Chapter 5

Graph Neural Networks

5.1 Introduction

The main task we deal with in this chapter is supervised learning from graph data at the *instance* level, whereas in the previous chapter we dealt with unsupervised learning from graph data at the node level. What it means is that now the observed data is $\mathcal{G}_1, \ldots, \mathcal{G}_M$, where $\mathcal{G}_i$ is the i-th graph of the collection and N_i is its number of nodes (i.e. the *order*), possibly varying from one graph to another. The observed graphs are assumed to be *attributed*, meaning that a feature vector in $\mathbb{R}^D$ is associated with each node of each graph. So, $\mathcal{G}_i$ is represented by the pair (A_i, X_i), where $A_i \in \{0,1\}^{N_i \times N_i}$ is the adjacency matrix and $X_i \in \mathbb{R}^{N_i \times D}$ is the feature matrix: its j-th row is the feature vector associated with the j-th node of $\mathcal{G}_i$.

Remark. *Several standard graph data sets in machine learning consist in sequences of attributed graphs. However, note that even when a graph is unattributed it is always possible to turn it into an attributed one, for instance by equipping each node with its spectral* ***positional encoding*** *(Section 4.A).*

In this chapter, we introduce graph neural networks (GNNs), namely a class of deep architectures for graph-structured data. The main intuition is: GNNs manipulate the input graph (A_i, X_i) in such a way to cleverly "convert" it into a vector in $\mathbb{R}^D$ (or another dimension) that can then be treated in a standard way in order to solve the supervised classification problem. Note, however that GNNs are very powerful tools that can be used for tasks other than standard graph classification (e.g. link prediction, classification of nodes

etc.). For an in depth treatment of GNNs the reader is referred to C. M. Bishop and H. Bishop, 2023, Ch.13 and K. Xu et al., 2018.

Since GNN architectures rely on and generalize standard feed-forward neural networks we start by quickly reviewing a basic neural net: the multilayer perceptron.

5.2 Feed forward neural networks

A standard task in supervised machine learning can be summarized shortly as follows. Imagine we observe $x_1, \dots, x_N$ feature vectors, living in $\mathbb{R}^D$ together with their labels $y_1, \dots, y_N$. Our aim is to learn a function f_θ linking x_i to y_i, for all $i \in \{1, \dots, N\}$ in such a way to make predictions $\hat{y}_i := f_\theta(x_i)$ being "as faithful as possible" to the observed labels $y_1, \dots, y_N$. More formally y_i can be either a real/integer vector (regression) or a categorical variable in 1-to-K binary encoding (classification, K classes). Depending on the nature of y_i, one defines a loss function $L(y_i, \hat{y}_i)$ (e.g. mean squared error or cross-entropy) and seek to minimize the (empirical) expected loss with respect to θ

$$\min_\theta \left(\frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i) \right). \tag{5.1}$$

That is, we seek to estimate (or learn) a value of θ allowing us to minimize the average prediction error on the train data set.

Recall. *Although solving the above minimization problem is something that we need in order to "learn" a good predictor/classifier, that minimization only is an intermediate objective. The final objective is to be able to correctly predict/classify any new test data point x^*, via $f_\theta(x^*)$. In other terms we seek to avoid both* overfitting *and* underfitting.

In order to avoid overfitting, one might modify the above objective by adding some regularisation terms, such as ℓ_2 or ℓ_1 penalties, for instance, and/or monitor the loss on a validation data set in such a way to early stop the optimisation if needed (useful when θ is optimised numerically, e.g. via stochastic gradient descent).

Underfitting, is mainly related to the way f_θ is specified. Indeed we already encountered in Chapter 1 two declinations of the general framework just mentioned:

i) the linear regression model, where we assumed $f_\theta(\cdot) := \langle \theta, \cdot \rangle$, with θ being in that case an unknown vector in $\mathbb{R}^D$ and the loss function was the mean squared error $L(y_i, \hat{y}_i) := ||y_i - \hat{y}_i||_2^2$;

ii) the logistic regression, which despite its name indeed is a *classifier*, with $f_\theta(\cdot)$ defined as in the linear regression model but with a different loss

$$L(y_i, \hat{y}_i) = \log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i,$$

which is known as binary cross entropy, however it is nothing but the negative log-likelihood that we encountered in Eq. (1.9).

As it can be seen both the linear and the logistic regression assume that f_θ is a *linear* function of the covariates. Imposing linearity has some advantages (e.g. simplicity, explainability, sometimes closed formulas) but might be two simplistic and lead to some underfitting. One way to go beyond the linearity assumption is to adopt **neural networks** in order to parametrise f_θ. The simplest (not trivial) neural net one might think of is the multilayer perceptron (**MLP**)

$$f_\theta(x_i) := \sigma_2 \left(W_2 \sigma_1 (W_1 x_i + b_1) + b_2 \right), \tag{5.2}$$

where $W_1 \in \mathbb{R}^{D \times P}$, $b_1 \in \mathbb{R}^P$ with P denoting denoting here the number of *neurons* or hidden layer size and $W_2 \in \mathbb{R}^{P \times K}$, $b_2 \in \mathbb{R}^K$. So the learnable parameters are $\theta := \{W_1, W_2, b_1, b_2\}$ and σ_i denotes here *non*-linear activation functions (e.g. ReLU[1], tanh[2], softmax[3]) which act element-wise. Here, K denotes the output size: in case of linear regression or binary classification $K = 1$ and $\sigma_2 = Id$. For multiclass classification, $K > 1$ and σ_2 is a softmax function.

Thanks to an additional notation we can reformulate the definition of f_θ recursively:

$$\begin{aligned} h_i^{(1)} &:= \sigma_1 \left(W_1 x_i + b_1 \right) \\ f_\theta(x_i) &:= \sigma_2 \left(W_2 h_i^{(1)} + b_2 \right). \end{aligned}$$

This definition puts in light two important things: first an MLP is noting but a composition of linear layers interleaved by a non-linear activation function.

[1] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[2] https://reference.wolfram.com/language/ref/Tanh.html
[3] https://en.wikipedia.org/wiki/Softmax_function

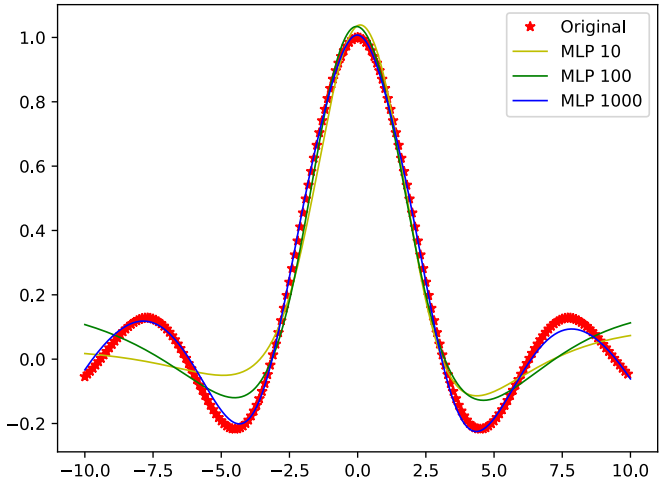

Figure 5.1: Approximation of the function $f(x) = \frac{\sin(x)}{x}$ via a MLP with one hidden layer with variable number of nodes $P = 10, 100, 1000$.

Second, the above MLP has three layers, namely input (x_i), intermediate or hidden $(h_i^{(1)})$ and output $(f_\theta(x_i))$. However we can define neural nets with an arbitrary number of hidden layers L

$$
\begin{aligned}
h_i^{(1)} &:= \sigma_1(W_1 x_i + b_1) \\
h_i^{(2)} &:= \sigma_2\left(W_2 h_i^{(1)} + b_2\right) \\
\cdots &:= \ldots \\
h_i^{(L)} &:= \sigma_L\left(W_L h_i^{(L-1)} + b_L\right) \\
f_\theta(x_i) &:= \sigma_{L+1}\left(W_{L+1} h_i^{(L)} + b_{L+1}\right).
\end{aligned}
$$

Even without needing more than one hidden layer, MLP equipped with any non polynomial activation function can approximate any continuous function between two Euclidean spaces to any arbitrary precision, in the limit $P \to \infty$ (Universal Approximation Theorem[4]). Being a formal statement and proof of this claim behind the scope of this course, we play with it empirically in Figure 5.1. In red (stars) one sees the function $\frac{\sin(x)}{x}$ evaluated on a regular

[4] https://en.wikipedia.org/wiki/Universal_approximation_theorem

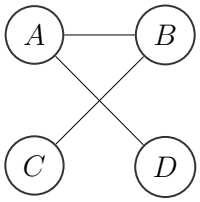

Figure 5.2: An undirected graph with four nodes.

grid between -10 and 10. Three MLPs with one hidden layer were trained in order to approximate that function (MSE loss) with number of neurons in the hidden layer (P) varying from 10 to 1000. More details will be discussed in a dedicated Python notebook.

In order to conclude this short section on supervised learning and feed forward neural nets, we recall that, after plugging Eq. (5.2) into the minimisation problem (5.1), one has to solve a non convex optimisation problem which is usually attacked by means of stochastic gradient descent (Bottou, 2010), where the gradient w.r.t. θ can be computed via automatic differentiation (Baydin et al., 2018).

5.3 Inductive bias and graph data

There are multiple reasons why standard MLPs cannot be used for supervised classification when the observed (graph) data is $(A_1, X_1), \ldots, (A_M, X_M)$, together with labels $y_1, \ldots, y_N$. First even if one wanted to concatenate (A_i, X_i) in a single matrix with N_i rows and $N_i + D$ columns and then flattened that matrix in order to obtain a standard feature vector, the M training feature vectors would have different size, since each graph is allowed to have a different order. Second, although the order of the input graph was the same, the above strategy would be a poor one. In order to figure out why, look at the simple (not attributed) graph in Figure 5.2. We might "read" that graph in the alphabetical order, (i.e. $\{A, B, C, D\}$) and the corresponding adjacency matrix would read

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

If instead we read it the clockwise order, (i.e. $\{C, A, B, D\}$), its adjacency matrix now reads

$$\bar{A} := \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

however the graph is still the same one! Vectorising both the above matrices would provide a feed-forward neural net with two different inputs, since $||A - \dot{A}||_F \neq 0$[5].

We can link the two matrices above via the following permutation matrix

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

since $\dot{A} = PAP^T$ and if we enrich the graphs with feature matrices X and $\dot{X}$, we have $\dot{X} = PX$. So two fundamental properties that we want that our architecture to exhibit are

1. permutation **invariance**: we want the final output of our network f_θ to be such that $f_\theta(A, X) = f_\theta(PAP^T, PX)$, for any permutation matrix P;

2. permutation **equivariance**: any time our network updates the node representation, for instance in an hidden layer ℓ_θ, we want $\ell_\theta(PAP^T, PX) = P l_\theta(A, X)$, for any permutation matrix P.

Additionally, we want that our network is able to handle input data of different order and that each layer is a differentiable (non-linear) function of some learnable parameters.

5.4 Neural message passing

Roughly speaking, a graph neural network is a sequence of L layers that successively process node embeddings. In more detail, at each layer l we have

[5] $||\cdot||_F$ is the Frobenious norm

a matrix $H^{(l)} \in \mathbb{R}^{N \times D}$, with N being the order of the input graph and D assumed here to be the same as the features dimensionality for simplicity. The j-th row of $H^{(l)}$, say $H_j^{(l)}$ is the latent representation of the j-th node of the graph at layer l and is obtained as

$$H^{(l)} := \ell_\theta(H^{(l-1)}, A), \tag{5.3}$$

where θ is a set of learnable parameters which usually is **shared** among layers and $H^{(0)} = X$. We need to set up ℓ_θ is such away it is permutation equivariant and we do it by decomposing it into two operators: AGGREGATE and UPDATE.

Aggregation. This is the heart of GNNs. An intermediate quantity is introduced

$$z_j^{(l)} := \textsc{Aggreate}(\{H_m^{(l)} : m \in \mathcal{N}(j)\}), \tag{5.4}$$

where $\mathcal{N}(j)$ denotes the neighbourhood of node j and AGGREGATE is a differentiable map *independent* of the ordering of the nodes. The easiest example of aggregation is the sum

$$\textsc{Aggreate}(\{H_m^{(l)} : m \in \mathcal{N}(j)\}) := \sum_{m \in \mathcal{N}(j)} H_m^{(l)}$$

or a weighted average, such as the one

$$\textsc{Aggreate}(\{H_m^{(l)} : m \in \mathcal{N}(j)\}) := \sum_{m \in \mathcal{N}(j)} \frac{H_m^{(l)}}{\sqrt{|\mathcal{N}(j)||\mathcal{N}|(m)|}}$$

used in Graph Convolutional Networks (Kipf and Welling, 2016). The two cited examples do not even have learnable parameters. We might cite in the same category the element-wise maximum or minimum of the neighbour embedding vectors since they also allow to account for a different number of neighbours and are independent of the nodes ordering.

A (much) more involved aggregation operator might be

$$\textsc{Aggreate}(\{H_m^{(l)} : m \in \mathcal{N}(j)\}) := \mathrm{MLP}_{\eta_1}\left(\sum_{m \in \mathcal{N}(j)} \mathrm{MLP}_{\eta_2}(H_j^{(l)})\right),$$

where MLP_η denotes the multilayer perceptron that we introduced in Eq. (5.2) and following and η is the set of its learnable parameters.

Update. The embedding of the j-th node itself, namely $H_j^{(l+1)}$ is then obtained via

$$H_j^{(l+1)} := \textsc{Update}(z_j^{(l)}, H_j^{(l)}).$$

Also update operators span from very simple options, such as

$$\textsc{Update}(z_j^{(l)}, H_j^{(l)}) := \sigma\left(z_j^{(l)} W_1 + H_j^{(l)} W_2 + b\right), \tag{5.5}$$

with W_1 and W_2 learnable parameters in $\mathbb{R}^{D\times D}$, b a lernable intercept in $\mathbb{R}^D$ and $\sigma(\cdot)$ a non-linear element-wise activation, to more accurate ones, such as

$$\textsc{Update}(z_j^{(l)}, H_j^{(l)}) := \text{MLP}_{\eta^{(l)}}\left((1+\epsilon^{(l)})H_j^{(j)} + z_j^{(l)}\right),$$

adopted in the Graph Isomorphism Network (K. Xu et al., 2018), where $z_j^{(l)}$ is obtained via Eq (5.4), $\eta^{(l)}$ denotes a set of layer specific parameters and $\epsilon^{(1)}, \ldots, \epsilon^{(L)}$ is a sequence of user defined (or learned) constants.

Example. In order to fix the ideas, the simplest architecture where aggregation is performed via the simple sum over neighbours, as shown in Eq (5.4) and the update is performed as in Eq. (5.5). In that case, we have:

$$H_j^{(l+1)} = \sigma\left(z_j^{(l)} W_1 + H_j^{(l)} W_2 + b\right).$$

Given that $z_j^{(l)} = \langle A_j, H^{(l)}\rangle$, in matrix notation we have:

$$H^{(l+1)} = \sigma\left(AH^{(l)} W_1 + H^{(l)} W_2 + \mathbf{1}_D b\right),$$

where $\mathbf{1}_D$ is a column vector of length D. Comparing the above equation with Eq. (5.3) we now have a specification of ℓ_θ with $\theta = \{W_1, W_2, b\}$. It is easy to show that this layer is equivariant (**exercise**).

5.5 Global aggregation

In order we can train a classifier for graph data, the last step is to aggregate the finally representation $H^{(L)}$ that we have for each input graph in order to

obtain M vectors in dimension D that can finally be input to a last MLPs. One way of doing that is to *sum* the rows of H^L. In more details:

$$\hat{y}_j := \text{MLP}\left(\sum_{i=1}^{N_j} H_i^{(j,L)}\right)$$

when $H_i^{(j,L)}$ is the i-th row of the last convolutional layer for the j-th observed graph, whose number of nodes is (we recall) N_j. The sum inside parentheses could be replaced by other invariant aggregation function such as element-wise minimum or maximum.

In case of multiclass classification, it is assumed that a softmax function is placed on top of the last MPL, such that $\hat{y}_j$ is a probability vector. Its k-th element is the probability of affectation of the j-th input data to the k-th class. Our loss function would in that case be

$$L(\Theta) := \sum_{j=1}^{M}\sum_{k=1}^{K} y_{jk} \log \hat{y}_{jk},$$

where Θ is the set of all learnable parameters mentioned so far. The above loss can be minimised w.r.t. Θ by stochastic gradient descent.