost function for training and also for making predictions on new data using the trained network.

Consider first the task of classifying the nodes in a graph, which is one of the most common uses for graph neural networks. We can define an output layer, sometimes called a *readout layer*, which calculates a softmax function for each node corresponding to a classification over C classes, of the form

$$y_{ni} = \frac{\exp(\mathbf{w}_i^{\mathrm{T}}\mathbf{h}_n^{(L)})}{\sum_j \exp(\mathbf{w}_j^{\mathrm{T}}\mathbf{h}_n^{(L)})} \tag{13.20}$$

where $\{\mathbf{w}_i\}$ is a set of learnable weight vectors and $i = 1, \ldots, C$. We can then define a loss function as the sum of the cross-entropy loss across all nodes and all classes:

$$\mathcal{L} = -\sum_{n\in\mathcal{V}_{\text{train}}} \sum_{i=1}^{C} y_{ni}^{t_{ni}} \tag{13.21}$$

where $\{t_{ni}\}$ are target values with a one-hot encoding for each value of n. Because the weight vectors $\{\mathbf{w}_i\}$ are shared across the output nodes, the outputs y_{ni} are equivariant to permutation of the node ordering, and hence the loss function (13.21) is invariant. If the goal is to predict continuous values at the outputs then a simple linear transformation can be combined with a sum-of-squares error to define a suitable loss function.

The sum over n in (13.21) is taken over the subset of the nodes denoted by $\mathcal{V}_{\text{train}}$ and used for training. We can distinguish between three types of nodes as follows:

1. The nodes $\mathcal{V}_{\text{train}}$ are labelled and included in the message-passing operations of the graph neural network and are also used to compute the loss function used for training.

2. There is potentially also a *transductive* subset of nodes denoted by $\mathcal{V}_{\text{trans}}$, which are unlabelled and which do not contribute to the evaluation of the loss function used for training. However, they still participate in the message-passing operations during both training and inference, and their labels may be predicted as part of the inference process.

3. The remaining nodes, denoted $\mathcal{V}_{\text{induct}}$, are a set of *inductive* nodes that are not used to compute the loss function, and neither these nodes nor their associated edges participate in message-passing during the training phase. However, they do participate in message-passing during the inference phase and their labels are predicted as the outcome of inference.

If there are no transductive nodes, and hence the test nodes (and their associated edges) are not available during the training phase, then the training is generally referred to as *inductive learning*, which can be considered to be a form of *supervised learning*. However, if there are transductive nodes then it is called *transductive learning*, which may be viewed as a form of *semi-supervised* learning.

13.2.6 Edge classification

In some applications we wish to make predictions about the edges of the graph rather than the nodes. A common form of edge classification task is edge completion in which the goal is to determine whether an edge should be present between two nodes. Given a set of node embeddings, the dot product between pairs of embeddings can be used to define a probability $p(n, m)$ for the presence of an edge between nodes n and m by using the logistic sigmoid function:

$$p(n, m) = \sigma\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{h}_m\right). \tag{13.22}$$

An example application would be predicting whether two people in a social network have shared interests and therefore might wish to connect.

13.2.7 Graph classification

In some applications of graph neural networks, the goal is to predict the properties of new graphs given a training set of labelled graphs $\mathcal{G}_1, \ldots, \mathcal{G}_N$. This requires that we combine the final-layer embedding vectors in a way that does not depend on the arbitrary node ordering, thereby ensuring that the output predictions will be invariant to that ordering. The goal is somewhat like that of the Aggregate function except that all nodes in the graph are included, not just the neighbourhood sets of the individual nodes. The simplest approach is to take the sum of the node-embedding vectors:

$$\mathbf{y} = \mathbf{f}\left(\sum_{n\in\mathcal{V}} \mathbf{h}_n^{(L)}\right) \tag{13.23}$$

where the function $\mathbf{f}$ may contain learnable parameters such as a linear transformation or a neural network. Other invariant aggregation functions can be used such as averages or element-wise minimum or maximum.

A cross-entropy loss is typically used for classification problems, such as labelling a candidate drug molecule as toxic or safe, and a squared-error loss for regression problems, such as predicting the solubility of a candidate drug molecule. Graph-level predictions correspond to an inductive task since there must be separate sets of graphs for training and for inference.

13.3. General Graph Networks

There are many variations and extensions of the graph networks considered so far. Here we outline a few of the key concepts along with some practical considerations.

13.3.1 Graph attention networks

Section 12.1

The attention mechanism is very powerful when used as the basis of a transformer architecture. It can be used in the context of graph neural networks to construct an aggregation function that combines messages from neighbouring nodes. The incoming messages are weighted by attention coefficients A_{nm} to give

$$\mathbf{z}_n^{(l)} = \text{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right) = \sum_{m \in \mathcal{N}(n)} A_{nm}\mathbf{h}_m^{(l)} \tag{13.24}$$

where the attention coefficients satisfy

$$A_{nm} \geqslant 0 \tag{13.25}$$

$$\sum_{m \in \mathcal{N}(n)} A_{nm} = 1. \tag{13.26}$$

This is known as a *graph attention network* (Veličković *et al.*, 2017) and can capture an inductive bias that says some neighbouring nodes will be more important than others in determining the best update in a way that depends on the data itself.

There are multiple ways to construct the attention coefficients, and these generally employ a softmax function. For example, we can use a bilinear form:

$$A_{nm} = \frac{\exp\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{W}\mathbf{h}_m\right)}{\sum_{m' \in \mathcal{N}(n)} \exp\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{W}\mathbf{h}_{m'}\right)} \tag{13.27}$$

where $\mathbf{W}$ is a $D \times D$ matrix of learnable parameters. A more general option is to use a neural network to combine the embedding vectors from the nodes at each end of the edge:

$$A_{nm} = \frac{\exp\left\{\text{MLP}\left(\mathbf{h}_n, \mathbf{h}_m\right)\right\}}{\sum_{m' \in \mathcal{N}(n)} \exp\left\{\text{MLP}\left(\mathbf{h}_n, \mathbf{h}_{m'}\right)\right\}} \tag{13.28}$$

where the MLP has a single continuous output variable whose value is invariant if the input vectors are exchanged. Provided the MLP is shared across all the nodes in the network, this aggregation function will be equivariant under node reordering.

Exercise 13.8

Section 12.1.6

A graph attention network can be extended by introducing multiple attention heads in which H distinct sets of attention weights $A_{nm}^{(h)}$ are defined, for $h = 1, \ldots, H$, in which each head is evaluated using one of the mechanisms described above and with its own independent parameters. These are then combined in the aggregation step using concatenation and linear projection. Note that, for a fully-connected network, a multi-head graph attention network becomes a standard transformer encoder.

Exercise 13.9

13.3.2 Edge embeddings

The graph neural networks discussed above use embedding vectors that are associated with the nodes. We have seen that some networks also have data associated with the edges. Even when there are no observable values associated with the edges, we can still maintain and update edge-based hidden variables and these can contribute to the internal representations learned by the graph neural network.

In addition to the node embeddings given by $\mathbf{h}_n^{(l)}$, we therefore introduce edge embeddings $\mathbf{e}_{nm}^{(l)}$. We can then define general message-passing equations in the form

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}\right) \tag{13.29}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right) \tag{13.30}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}\right). \tag{13.31}$$

The learned edge embeddings $\mathbf{e}_{nm}^{(L)}$ from the final layer can be used directly to make predictions associated with the edges.

13.3.3 Graph embeddings

In addition to node and edge embeddings we can also maintain and update an embedding vector $\mathbf{g}^{(l)}$ that relates to the graph as a whole. Bringing all these aspects together allows us to define a more general set of message-passing functions, and a richer set of learned representations, for graph-structured applications. Specifically, we can define general message-passing equations (Battaglia *et al.*, 2018):

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right) \tag{13.32}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right) \tag{13.33}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right) \tag{13.34}$$

$$\mathbf{g}^{(l+1)} = \text{Update}_{\text{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)} : n \in \mathcal{V}\}, \{\mathbf{e}_{nm}^{(l+1)} : (n,m) \in \mathcal{E}\}\right). \tag{13.35}$$

These update equations start in (13.32) by updating the edge embedding vectors $\mathbf{e}_{nm}^{(l+1)}$ based on the previous states of those vectors, on the node embeddings for the nodes connected by each edge, and on a graph-level embedding vector $\mathbf{g}^{(l)}$. These updated edge embeddings are then aggregated across every edge connected to each node using (13.33) to give a set of aggregated vectors. These in turn then contribute to the update of the node-embedding vector $\{\mathbf{h}_n^{(l+1)}\}$ based on the current node-embedding vectors and on the graph-level embedding vector using (13.34). Finally, the graph-level embedding vector is updated using (13.35) based on information from all the nodes and all the edges in the graph along with the graph-level embedding from the previous layer. These message-passing updates are illustrated in Figure 13.5 and are summarized in Algorithm 13.2.

13.3.4 Over-smoothing

One significant problem that can arise with some graph neural networks is called *over-smoothing* in which the node-embedding vectors tend to become very similar to each other after a number of iterations of message-passing, which effectively limits the depth of the network. One way to help alleviate this issue is to introduce residual connections. For example, we can modify the update operator (13.34): *Section 9.5*

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right) + \mathbf{h}_n^{(l)}. \tag{13.36}$$

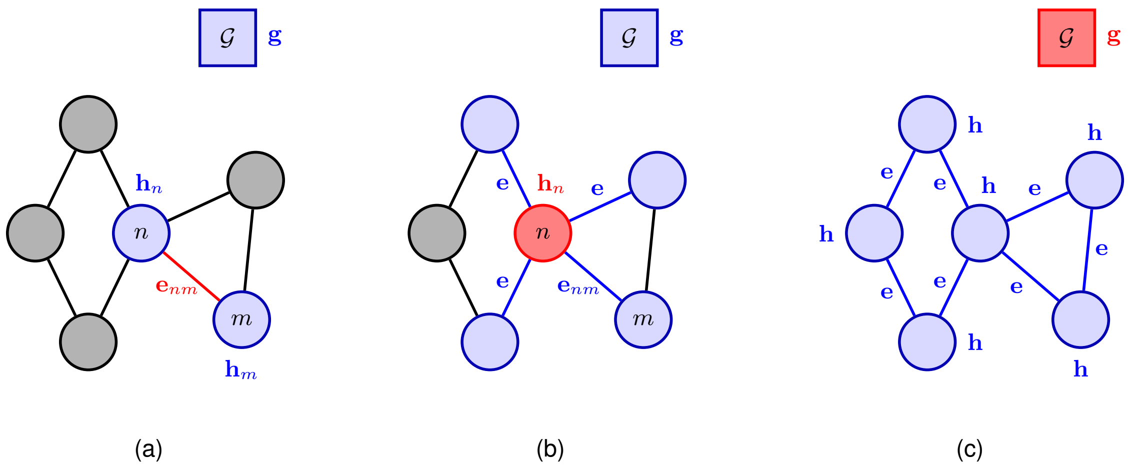


Figure 13.5 Illustration of the general graph message-passing updates defined by (13.32) to (13.35), showing (a) edge updates, (b) node updates, and (c) global graph updates. In each case the variable being updated is shown in red and the variables that contribute to that update are those shown in red and blue.

Another approach for mitigating the effects of over-smoothing is to allow the output layer to take information from all previous layers of the network and not just the final convolutional layer. This can be done for example by concatenating the representations from previous layers:

$$\mathbf{y}_n = \mathbf{f}\left(\mathbf{h}_n^{(1)} \oplus \mathbf{h}_n^{(2)} \oplus \cdots \oplus \mathbf{h}_n^{(L)}\right) \tag{13.37}$$

where $\mathbf{a} \oplus \mathbf{b}$ denotes the concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$. A variant of this would be to combine the vectors using max pooling instead of concatenation. In this case each element of the output vector is given by the max of all the corresponding elements of the embedding vectors from the previous layers.

13.3.5 Regularization

Chapter 9

Standard techniques for regularization can be used with graph neural networks, including the addition of penalty terms, such as the sum-of-squares of the parameter values, to the loss function. In addition, some regularization methods have been developed specifically for graph neural networks.

Graph neural networks already employ weight sharing to achieve permutation equivariance and invariance, but typically they have independent parameters in each layer. However, weights and biases can also be shared across layers to reduce the number of independent parameters.

Dropout in the context of graph neural networks involves omitting random subsets of the graph nodes during training, with a fresh random subset chosen for each forward pass. This can likewise be applied to the edges in the graph in which randomly selected subsets of entries in the adjacency matrix are removed, or masked, during training.

Algorithm 13.2: Graph neural network with node, edge, and graph embeddings

```
Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
       Initial node embeddings $\{\mathbf{h}_n^{(0)}\}$
       Initial edge embeddings $\{\mathbf{e}_{nm}^{(0)}\}$
       Initial graph embedding $\mathbf{g}^{(0)}$
Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$
        Final edge embeddings $\{\mathbf{e}_{nm}^{(L)}\}$
        Final graph embedding $\mathbf{g}^{(L)}$
---------------------------------------------------------------
// Iterative message-passing
for $l \in \{0, \ldots, L-1\}$ do
    $\mathbf{e}_{nm}^{(l+1)} \leftarrow \mathrm{Update}_{\mathrm{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right)$
    $\mathbf{z}_n^{(l+1)} \leftarrow \mathrm{Aggregate}_{\mathrm{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right)$
    $\mathbf{h}_n^{(l+1)} \leftarrow \mathrm{Update}_{\mathrm{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right)$
    $\mathbf{g}^{(l+1)} \leftarrow \mathrm{Update}_{\mathrm{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)}\}, \{\mathbf{e}_{nm}^{(l+1)}\}\right)$
end for
return $\{\mathbf{h}_n^{(L)}\}$, $\{\mathbf{e}_{nm}^{(L)}\}$, $\mathbf{g}^{(L)}$
```

13.3.6 Geometric deep learning

We have seen how permutation symmetry is a key consideration when designing deep learning models for graph-structured data. It acts as a form of inductive bias, dramatically reducing the data requirements while improving predictive performance. In applications of graph neural networks associated with spatial properties, such as graphics meshes, fluid flow simulations, or molecular structures, there are additional equivariance and invariance properties that can be built into the network architecture.

Consider the task of predicting the properties of a molecule, for example when exploring the space of candidate drugs. The molecule can be represented as a list of atoms of given types (carbon, hydrogen, nitrogen, etc.) along with the spatial coordinates of each atom expressed as a three-dimensional column vector. We can introduce an associated embedding vector for each atom n at each layer l, denoted by $\mathbf{r}_n^{(l)}$, and these vectors can be initialized with the known atom coordinates. However, the values for the elements of these vectors depends on the arbitrary choice of coordinate system, whereas the properties of the molecule do not. For example, the solubility of the molecule is unchanged if it is rotated in space or translated to a new position relative to the origin of the coordinate system, or if the coordinate system itself is reflected to give the mirror image version of the molecule. The molecular properties should therefore be invariant under such transformations.

By making careful choices of the functional forms for the update and aggregation operations (Satorras, Hoogeboom, and Welling, 2021), the new embeddings $\mathbf{r}_n^{(l)}$ can be incorporated into the graph neural network update equations (13.29) to (13.31) to achieve the required symmetry properties:

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \|\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\|^2\right) \tag{13.38}$$

$$\mathbf{r}_n^{(l+1)} = \mathbf{r}_n^{(l)} + C \sum_{(n,m)\in\mathcal{E}} \left(\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\right) \phi\left(\mathbf{e}_{nm}^{(l+1)}\right) \tag{13.39}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right) \tag{13.40}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}\right) \tag{13.41}$$

Note that the quantity $\|\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\|^2$ represents the squared distance between the coordinates $\mathbf{r}_n^{(l)}$ and $\mathbf{r}_m^{(l)}$, and this does not depend on translations, rotations, or reflections. Also, the coordinates $\mathbf{r}_n^{(l)}$ are updated through a linear combination of the relative differences $\left(\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\right)$. Here $\phi\left(\mathbf{e}_{nm}^{(l+1)}\right)$ is a general scalar function of the edge embeddings and is represented by a neural network, and the coefficient C is typically set equal to the reciprocal of the number of terms in the sum. It follows that under such transformations, the messages in (13.38), (13.40), and (13.41) are invariant and the coordinate embeddings given by (13.39) are equivariant.

Exercise 13.10

We have seen many examples of symmetries in structured data, from translations of objects within images and the permutation of node orderings on graphs, to rotations and translations of molecules in three-dimensional space. Capturing these symmetries in the structure of a deep neural network is a powerful form of inductive bias and forms the basis of a rich field of research known as *geometric deep learning* (Bronstein *et al.*, 2017; Bronstein *et al.*, 2021).

Exercises

13.1 ($\star$) Show that the permutation $(A, B, C, D, E) \to (C, E, A, D, B)$ corresponding to the two choices of node ordering in Figure 13.2 can be expressed in the form (13.5) with a permutation matrix given by (13.1).

13.2 ($\star\star$) Show that the number of edges connected to each node of a graph is given by the corresponding diagonal element of the matrix $\mathbf{A}^2$ where $\mathbf{A}$ is the adjacency matrix.

13.3 ($\star$) Draw the graph whose adjacency matrix is given by

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{13.42}$$