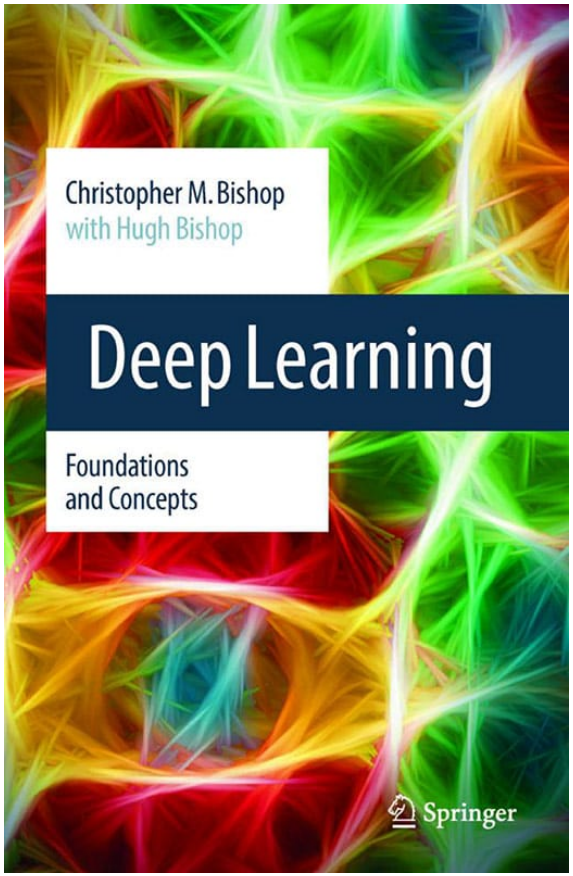
Graph Neural Networks

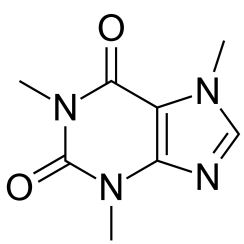
Shuiwang Ji, Xiner Li, Shurui Gui
Department of Computer Science & Engineering
Texas A&M University

These slides are based on Chapter 13 of
Deep Learning: Foundations and Concepts

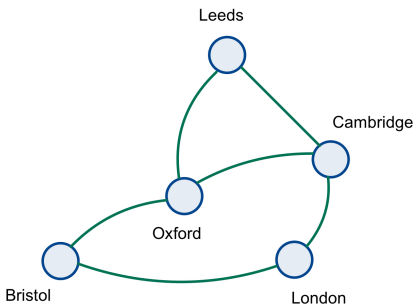


Graphs

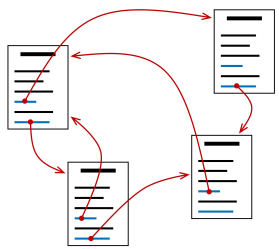
- A graph consists of a set of objects, known as nodes, connected by edges
- Both the nodes and the edges can have data associated with them
- For example, in a molecule the nodes and edges are associated with discrete variables corresponding to the types of atom (carbon, nitrogen, hydrogen, etc.) and the types of bonds (single bond, double bond, etc.)
- An image is a special instance of graph-structured data in which the nodes are the pixels and the edges describe which pixels are adjacent
- 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



(a)



(b)

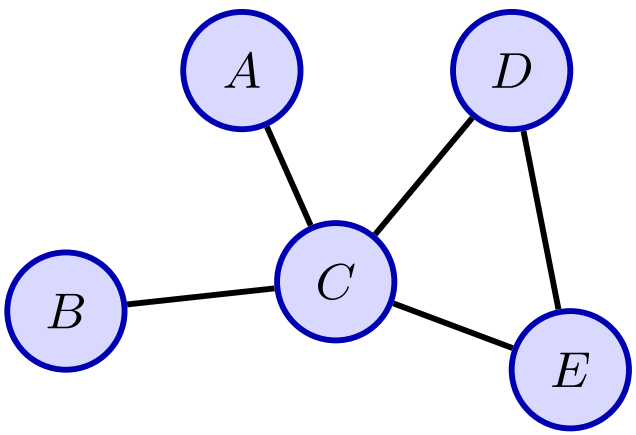

(c)

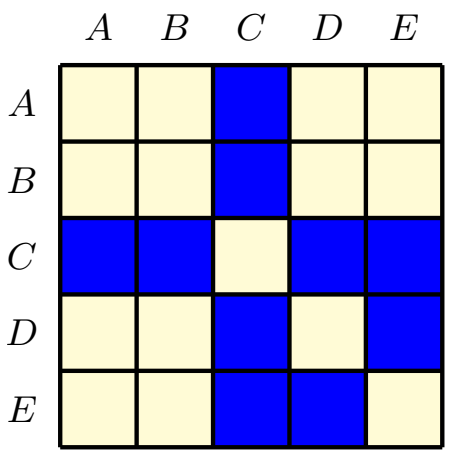
Graphs

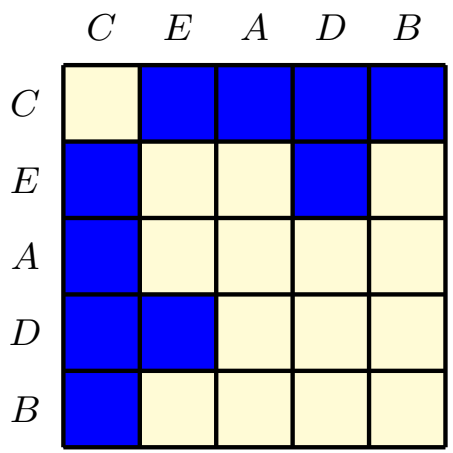
- A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of nodes $\mathcal{V}$, along with a set of edges $\mathcal{E}$
- We index the nodes by $n = 1, \ldots N$, and we write the edge from node n to node m as (n, m)
- If two nodes are linked by an edge they are called neighbours, and the set of all neighbours of node n is denoted by $\mathcal{N}(n)$
- For each node n we can represent the node variables as a D-dimensional 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 $\mathbf{x}_n^T$

Adjacency matrix

- We first choose an ordering for the nodes. If there are N nodes in the graph, we can index them using $n = 1, \ldots, N$
- A key consideration is to ensure either equivariance or invariance with respect to a reordering of the nodes in the graph
- 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







Permutation equivariance

- We can express node label permutation mathematically by introducing the concept of a permutation matrix $\mathbf{P}$
- Consider the permutation from $(A, B, C, D, E) \rightarrow (C, E, A, D, B)$, the corresponding $\mathbf{P}$ is

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

- When we reorder the nodes, the effect on the node data matrix $\mathbf{X}$ is to permute the rows by pre-multiplication of $\mathbf{P}$ as

$$\widetilde{\mathbf{X}} = \mathbf{PX}$$

- For the adjacency matrix, both the rows and the columns become permuted as

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

Permutation equivariance

- When applying deep learning to graph-structured data, we need to 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

$$y(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) = y(\mathbf{X}, \mathbf{A}) \quad \text{Invariance}$$

- Node predictions: If we reorder the node labelling then the corresponding predictions should show the same reordering
- In other words, node predictions should be equivariant with respect to node label reordering as

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

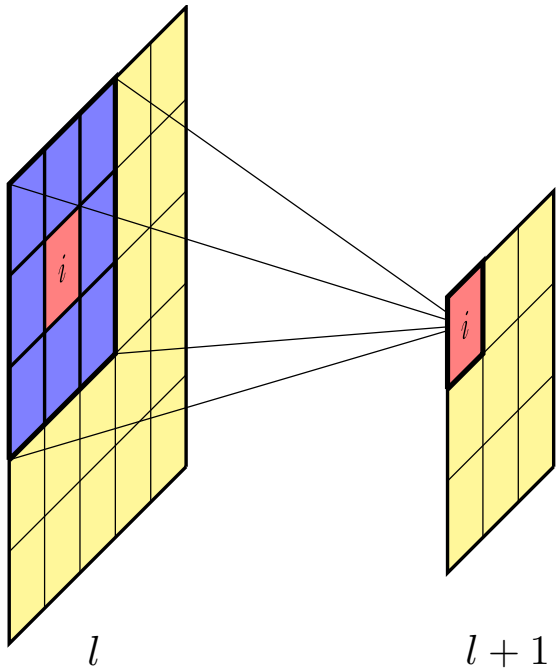
where $\mathbf{y}(\cdot, \cdot)$ is a vector of network outputs, with one element per node

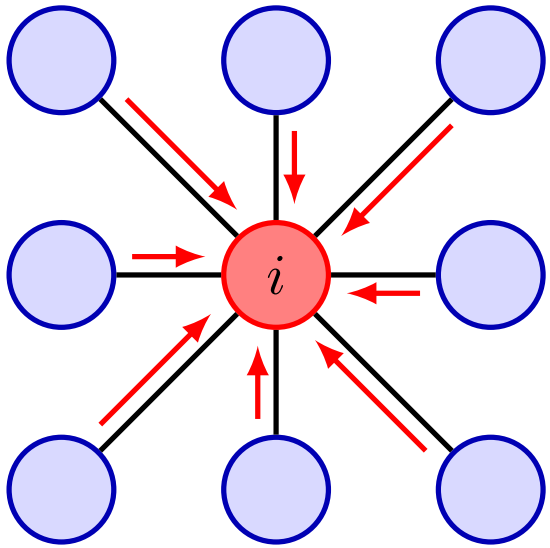
Convolutional filters

- An image can be viewed as a specific instance of graph
- Consider a convolutional layer using 3×3 filters as

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

  $f(\cdot)$ is a differentiable nonlinear function such as ReLU, and the sum over j is taken over all nine pixels in a small patch in layer l
- It is not equivariant under reordering of the nodes





Convolutional filters

- Assume that a single weight parameter w_{neigh} is shared across the neighbours so that

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

  where node i has its own weight parameter w_{self}
- The information from the neighbouring nodes is aggregated through a simple summation, and this is clearly invariant to any permutation
- The operation is applied 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.

Graph convolutional networks

- Define a nonlinear transformation that maps the embeddings $\mathbf{h}_n^{(l)}$ in layer l into corresponding embeddings in layer $l+1$
- Aggregation: 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

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

- Update: the aggregated information is combined with local information from the node itself and used to calculate a revised embedding vector for that node

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

- The node embeddings are typically initialized using observed node data so that $\mathbf{h}_n^{(0)} = \mathbf{x}_n$
- This framework is called a message-passing neural network

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
- The simplest form is summation:

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

- Another variation is the average

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

- Takes account of the number of neighbours for each of the neighbouring nodes:

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

Aggregation operators with learnable parameters

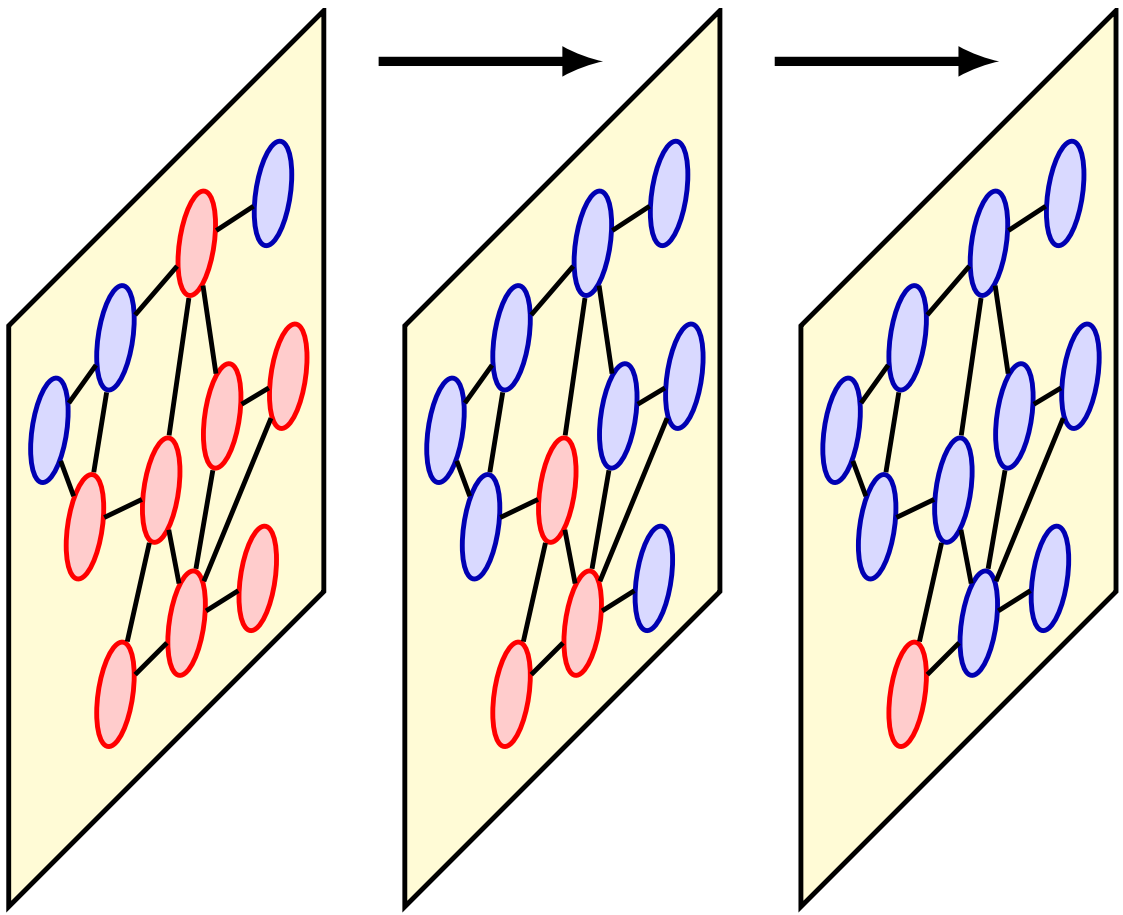
- We can introduce learnable parameters by transforming the embedding vectors from neighbouring nodes using MLP_ϕ, before combining their outputs
- So long as the network has a structure and parameter values that are shared across nodes then this aggregation operator again be permutation invariant
- We can also transform the combined vector with another neural network $\mathrm{MLP}_{\boldsymbol{\theta}}$:

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

  in which MLP_ϕ and MLP $_\theta$ are shared across layer l
- Due to the flexibility of MLPs, the transformation represents a universal approximator for any permutation-invariant function that maps a set of embeddings to a single embedding

Effective receptive field

- As information is processed through successive layers, effective receptive field is increased



Update operators

- A simple form for update would be

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

- If we choose sum as the aggregation function and if we share the same weight matrix as $\mathbf{W} = \mathbf{W}_{\text{self}} = \mathbf{W}_{\text{neigh}}$, we obtain

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

- In matrix form:

$$\boxed{\mathbf{H}^{(l+1)} = \mathbf{F}\left(\mathbf{A}\mathbf{H}^{(l)}\mathbf{W}^T + \mathbf{b}\right)}$$

where $\left(\mathbf{h}_n^{(l+1)}\right)^T$ is the n-th row of $\mathbf{H}^{(l+1)}$, and $\mathbf{A}$ is the adjacency matrix

- Note $\mathbf{W}$ and $\mathbf{b}$ are layer-specific, but layer indices are omitted for simplicity

Graph neural networks

- Overall, we can represent a GNN 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 $\mathbf{h}_n^{\mathrm{T}}$, we can write the successive transformations as

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

- Under a node reordering, the transformation is equivariant:

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

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

Node classification

- A GNN can be viewed as a series of layers each of which transforms a set of node-embedding vectors $\left\{\mathbf{h}_n^{(l)}\right\}$ into a new set $\left\{\mathbf{h}_n^{(l+1)}\right\}$
- For node classification, we use an output layer, sometimes called a readout layer as

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

  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

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

  where $\{t_{ni}\}$ are one-hot encoding of target values
- Because the weight vectors $\{\mathbf{w}_i\}$ are shared across the output nodes, the outputs are equivariant to node permutation and hence the loss function is invariant

Node classification

Three types of nodes:

1. Training nodes $\mathcal{V}_{\text{train}}$: labelled and included in the message-passing operations and are also used to compute the loss function in training
2. Transductive nodes $\mathcal{V}_{\text{trans}}$: unlabelled and do not contribute to the loss function in 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. Inductive nodes $\mathcal{V}_{\text{induct}}$: 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

- Inductive (supervised) learning: no transductive nodes, and hence the test nodes (and their associated edges) are not available during the training phase
- Transductive (semi-supervised) learning: There are transductive nodes

Edge classification

- 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:

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

Graph classification

- The goal is to predict the properties of new graphs given a training set of labelled graphs $\mathcal{G}_1, \ldots, \mathcal{G}_N$
- Combine the final-layer embedding vectors in a way that does not depend on the arbitrary node ordering
- The simplest approach is to take the sum of the node-embedding vectors:

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

- $\mathbf{f}$ may contain learnable parameters such as a linear transformation or a neural network
- Graph-level predictions correspond to an inductive task since there must be separate sets of graphs for training and for inference

Graph attention networks

- The incoming messages are weighted by attention coefficients A_{nm} as

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

  where the attention coefficients satisfy

$$A_{nm} \geqslant 0$$
$$\sum_{m \in \mathcal{N}(n)} A_{nm} = 1$$

- Some neighbouring nodes will be more important than others in determining the best update in a way that depends on the data itself

Graph attention networks

There are multiple ways to construct the attention coefficients:

- A bilinear form:

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

  where $\mathbf{W}$ is a $D \times D$ matrix of learnable parameters

- Use a neural network to combine the embedding vectors from the nodes at each end of the edge:

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

  where the MLP is shared across all the nodes and has a single continuous output variable whose value is invariant if the input vectors are exchanged

Edge embeddings

- Some networks have data associated with the edges
- In addition to the node embeddings given by $\mathbf{h}_n^{(l)}$, we introduce edge embeddings $\mathbf{e}_{nm}^{(l)}$
- We can define general message-passing equations as

$$\begin{aligned}
\mathbf{e}_{nm}^{(l+1)} &= \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}\right) \\
\mathbf{z}_n^{(l+1)} &= \text{Aggregate}_{\text{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right) \\
\mathbf{h}_n^{(l+1)} &= \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}\right)
\end{aligned}$$

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

General message-passing equations

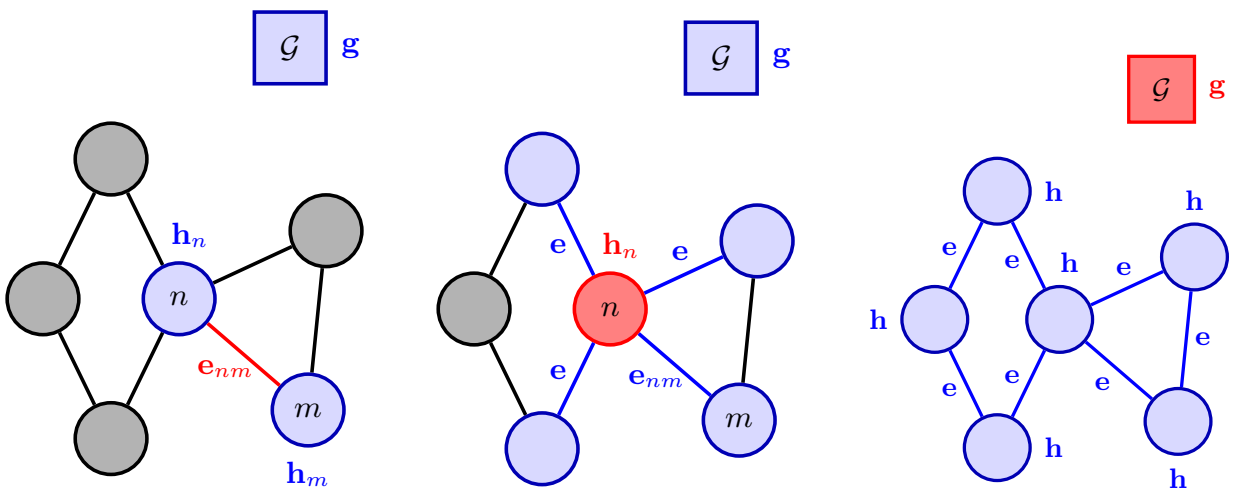
- We can also maintain and update an embedding vector $\mathbf{g}^{(l)}$ that relates to the graph as a whole
- We can define general message-passing equations as

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

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

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

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



THANKS!