

# Deep Learning - Foundations and Concepts

## Chapter 13. Graph Neural Networks

nonlineark@github

March 24, 2025

# Outline

- 1 Machine Learning on Graphs
- 2 Neural Message-Passing
- 3 General Graph Networks

# Machine learning on graphs

There are many kinds of applications that we might wish to address using graph-structured data:

- Node prediction: Classify documents according to their topic based on the hyperlinks and citations between the documents.
- Edge prediction (graph completion): Knowing some of the interactions in a protein network and predict the presence of any additional ones.
- Graph prediction: Predict whether a particular molecule is soluble in water.

# Graph 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  $n = 1, \dots, N$ , 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 neighbors, and the set of all neighbors of node  $n$  is denoted by  $\mathcal{N}(n)$ .
- For each node  $n$  we can represent the corresponding node variables as a  $D$ -dimensional column vector  $x_n$  and we can group these into a data matrix  $X$  of dimensionality  $N \times D$ .

# Adjacency matrix

A convenient way to specify the edges in a graph is to use an adjacency matrix denoted by  $A$ :

- To define the adjacency matrix we first have to choose an ordering for the nodes.
- 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.
- We could consider using adjacency matrix 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.
- The node ordering invariance should be treated as an inductive bias when constructing a network architecture.

# Permutation equivariance

- A permutation  $\pi$  is a bijection from  $\{1, \dots, N\}$  to itself.
- Let  $\{e_1, \dots, e_N\}$  be the standard basis of  $\mathbb{R}^N$ , given a permutation  $\pi$ , the corresponding permutation matrix  $P$  is defined as:

$$P = \begin{pmatrix} e_{\pi(1)} & \dots & e_{\pi(N)} \end{pmatrix}^T$$

- When we reorder the labelling on the nodes of a graph:
  - The data matrix  $X$  is changed to  $\tilde{X} = P^T X$ .
  - The adjacency matrix  $A$  is changed to  $\tilde{A} = P^T A P$ .
  - Any global property of the graph does not depend on node label reordering:  $y(\tilde{X}, \tilde{A}) = y(X, A)$ .
  - Node predictions should be equivariant with respect to node label reordering:  $y(\tilde{X}, \tilde{A}) = P^T y(X, A)$ .

# Convolutional filters

Consider a convolutional layer using  $3 \times 3$  filters. The computation performed by a single filter at a single pixel in layer  $l + 1$  can be expressed as:

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

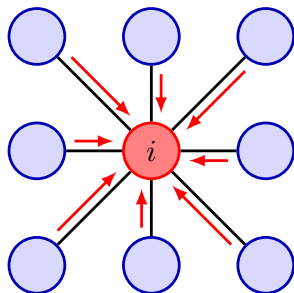
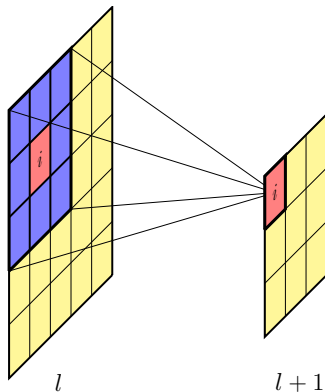
This is not equivariant under reordering of the nodes in layer  $l$ . However, we can achieve equivariance with some simple modifications as follows:

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

where a single weight parameter  $w_{\text{neigh}}$  is shared across the neighbors, and node  $i$  has its own weight parameter  $w_{\text{self}}$ .

# Convolutional filters

**Figure:** A convolutional filter for images can be represented as a graph-structured computation





# Graph convolutional networks

We can view each layer of processing as having two successive stages:

- Aggregation stage: For each node  $n$ , messages are passed to that node from its neighbors and combined to form a new vector  $z_n^{(l)}$  in a way that is permutation invariant:  
$$z_n^{(l)} = \text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}).$$
- Update stage: The aggregated information from neighboring nodes is combined with local information from the node itself and used to calculate a revised embedding vector for that node:  
$$h_n^{(l+1)} = \text{Update}(h_n^{(l)}, z_n^{(l)}).$$

# Graph convolutional networks

---

## Algorithm 1: Simple message-passing neural network

---

**for**  $l \leftarrow 0$  **to**  $L - 1$  **do**

$z_n^{(l)} \leftarrow \text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\});$   
     $h_n^{(l+1)} \leftarrow \text{Update}(h_n^{(l)}, z_n^{(l)});$

**end**

**return**  $\{h_n^{(L)}\};$

---

# Aggregation operators

Some simple aggregation functions:

$$\text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) = \sum_{m \in \mathcal{N}(n)} h_m^{(l)}$$

$$\text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) = \frac{1}{|\mathcal{N}(n)|} \sum_{m \in \mathcal{N}(n)} h_m^{(l)}$$

$$\text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) = \sum_{m \in \mathcal{N}(n)} \frac{h_m^{(l)}}{\sqrt{|\mathcal{N}(n)| |\mathcal{N}(m)|}}$$

$$\text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) = \max(\{h_m^{(l)} : m \in \mathcal{N}(n)\})$$

# Aggregation operators

We can introduce learnable parameters by:

- First transforming each of the embedding vectors from neighboring nodes using a multilayer neural network, denoted by  $\text{MLP}_\phi$ .
- Then transforming the combined vector with another neural network  $\text{MLP}_\theta$ .

to give an overall aggregation operator:

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

in which  $\text{MLP}_\phi$  and  $\text{MLP}_\theta$  are shared across layer  $l$ .

# Update operators

A simple form for the update operator would be:

$$\text{Update}(h_n^{(l)}, z_n^{(l)}) = f(W_{\text{self}}h_n^{(l)} + W_{\text{neigh}}z_n^{(l)} + b)$$

If we choose a simple summation as the aggregation function and if we also share the same weight matrix between nodes and their neighbors, we obtain a particularly simple form of the update operator:

$$h_n^{(l+1)} = \text{Update}(h_n^{(l)}, z_n^{(l)}) = f(W^{(l+1)} \sum_{m \in \mathcal{N}(n), n} h_m^{(l)} + b)$$

# Node classification

We need to define a cost function for training:

- For node classification over  $C$  classes, we can use  $\text{softmax}(H_n^{(L)} W^{(o)})$  to output the logits, where  $W^{(o)}$  is a learnable  $D \times C$  matrix. The loss function is defined as the cross-entropy loss across all nodes and all classes.
- 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.

# Node classification

We can distinguish between three types of nodes as follows:

	$\mathcal{V}_{\text{train}}$	$\mathcal{V}_{\text{trans}}$	$\mathcal{V}_{\text{induct}}$
Labelled	Yes	No	No
Included in the message-passing operations of the graph neural network	Yes	Yes	No
Used to compute the loss function used for training	Yes	No	No

If there are no transductive nodes, then the training is generally referred to as inductive learning. However, if there are transductive nodes then it is called transductive learning.

# 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. The probability  $p(n, m)$  for the presence of an edge between nodes  $n$  and  $m$  can be defined as:

$$p(n, m) = \sigma(h_n^T h_m)$$



# 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  $\mathcal{G}_1, \dots, \mathcal{G}_N$ . This requires that we combine all the final-layer embedding vectors in a way that does not depend on the arbitrary node ordering:

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

where the function  $f$  may contain learnable parameters. Other invariant aggregation functions can be used such as averages or element-wise minimum or maximum.

# Graph attention networks

Graph attention networks can capture an inductive bias that says some neighboring nodes will be more important than others in determining the best update in a way that depends on the data itself:

$$z_n^{(l)} = \text{Aggregate}(\{h_m^{(l)} : m \in \mathcal{N}(n)\}) = \sum_{m \in \mathcal{N}(n)} A_{nm} h_m^{(l)}$$

where the attention coefficients  $A_{nm}$  should satisfy:

$$\begin{aligned} A_{nm} &\geq 0 \\ \sum_{m \in \mathcal{N}(n)} A_{nm} &= 1 \end{aligned}$$

# Graph attention networks

Some ways to construct the attention coefficients:

$$A_{nm} = \frac{\exp(h_n^T W h_m)}{\sum_{m' \in \mathcal{N}(n)} \exp(h_n^T W h_{m'})}$$

where  $W$  is a  $D \times D$  matrix of learnable parameters. Or:

$$A_{nm} = \frac{\exp(\text{MLP}(h_n, h_m))}{\sum_{m' \in \mathcal{N}(n)} \text{MLP}(h_n, h_{m'})}$$

where the MLP has a single continuous output variable whose value is invariant if the input vectors are exchanged.

# Edge embeddings

Some networks also have data associated with the edges. In addition to the node embeddings given by  $h_n^{(l)}$ , we therefore introduce edge embeddings  $e_{nm}^{(l)}$ . We can then define general message-passing equations in the form:

$$e_{nm}^{(l+1)} = \text{Update}_{\text{edge}}(e_{nm}^{(l)}, h_n^{(l)}, h_m^{(l)})$$

$$z_n^{(l+1)} = \text{Aggregate}_{\text{node}}(\{e_{nm}^{(l+1)} : m \in \mathcal{N}(n)\})$$

$$h_n^{(l+1)} = \text{Update}_{\text{node}}(h_n^{(l)}, z_n^{(l+1)})$$

# Graph embeddings

In addition to node and edge embeddings we can also maintain and update an embedding vector  $g^{(l)}$  that relates to the graph as a whole. We can define general message-passing equations:

$$e_{nm}^{(l+1)} = \text{Update}_{\text{edge}}(e_{nm}^{(l)}, h_n^{(l)}, h_m^{(l)}, g^{(l)})$$

$$z_n^{(l+1)} = \text{Aggregate}_{\text{node}}(\{e_{nm}^{(l+1)} : m \in \mathcal{N}(n)\})$$

$$h_n^{(l+1)} = \text{Update}_{\text{node}}(h_n^{(l)}, z_n^{(l+1)}, g^{(l)})$$

$$g^{(l+1)} = \text{Update}_{\text{graph}}(g^{(l)}, \{h_n^{(l+1)} : n \in \mathcal{V}\}, \{e_{nm}^{(l+1)} : (n, m) \in \mathcal{E}\})$$

# Graph embeddings

---

**Algorithm 2:** Graph neural network with node, edge, and graph embeddings

---

**for**  $l \leftarrow 0$  **to**  $L - 1$  **do**

$e_{nm}^{(l+1)} \leftarrow \text{Update}_{\text{edge}}(e_{nm}^{(l)}, h_n^{(l)}, h_m^{(l)}, g^{(l)});$

$z_n^{(l+1)} \leftarrow \text{Aggregate}_{\text{node}}(\{e_{nm}^{(l+1)} : m \in \mathcal{N}(n)\});$

$h_n^{(l+1)} \leftarrow \text{Update}_{\text{node}}(h_n^{(l)}, z_n^{(l+1)}, g^{(l)});$

$g^{(l+1)} \leftarrow \text{Update}_{\text{graph}}(g^{(l)}, \{h_n^{(l+1)} : n \in \mathcal{V}\}, \{e_{nm}^{(l+1)} : (n, m) \in \mathcal{E}\});$

**end**

**return**  $\{h_n^{(L)}\}, \{e_{nm}^{(L)}\}, g^{(L)};$

---

# 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. One way to help alleviate this issue is to introduce residual connections:

$$h_n^{(l+1)} = \text{Update}_{\text{node}}(h_n^{(l)}, z_n^{(l+1)}, g^{(l)}) + h_n^{(l)}$$

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:

$$y_n = f(h_n^{(1)} \oplus \dots \oplus h_n^{(L)})$$

where  $a \oplus b$  denotes the concatenation or max pooling of vectors  $a$  and  $b$ .

# Regularization

- Standard techniques: For example, adding the sum-of-squares of the parameter values to the loss function.
- Parameter sharing: Weights and biases can also be shared across layers to reduce the number of independent parameters.
- Dropout: Omitting random subsets of the graph nodes or edges during training.



# Geometric deep learning

In applications of graph neural networks associated with spatial properties, there are additional equivariance and invariance properties that can be built into the network architecture. For example, the task of predicting the properties of a molecule:

- The molecule can be represented as a list of atoms of given types along with the spatial coordinates of each atom denoted by  $r_n^{(l)}$ .
- The values for  $r_n^{(l)}$  depends on the arbitrary choice of coordinate system, whereas the properties of the molecule do not:
  - Translation.
  - Rotation.
  - Reflection.

# Geometric deep learning

The new embeddings  $r_n^{(l)}$  can be incorporated into the graph neural network update equations to achieve the required symmetry properties:

$$e_{nm}^{(l+1)} = \text{Update}_{\text{edge}}(e_{nm}^{(l)}, h_n^{(l)}, h_m^{(l)}, ||r_n^{(l)} - r_m^{(l)}||^2)$$

$$r_n^{(l+1)} = r_n^{(l)} + C \sum_{m \in \mathcal{N}(n)} (r_n^{(l)} - r_m^{(l)}) \phi(e_{nm}^{(l+1)})$$

$$z_n^{(l+1)} = \text{Aggregate}_{\text{node}}(\{e_{nm}^{(l+1)} : m \in \mathcal{N}(n)\})$$

$$h_n^{(l+1)} = \text{Update}_{\text{node}}(h_n^{(l)}, z_n^{(l+1)})$$

where  $\phi$  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.