UNIVERSITY NAME

DOCTORAL THESIS

Deep Learning on Graphs with Applications to the Life Sciences

---

Author:
John SMITH

Supervisor:
Dr. James SMITH

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Research Group Name
Department or School Name

December 15, 2020

Declaration of Authorship

I, John Smith, declare that this thesis titled, "Deep Learning on Graphs with Applications to the Life Sciences" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“*Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.*”

Dave Barry

UNIVERSITY NAME

Abstract

Faculty Name
Department or School Name

Doctor of Philosophy

Deep Learning on Graphs with Applications to the Life Sciences

by John Smith

The application of Deep Learning models to complex biological problems has recently revolutionized the field of life sciences, leading to advancements that could potentially change the quality and length of human life for the better. A major contribution to this success comes from the ability of deep neural networks to well approximate non-linear biological functions, as well as the flexibility to learn from structured biological data directly. This thesis presents two relevant applications where Deep Learning is used on biological graphs to learn complex tasks. The first concerns the prediction of dynamical properties of chemical reactions at the cellular level, represented as Petri graphs. Our contribution is a Deep Learning model that can learn the task solely relying on the structure of such graphs, which is orders of magnitude faster than running expensive simulations. The second application is about accelerating the drug design process by discovering novel drug candidates. We present a deep and generative framework in which novel graphs, corresponding to molecules with desired characteristics, can be obtained by combining chemically meaningful molecular fragments. Our work suggests that coupling the power of Deep Learning with the ability to handle structured data remains one of the preferred avenues to pursue towards solving well-known biological problems, as well as an effective methodology to tackle new ones.

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

For/Dedicated to/To my…

Chapter 1

Introduction

Part I

Preliminaries

Chapter 2

Machine Learning and Neural Networks

In this chapter, we provide an overview of Machine Learning and neural networks. We start with a short introduction to Machine Learning, laying out the foundations on how learning from data is possible, as well as more practical aspects about the selection and evaluation of machine learning models. In Section 2.4, we briefly review the core concepts about Neural Networkss (NNs), such as the procedure by which they are trained, the loss functions they optimize, and how they are regularized.

2.1 Machine Learning

Many real-world phenomena are not clearly understood, or cannot be characterized in terms of simple mathematical equations. However, one can usually collect quantitative and/or qualitative measurements about them, and observe their effects in the environment where they manifest. Machine Learning (ML) is a branch of Artificial Intelligence that studies algorithms and provides tools to *learn* these unknown processes from data. According to Mitchell, **learning** is defined as improving at some task from experience (Mitchell, 1997). To start off, let us first briefly describe the key components of a learning system in detail, introducing other useful notation, definitions and concepts along the way:

- the **experience**, in the context of ML, refers to the data which is available to the learner. Data is provided in the form of *examples*, (also called *observations* or *data points*). Each example is a set of qualitative or quantitative measurements about some phenomenon of interest;

- the **task** refers to some function f (called *target function*) that the learner needs to estimate from data. ML tasks are multiple, and of potentially very different nature. Three well-known examples of tasks are are *classification*, where f is a function that assigns a categorical *label* to a data point; *regression*, where f associates a desired numerical quantity to a given example; and *density estimation*, where f is the probability density (or mass, for the discrete case) function from which the examples are drawn from;

- the **performance** is a function which returns a quantitative measurement of how "well" the task is being learned. For example, in a classification task, one can measure improvement measuring the *accuracy* of the learner, *i.e.* the proportion of correctly classified examples out of the total number of available examples. Clearly, higher accuracies indicate that a task is being learned.

Machine Learning can be broadly divided in three main areas: supervised, unsupervised, and reinforcement learning. In **supervised learning**, the learner is given a set of input-output pairs, and the goal is to to learn the relationship between the inputs and the outputs. Classification and regression fall into the supervised paradigm. In **unsupervised learning**, data only consists of inputs, and the goal is to learn some property of the distribution that generates the data. Typical unsupervised tasks include clustering and density estimation. **Reinforcement learning** is about learning to act in an environment, where the actions performed yield rewards or penalties. This work exclusively deals with supervised and unsupervised learning.

Regardless of the learning paradigm, the central issue in ML is **generalization**, that is, the learned function should be such that it works correctly on previously unseen examples, not used during the learning process. As it turns out, generalization is possible if the learning process is carefully crafted.

2.2 Learning and Generalization

Informally, learning can be thought of as finding a "good" approximation of the target function in some space of candidate functions. The search process is often referred to as *training*. During training, several candidates are evaluated until one that best approximates the target function is found. A program that implements this learning process is called **learning algorithm**.

The training process is driven by the data. Given a task, the learning algorithm has access to a **dataset** $\mathbb{D}_n = \{z^{(i)}\}_{i=1}^{n}$, a set of n independent and identically distributed (i.i.d.) samples drawn from some data domain $\mathcal{Z}$ according to a fixed but unknown distribution $p(z)$. The nature of $\mathcal{Z}$ depends on the learning paradigm. In the supervised case, we have $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is called *input space* and $\mathcal{Y}$ *output space*. The elements $z \in \mathcal{Z}$ are pairs (x, y), where y is called *label.* The output space $\mathcal{Y}$ is tightly coupled to the task to be learned; for example, in classification tasks, $\mathcal{Y}$ is a discrete set; in regression tasks, $\mathcal{Y} = \mathbb{R}$. In unsupervised learning, $\mathcal{Z}$ is just the input space $\mathcal{X}$. For this reason, data used in unsupervised tasks is often called *unlabelled.* For the moment, we assume that the input space $\mathcal{X}$ is the set of d-dimensional real-valued vectors $\mathbb{R}^d$. We call elements $\mathbf{x} \in \mathbb{R}^d$ *feature vectors*, and their elements **features**. Given a vector $\mathbf{x}$, we use the notation $\mathbf{x}_i$ to indicate its i-th feature. Later on, we shall generalize the notion of input space to more complex spaces than just vectors, to apply ML to more complex objects.

The space of candidate functions explored by the learning algorithm is called **hypotheses space**. Formally, a hypotheses space is a set of functions $\mathcal{H}_\Omega = \{h_\omega \mid$

$\omega \in \Omega\}$ whose members are called *hypotheses*. The hypotheses space is used indirectly by the learning algorithm, which rather operates in *parameter space* Ω. The elements of the parameter space are called *parameters*, and each parameter $\omega \in \Omega$ uniquely identifies a hypotheses $h_\omega \in \mathcal{H}_\Omega$. Usually, the parameters ω correspond to a set of real-valued vectors.

Besides the dataset, the learning algorithm is also given a **loss function** $L : \Omega \times \mathcal{Z} \to \mathbb{R}_+$. The role of the loss function is to provide a non-negative value to measure the error committed in approximating the unknown function f with a hypothesis h_ω. The objective of the learning algorithm is thus to find the optimal hypotheses h_{ω^*} whose error approximating f is the lowest. This can be achieved by solving the following optimization problem:

$$h_{\omega^*} = \arg\min_{\omega \in \Omega} R(h_\omega) \overset{\text{def}}{=} \int \mathcal{L}(\omega, z)\, dz,$$

where R is called *risk functional*, or **generalization error**. However, the above objective function is intractable, since the learning algorithm only has access to the specific portion of $\mathcal{Z}$ represented by the dataset $\mathbb{D}_n$, randomly sampled from $p(z)$. Therefore, a tractable optimization problem is used instead:

$$h_{\hat{\omega}} = \arg\min_{\omega \in \Omega} R_{\mathbb{D}_n}(h_\omega) \overset{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\omega, z^{(i)}),$$

where $R_{\mathbb{D}_n}$ is called *empirical risk functional*, or **training error**, and corresponds to selecting the hypothesis whose average loss computed on the dataset is the lowest. We call the hypothesis $h_{\hat{\omega}}$ given as output by the learning algorithm a **model**. The learning principle corresponding to this optimization problem is called Empirical Risk Minimization (ERM).

2.2.1 Gradient-Based Optimization

There exist several methods to optimize the learning objective; in this thesis, we focus on *gradient-based* methods, which use the information provided by the gradient of the loss function (or an approximation thereof) to minimize the training error. Clearly, to apply gradient-based methods, the loss function has to be differentiable. By far, the most widely used gradient-based method in modern ML is Stochastic Gradient Descent (SGD). Briefly, SGD reduces the value of the loss function by "moving" the parameters in the direction of its negative gradient. The process requires to initialize the parameters to some randomly chosen value $\omega(0)$, and to iterate over the dataset several times. At each iteration t, the parameters are updated according to the following update rule:

$$\omega(t+1) = \omega(t) - \eta_t \boldsymbol{\nabla} J(\omega(t)),\ t = 0, 1, \ldots,$$

where η_t is a per-iteration *learning rate* that controls the magnitude of the update, and:

$$\boldsymbol{\nabla} J(\omega(t)) = \frac{1}{m}\boldsymbol{\nabla}_{\boldsymbol{\theta}} \sum_{i=1}^{n_B} \mathcal{L}(\omega(t), z^{(i)})$$

is an estimate of the true gradient of the loss function, averaged over a *mini-batch* of $m \ll n$ data points. SGD has several desirable properties as regards its convergence: specifically, given a convex loss function, if the learning rate is decreased at an appropriate rate, SGD converges to its global minimum almost surely as $t \to \infty$; if the loss function is not convex, under the same assumptions SGD converges to one local minimum almost surel. Over the years, different variants of SGD have been developed; a program that implements a specific variant is called **optimizer**.

2.2.2 Overfitting

One desired property of ERM is that the empirical risk is guaranteed to approximate arbitrarily well the true risk as $n \to \infty$, provided that the hypotheses space has enough **capacity**. The capacity of a hypotheses space is the scope of functions it is able to learn: the higher it is, the more "complex" functions can be learned. However, if the capacity of the hypotheses space is unconstrained, one might incur in **overfitting**, a phenomenon where the learned model perfectly predicts the examples in the dataset (achieves very low training error), but is unable to generalize to unseen examples (has high generalization error). Intuitively, an overly-expressive hypotheses space is likely to contain hypotheses so complex, that are able to fit not only the true relationships in the data, but also the sampling noise in the dataset. Such hypotheses would be then wrongly chosen as the best hypotheses according to the ERM principle. Overfitting can be observed by dividing the dataset in two disjoint partitions: a **training set** which is used for learning, and a **test set**, which is held out from the training procedure and is used to get an estimate of the true generalization error. To detect overfitting, one should monitor if the generalization error (estimated in the test set) diverges from the training error (computed on the training set) during the learning process.

2.2.3 Regularization

Besides using a separate test set for detection, overfitting can be prevented *a priori* using **regularization** techniques. The general idea of regularization is to limit, directly or indirectly, the **complexity** of the hypoteses space used by the learning algorithm. A principled form of regularization of the learning process derives from the field of Statistical Learning Theory (SLT), which studies, among other problems, the relationship between the training error and the generalization error. One important result of SLT is the so-called **generalization bound**, which states that, if the complexity $\mathrm{C}(\mathcal{H}_\Omega)$ of a hypotheses space is known[1], the following inequality:

[1] The complexity of a hypotheses space can be measured, among other techniques, calculating its *VC-Dimension*, whose precise definition is beyond the scope of this work.

$$R(h_\omega) \leq R_{\mathbb{D}_n}(h_\omega) + \varepsilon(n, \mathrm{C}(\mathcal{H}_\Omega), \delta)$$

holds with probability of at least $1-\delta$ if $n > \mathrm{C}(\mathcal{H})$. In other words, the generalization bound tells us that the generalization error is bounded by above by the training error and a *confidence term* ε, which depends on the number of examples n and the complexity of $\mathcal{H}_\Omega$. These two terms are tightly related: the confidence term can be decreased by getting more data (increasing n) if possible, or by restricting the complexity of the hypothesis space using regularization. This second choice, however, leads to an increase of the training error. The bound induces a simple principle to learn effectively while avoiding overfitting, called Structural Risk Minimization (SRM), which requires minimizing both the training error and the confidence term.

2.3 Model Evaluation

With the goal of generalization in mind, a learning algorithm needs to be evaluated on unseen data. The name **model evaluation**, or *model assessment*, refers to the task of getting a proper estimate of generalization capability of the model. In model evaluation, one is not necessarily interested to evaluating the generalization error; in most practical cases, another performance metric is used. For example, in classification tasks, the *accuracy* of the model is evaluated instead. The evaluation is performed on the test set; different estimators are defined by the different ways in which we can split the dataset into training and test partitions. The simplest strategy is the **hold-out** strategy, which splits the data into one training set and one test set, according to some predefined proportion. The parameters of the model are found using the training set, and the performance estimate is obtained from the test set. According to the field of statistics, an estimator can be decomposed in two related quantities: *bias* and *variance* (see *e.g.* **?** for a formal treatment of the subject). Informally, bias is related to how much the estimation is close to the true value; variance is related to how much the estimation depends on the specific dataset on which it is obtained. "Good" estimators trade-off between the two. For small datasets, the hold-out estimator has high variance, since it is obtained on a single test set and over-estimate the true performance just by chance. To reduce the variance of the estimation, k-fold Cross-Validation (CV) can be used instead. k-fold CV consists in splitting the data in k disjoint partitions and repeat the evaluation k times. Each repetition is a hold-out estimation where one partition in turn acts as test set, and the remaining $k-1$ form the training set. The final estimator is the average of the k hold-out estimators. In practical cases, $k = 5$ or $k = 10$ are often used.

2.3.1 Model Selection

Learning algorithms are such that finding good values of the parameters is usually not enough. In fact, the learning process is also influenced by other settings, not directly optimizable by gradient descent in parameter space. These extra settings are called **hyper-parameters**. Some examples are the learning rate η and the number of iterations of SGD; other hyper-parameters are specific to the particular learning algorithm used. The process of jointly choosing the parameters and the hyper-parameters of a model is called **model selection**, or *hyper-parameter tuning.* Model selection requires some a set of hyper-parameter configurations to be evaluated. Given a configuration, a model is instantiated with the corresponding hyper-parameters, trained on the training set and evaluated on some held-out dataset. Usually, the hyper-parameters set is specified as a *grid*, where each hyper-parameter is associated to a discrete set of possible choices. When this is the case, model selection is referred to as *grid search*, and the algorithm is simply an exhaustive evaluation of all possible hyper-parameter combinations. Other strategies for defining the set of hyper-parameters and the model selection algorithm are also possible **?**. Model selection requires a separate set of data than the test set. In fact, if done on the test set, the set of hyper-parameters found would be tailored on that specific test set, and the corresponding performance would be an over-optimistic (biased) estimate of the true performance. In general, the test set cannot be used to take decisions about the learning process, regardless of whether the decision concerns the parameters or hyper-parameters of the model. This problem is solved by using one or more parts of the training set as a **validation set**, where the effect of the different hyper-parameters on the performances is estimated. In this sense, model selection can be viewed as a nested assessment inside the more broad model evaluation task, and can be tackled using the same kinds of estimators such as k-fold CV. Later on in this thesis, we shall see some strategies of how model selection and evaluation are performed jointly in practical settings.

2.4 Neural Networks

This work revolves around artificial NNs, or NN in short, a learning algorithm inspired by the mechanisms through which the neurons in our brain learn. In brief, biological neurons acquire electrical signal from neurons they are connected to, and send it over to other neurons if the strength of such signal exceeds a given threshold. Learning in this context can be intended as the process that adjusts the “strength” of the neural connections according to the signal provided in input, such that a specific behaviour is obtained. For this reason, the neural approach to ML is often called *connectionist*, borrowing this terminology from the computational neuroscience field. Mathematically speaking, given an input signal $\mathbf{x} \in \mathbb{R}^d$, the general function performed by a neuron can be abstracted by the following:

$$y = g\left(\sum_{i=1}^{d} \mathbf{w}_i \mathbf{x}_i + b\right),$$

where $\mathbf{w} \in \mathbb{R}^d$ are the weighted connections, $b \in \mathbb{R}$ is called bias term, g is the threshold function (usually called **activation function**), and $y \in \mathbb{R}$ is the output.

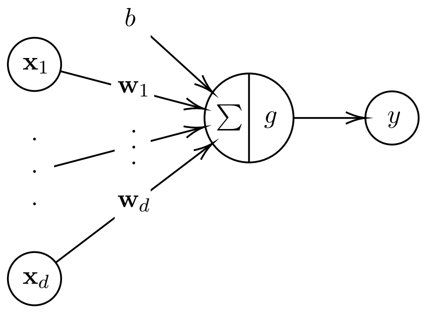


FIGURE 2.1: Schematics of a computational neuron.

A visual representation of the artificial neuron is shown in Figure 2.1. NNs typically implement hypotheses spaces consisting of a hierarchy of function compositions to map an input $\mathbf{x}$ to the network output:

$$g_{\boldsymbol{\theta}}(\mathbf{x}) = (g_{\boldsymbol{\theta}_{\text{out}}} \circ g_{\boldsymbol{\theta}_L} \circ ... \circ g_{\boldsymbol{\theta}_2} \circ g_{\boldsymbol{\theta}_1})(\mathbf{x}),$$

where $g_{\boldsymbol{\theta}_1}$ is called **input layer**, $g_{\boldsymbol{\theta}_{\text{out}}}$ is called **output layer**, $g_{\boldsymbol{\theta}_l}$ for $l = 2, \dots, L$ are called **hidden layers** and $\boldsymbol{\theta} = (\boldsymbol{\theta}_{out}, \dots, \boldsymbol{\theta}_1)$ are the parameters or **weights** of the network. Intuitively, a layer is a group of output neurons (also called **units**) attached to the same input neurons, as in the example of Figure 2.2a. Each layer of the network computes the following function:

$$g_{\boldsymbol{\theta}_l}(\mathbf{x}) = g_l(\mathbf{W}_l^{\mathsf{T}}\mathbf{x} + \mathbf{b}_l),$$

where $\boldsymbol{\theta}_l = (\mathbf{W}_l, \mathbf{b}_l)$ are its weights. More specifically, if $g_{\boldsymbol{\theta}_{l-1}}$ and $g_{\boldsymbol{\theta}_l}$ are two consecutive layers with d_{l-1} and d_l units respectively, $\mathbf{W}_l$ is a weight matrix in $\mathbb{R}^{d_{l-1} \times d_l}$, and $\mathbf{b}_l$ is a bias vector in $\mathbb{R}^{d_l}$. The activation function g_l of a hidden layer is an

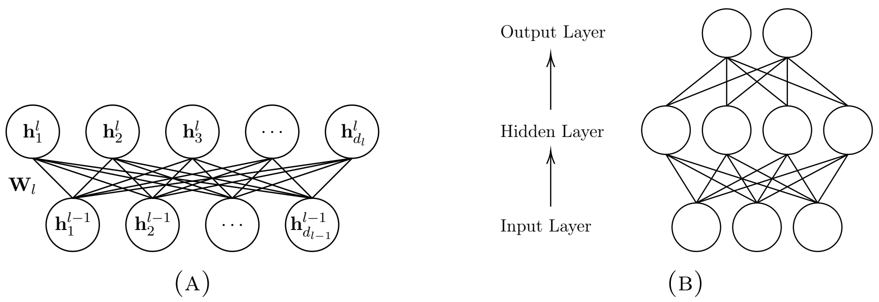


FIGURE 2.2: (A) An example of a neural network layer. The superscript above the units indicates the layer they belong to. (B): A Multi-Layer Perceptron with $L = 3$ layers (biases not shown). The arrows indicate the flow of forward propagation.

element-wise non-linearity, and its output is often called *activation.* The most common non-linear activation functions for hidden layers are shown in Figure 2.3. To work with SGD, they are usually differentiable. Two very common examples are the *sigmoid* function, defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

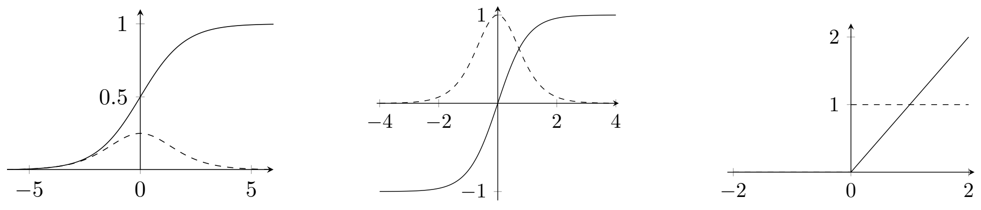
which maps its input in the range $(0, 1) \subset \mathbb{R}$, and the *hyperbolic tangent* function, defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

whose output range is $(-1, 1) \subset \mathbb{R}$. In modern NNs, the activation function of the hidden layers is generally some variant of the Rectified Linear Unit (ReLU) function, which is defined as:

$$\mathrm{ReLU}(x) = \max(0, x).$$

Notice that, even though the ReLU function is not differentiable at $x = 0$, it can be implemented to work in practical settings. The activation function of the output layer is task-dependent, and is tightly coupled with the loss function used to train the network, as we shall see in Section 2.4.2. The *architecture* of a network is a specification



(A) Sigmoid (B) Hyperbolic Tangent (C) Rectified Linear Unit

FIGURE 2.3: Examples of activation functions for the hidden layers (in solid black) and their derivatives (in dashed black).

of the number of layers L, the dimensions of the parameters $\boldsymbol{\theta}$ for each layer, and the corresponding activation functions. In practical settings, the architecture is a hyper-parameter and needs to be found with model selection. Clearly, the only part of the architecture that is not a hyper-parameter is the dimension of the input layer, which depends on the data the network uses. A NN with $L \geq 2$ layers (including the output layer), where g_l is a non-linear activation function is called Multi-Layer Perceptron (MLP), and is the most common form of NN. MLPs are extremely versatile learning algorithms: in fact, it has been proved that they are *universal function approximators*, meaning that they can approximate any computable function with arbitrary precision, given an appropriate architecture **?**. Figure 2.2b shows an example of MLP with two hidden layers.

One distinctive advantage of NNs with respect to other learning algorithms is *automatic feature engineering*: in fact, the hidden layers of a NN extract features from the data without an explicit guidance from the end user, nor from the training procedure. This means that the network itself decides which **representation** of the data is relevant to solve a given task by tuning the parameters during learning; this decision is driven by the high-level goal of approximating the input-output relationship of interest. This ability is crucial in tasks where knowledge is difficult to represent, or where there is not enough domain expertise to manually devise features, and has been the key of the success of NNs in ML.

2.4.1 Training

Learning in a NN requires several passes through the training set. Each pass is called an *epoch*, and consists of two phases. In the first phase, training data points are fed to the network and passed to each of its layers sequentially, such that the output of a layer becomes the input of the following layer. The output layer of the network produces a *prediction* for each data point. This phase is called *forward propagation*, which is why NNs are often called **feed-forward**. Subsequently, the error between the prediction and the ground truth (the labels y) is calculated via the loss function. The second phase consists in propagating the error back to the hidden layers to calculate the gradient of the loss function at each layer. This is done via the well-known **backpropagation** algorithm, which is basically an application of the chain rule of derivation for function composition. Once the gradient of the loss function is available at each layer, the parameters are updated using SGD. Training NNs requires particular care to ensure the convergence of the SGD algorithm. In particular, the initial values of the weight matrices should be initialized with small random values around 0 for *simmetry breaking*, and the learning rate must be < 1, to prevent too large update steps which would make the objective function diverge from the local minima.

2.4.2 Loss Functions

As anticipated, the choice of the output layer of a NN determines the kinds of tasks it is able to learn. In turn, each task is associated to a specific loss function that must be minimized by the optimizer, in order to find a set of parameters that generalize. NNs are usually trained with Maximum Likelihood Estimation (MLE) under a parameterized conditional distribution of the outputs of the network. More specifically, focusing on supervised learning for the moment, assume the unknown function f is some conditional $p(y \mid \mathbf{x}; \boldsymbol{\theta})$, which we approximate with a neural network $p_{\boldsymbol{\theta}}(y \mid \mathbf{x}) = g_{\boldsymbol{\theta}}(\mathbf{x})$. According to the MLE principle, this can be achieved by training the network to maximize the log-likelihood of the data. Given a dataset $\mathbb{D}_n = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, this is equivalent to the following objective function:

$$\arg\max_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y^{(i)} \mid \mathbf{x}^{(i)}).$$

Since SGD optimizers usually work by minimizing functions, it is more common to minimize the negative log-likelihood instead. Notice that MLE is tightly connected to the ERM principle. In particular, MLE is equivalent to ERM when the loss function between the true conditional and the conditional learned by the network is the Kullback-Leibler Divergence (KLD). Given a random variable x and two arbitrary distributions p and q, the KLD is defined as follows:

$$\mathrm{KLD}(p \parallel q) = \int_x p(x) \log\left(\frac{p(x)}{q(x)}\right) dx.$$

In practice, the KLD is a measure of discrepancy between distributions: it is asymmetric, meaning that $\mathrm{KLD}(p \parallel q) \neq \mathrm{KLD}(q \parallel p)$, and it evaluates to 0 if and only if $p = q$. Thus, minimizing the KLD under the ERM principle is equivalent to maximizing the likelihood under the MLE principle. Below, we detail about three of the most common output layers, and the loss functions by which the corresponding networks can be trained.

Linear Output Layer A linear output layer is used to solve regression tasks, *i.e.* tasks where the output space is continuous. Following, we consider $y \in \mathbb{R}$ for simplicity. A linear output layer is defined as:

$$g_{\theta_{\text{out}}}(\mathbf{h}) = \mathbf{W}_{\text{out}}^{\mathsf{T}}\mathbf{h} + \mathbf{b}_{\text{out}},$$

where $\mathbf{h}$ is the output of the previous layers of the network. Notice that, in this case, the activation function g_{out} is just the identity function. The model assumed for the true conditional is the following:

$$p(y \mid \mathbf{x}) \approx \mathbb{N}(g_{\theta}(\mathbf{x}), \mathbf{I}),$$

where $\mathbf{I}$ is a unit covariance matrix. In other words, the true conditional is a Gaussian distribution with unknown mean and unit variance, and the network outputs are an estimation of the true mean. A well known result of statistics tells us that applying the MLE principle to this problem is equivalent to minimizing the Mean Squared Error (MSE) loss function:

$$\mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}, y)) = \mathrm{MSE}(\mathbf{x}, y) \overset{\text{def}}{=} \|g_{\boldsymbol{\theta}}(\mathbf{x}) - y\|_2.$$

Notice that minimizing the MSE corresponds to minimizing the Euclidean distance between the target y and the prediction of the network $g_{\boldsymbol{\theta}}(\mathbf{x})$. A single linear output layer coupled with the MSE loss function is commonly known in ML literature as the Linear Regression model.

Logistic Output Layer A logistic output layer is used to solve binary classification tasks, *i.e.* tasks where the output space is the discrete set $\{0, 1\}$. The label 0 is usually called the *negative* class, while the label 1 is called *positive*. The layer is defined as follows:

$$g_{\theta_{\text{out}}}(\mathbf{h}) = \sigma\left(\mathbf{W}_{\text{out}}^{\mathsf{T}}\mathbf{h} + \mathbf{b}_{\text{out}}\right),$$

where $g_{\text{out}} = \sigma$ is the sigmoid function. Since the codomain of the sigmoid function is the real-valued interval $(0, 1) \subset \mathbb{R}$, it is suitable to express output probabilities. To apply MLE to a binary classification task, the following model for the true conditional is assumed:

$$p(y \mid \mathbf{x}) \approx \mathrm{Bernoulli}(g_{\boldsymbol{\theta}}(\mathbf{x})),$$

where. In other words, the true conditional is a Bernoulli distribution whose parameter is estimated by the neural network. According to statistics, applying MLE to the assumed model is equivalent to minimizing the Binary Cross-Entropy (BCE) loss function:

$$\mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}, y)) = \mathrm{BCE}(\mathbf{x}, y) \overset{\text{def}}{=} y \log(g_{\boldsymbol{\theta}}(\mathbf{x})) + (1 - y) \log(1 - g_{\boldsymbol{\theta}}(\mathbf{x})).$$

Training a single logistic output layer with the BCE loss function is equivalent to training a Logistic Regression model.

Softmax Output Layer The softmax output layer is used in multi-class classification tasks, *i.e.* tasks where the output space is a discrete set $\mathcal{C} = \{c_1, \dots, c_k\}$ of k mutually exclusive classes. Following, we assume the classes are represented as the integers from 1 to k for notational convenience. The layer is defined as:

$$g_{\text{out}}(\mathbf{h}) = \tau \left(\mathbf{W}_{\text{out}}^{\mathsf{T}} \mathbf{h} + \mathbf{b}_{\text{out}} \right),$$

where τ is the *softmax* activation function, defined element-wise over a generic vector $\mathbf{s} \in \mathbb{R}^k$ as:

$$\tau(\mathbf{s}_i) = \frac{e^{\mathbf{s}_i}}{\sum_{j=1}^{k} e^{\mathbf{s}_j}}.$$

In practice, a softmax layer outputs a score for each possible class, and the vector of scores is normalized to be a probability distribution by the softmax function. The model for the true conditional assumed in this case is the following:

$$p(y \mid \mathbf{x}) \approx \mathrm{Multinoulli}_k(g_{\boldsymbol{\theta}}(\mathbf{x})).$$

In other words, the true conditional is a Multinoulli (categorical) distibution for the k classes, whose parameter is estimated by the NN. Applying the MLE principle to this problem is equivalent to minimizing the Cross-Entropy (CE) loss function:

$$\mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}, y)) = \mathrm{CE}(\mathbf{x}, y) \overset{\text{def}}{=} -\sum_{i=1}^{k} \mathbb{I}[i = y] \log \mathbf{o}_i,$$

where $\mathbf{o} = g_{\boldsymbol{\theta}}(\mathbf{x})$ is the distribution outputted by the network, and $\mathbb{I}$ is the indicator function. Alternatively to the indicator function, a k-dimensional *one-hot encoded* vector can be used, where all positions are zero except the position of the true class y, which is one. In words, the one-hot vector encodes the discrete probability distribution over the possible classes, where all the mass is put on the class corresponding to the label y. Notice that the BCE loss function is just a special case of CE where $k = 2$. A NN with one single softmax output trained with CE is known in ML literature as the Softmax Regression model.

2.4.3 Regularization

As we have briefly described in Section 2.2, regularizing a ML model requires to somehow limit the complexity of the hypotheses space, such that the learning algorithm is more likely to select hypotheses that do not overfit the training data. For NNs, one straightforward approach to limit complexity is to reduce the number of hidden layers, and the number of units in each layer. Other strategies are discussed below.

Penalized Loss Function One very general regularization technique, which is not restricted to NNs, is to impose a *preference bias* to the possible values the weights might take, such that configurations that generalize achieve a lower training error than configurations that overfit. Focusing again on the supervised case, this can be achieved by augmenting the training ojective with a *penalty term* as follows:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}, y)) + \lambda \|\boldsymbol{\theta}\|_p,$$

where $\|.\|_p$ indicates a generic p-norm. The norm of the parameters implement the preference criteria used to avoid overfitting, whose influence on the training objective is controlled by a hyper-parameter λ. Depending on the value of p, we distinguish:

- $L1$ regularization, where $p = 1$. The bias of $L1$ regularization is to push the values of some weights to zero. This corresponds to training a NN with less parameters (parameters set to 0 do not contribute to the loss function);
- $L2$ regularization, where $p = 2$. The bias induced by the $L2$ norm is to penalize large values of the weights. Intuitively, restricting the range of the values that the weights can take limits the type of functions the network can approximate.

In ML literature, there exist several other types of penalties: for example, ElasticNet regularization combines $L1$ and $L2$ regularization together.

Early Stopping Early stopping is another general regularization scheme, which is widely adopted in NNs training. In short, it requires monitoring a performance metric (which can be the value of the loss function or other task-specific metrics) on the validation set, in order to stop the learning process as soon as overfitting is detected. Once learning has stopped, the parameters that yielded the best score according to the metric are chosen by the learning algorithm. Intuitively, early stopping implicitly biases the hypotheses space by limiting the number of training iterations, which in turn limits the number of hypotheses evaluated during training.

Dropout Dropout is a regularization technique that is specific to NNs. The idea behind dropout is to use a very expressive network for training, whose capacity is constrained stochastically at each iteration of the learning procedure. At each pass through a batch of training data, dropout "turns off" some of the units in a layer by multiplying their output with a binary vector mask, whose entries are samples from a Bernoulli distribution with hyper-parameter p_{keep}, called *dropout rate*. This has a double effect: first, a smaller number of parameters is used to compute a prediction (because some of them are made ineffective by the binary mask); second, the parameters used by the network are different at each pass (because of the stochasticity of the mask). According to this second implication, dropout implements a dynamic form of *model ensembling*, a ML technique in which one combines several low-capacity model to obtain a stronger predictor.

2.5 Auto-Encoders

An Auto-Encoder (AE) is a NN architecture for unsupervised learning. AEs are trained to reconstruct their inputs by jointly learning two mappings, one from the input space to a **latent space**, and another from the latent space back to the input space. During training, the latent space learns general features about the input that help the reconstruction. Given a d-dimensional input vector $\mathbf{x} \in \mathbb{R}^d$, the general form of an AE is defined as follows:

$$\begin{aligned} \mathbf{h} &= g_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x}) \\ \mathbf{r} &= g_{\boldsymbol{\theta}_{\text{dec}}}(\mathbf{h}), \end{aligned}$$

where $\mathbf{h} \in \mathbb{R}^h$ is an h-dimensional *latent code*, $\mathbf{r} \in \mathbb{R}^d$ is a *reconstruction* of the input, $g_{\boldsymbol{\theta}_{\text{enc}}}$ is an encoding NN or **encoder**, and $g_{\boldsymbol{\theta}_{\text{dec}}}$ is a decoding NN or **decoder**. Despite being an unsupervised model, AEs can be trained in a supervised way, by implicitly using a dataset of the form $\mathbb{D}_n = \{(\mathbf{x}^{(i)}, \mathbf{x}^{(i)})\}_{i=1}^n$, *i.e.* one where the input itself is the target. The loss function of an AE is called *reconstruction loss*, and measures some form of distance between the input and the reconstruction (the output of the decoder) as shown in Figure 2.4; generally, it can be the MSE for continuous inputs, or the CE for discrete inputs.

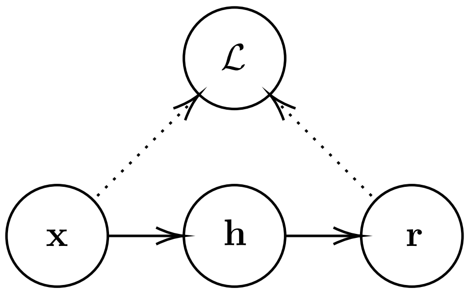


FIGURE 2.4: An Auto-Encoder.

If the latent space of the AE is not constrained in any way, it learns to just copy its input to the output: in fact, one can always have $\mathbf{x} = \mathbf{r}$ everywhere by simply imposing that $g_{\boldsymbol{\theta}_{\text{enc}}} \circ g_{\boldsymbol{\theta}_{\text{dec}}}$ is the identity function. This is not very useful in practice, as a network structured in this way would not learn anything useful about the data distribution; thus, many forms of structuring or constraining the latent space have been studied. Perhaps the simplest form of AE is the undercomplete AE, *i.e.* one where $h < d$. In this case, the latent space acts like a *bottleneck* that is trained to retain relevant features of the input, while discarding irrelevant ones. A useful byproduct of training an undercomplete AE is that the latent space acts as a *manifold*, *i.e.* a lower-dimensional subspace with Euclidean properties where data points are projected onto. One limitation of this kind of architecture is that one has to choose the capacity of the encoder and decoder very carefully. As an extreme case, consider that an over-capacitated AE with a 1-D latent space could learn to map each data point to a different integer. Clearly, this mapping would be uninformative with respect to the true data distribution.

2.5.1 Regularized Auto-Encoders

Regularized AEs generally use $h \geq d$, but impose constraints on the representations learned by the latent space through penalties on the loss function. Specifically, a regularized AE optimizes the following general objective function:

$$\arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}, \mathbf{r})) + \lambda\Psi(\mathbf{h}),$$

where different choices of Ψ define different variants. For example, *sparse* AEs are trained in such a way that the latent space produces sparse representations, meaning that only certain units are active for certain data inputs. This can be accomplished by constraining the mean activation of the units to be small. Specifically, in a sparse AE, if ρ is the average activation, the penalty is formulated as:

$$\Psi(\mathbf{h}) = \sum_{i=1}^{k} \rho \log \frac{\rho}{\hat{\rho}_i}(1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_i},$$

where $\hat{\rho}_j$ is the activation of the i-th hidden unit. If ρ is chosen to be small, the penalty constrains most hidden units to have zero activation. Similarly, *contractive* AEs regularize the objective function by imposing a penalty on the gradients of the hidden units with respect to the input. More in detail, the penalty term of a contractive AE is the following:

$$\Psi(\mathbf{h}) = \sum_{i=1}^{k} \|\boldsymbol{\nabla}_{\mathbf{x}}\mathbf{h}_i\|_2,$$

which intuitively corresponds to penalizing hidden activations that have large variation for small variations in the input. Thus, the local structure of the latent space is forced to be similar to that of the input space. Differently from the other variants, *denoising* AEs achieve regularization acting on the training procedure, rather than via a penalty term. In a denoising AE, the input $\mathbf{x}$ is corrupted before being passed to the network (usually through Gaussian noise addition). Thus, the network uses the corrupted version $\tilde{\mathbf{x}}$ during forward propagation, while the loss is calculated on the original input. By being trained to remove the noise from the input, the network is forced to learn meaningful patterns. The forward propagation phase of a denoising AE is shown in Figure 2.5.

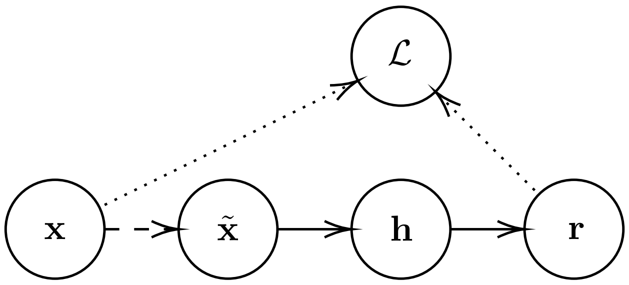


FIGURE 2.5: A denoising Auto-Encoder. The dashed arrow indicates the corruption process which transforms the input $\mathbf{x}$ into a noisy version $\tilde{\mathbf{x}}$, which is not directly part of the forward propagation.

2.6 Deep Learning

In Section 2.4, we have talked about the representational power of NNs. However, the way this representational power is attained is greatly influenced by the *depth* of the network, *i.e.* its number of layers. When a NN is "deep", the hidden features are organized during learning in a hierarchy, in which simpler features of the early layers are progressively combined into very sophisticated features in subsequent layers. Although "shallow" networks (networks with very few but large hidden layers) have their same approximating capabilities, deep networks combine features in a more computationally efficient way **?**. This data-driven *representational bias* has proven effective in many practical domains, such as computer vision **?** and Natural Language Processing (NLP) **?**, establishing the success of deep NNs in many ML tasks. The term Deep Learning (DL), coined around 2006, is used to refer to NN architectures composed of a large number of hidden layers, usually ≥ 3. The representational power of deep networks comes in hand with a number of computational challenges that arise from their depth. Below, we summarize the most well-known:

- the loss functions used for training deep networks are extremely complicated and non-convex, since the hidden layers have non-linear activation functions. This means that gradient-based methods may get stuck in bad local minima;
- very deep networks suffer from *vanishing* or *exploding gradient* issues, which can prevent the network from learning, or make the optimization numerically unstable, respectively;
- training deep NNs requires large datasets and an extensive amount of computational power.

These three major issues have been the focus of basic research on deep networks in the last years. As regards the first challenge, these problems have been tackled by better characterizing the properties of the loss function surface ?. In parallel, improvements have been achieved by devising more effective optimizers ? and more easily optimizable activation functions ?. As regards the second challenge, several mechanisms to prevent the two undesirable phenomena from happening have been proposed and applied with success; the most known are gradient clipping ? to prevent gradient explosion, batch normalization ? and residual connections ? to prevent gradient vanishing. As for the third challenge, major contributions came from the advent of the big data revolution, which ensured large and progressively more curated data sources, and the use of Graphical Processing Unit (GPU) vectorization, automatic differentiation and computational graphs to speed up the training and inference processes of deep networks.

2.6.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a kind of NN born in the field of computer vision. The development of CNNs started long before the "deep learning renaissance" in 2006, but they only recently became popular, after demonstrating their effectiveness in several image recognition tasks. Given their historical importance in the landscape of DL, we shall present them briefly even though they are not part of this thesis. The standard hidden layer of a CNN is called *convolutional layer*, and it is able to process 2-dimensional data such as images. It does so with parameterized *filters* which are applied to an input through a convolution operation. The output of a filter is usually called *feature map*. More in detail, a 2-D convolutional filter computes computes a feature map where each entry is defined as follows:

$$\mathbf{F}(i,j) = (\mathbf{M} * \mathbf{K})(i,j) = \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{M}(i+s, j+t)\mathbf{K}(s,t).$$

In the above, $\mathbf{M} \in \mathbb{R}^{n \times m}$ is a 2-D matrix (for example, an image with width n and height m), $\mathbf{K} \in \mathbb{R}^{s \times t}$ is a learnable filter, or *kernel*, with width $s \ll n$ and height $t \ll m$, and $*$ is a convolutional operator. In practice, the filter is "slid" on the input row-wise. The values in the feature map are higher if parts of the input match the filter, or more intuitively, if the pattern of the filter is detected (to some extent) in the input image. In turn, this implies that each feature of the feature map shares the same weights (those of the kernel). This approach is called *weight sharing*, and it implements *translational invariance*, *i.e.* features are detected regardless of their position in the input. Another essential hidden layer of a CNN is a *pooling layer*. Pooling works by dividing the input in non-overlapping regions, and computing an aggregation function (usually a max) for each region. Pooling serves a double purpose: firstly, it reduces the number of weights needed in subsequent layers, thus maintaining computational tractability as the depth of the network grows; secondly, it acts as a regularizer, as it discards the specific information of the region where it is applied, picking up only one representative pattern. Convolutional and pooling layers are applied sequentially to the input data, followed by a standard hidden layer activation function, usually a ReLU. The triple (Convolution-Pooling-ReLU) is the standard building block of convolutional architectures. The last layers of a CNN usually consist of a standard MLP (or a single output layer), which uses the final representation computed by the convolutional blocks as input, and compute the corresponding task-dependent output. Even though they were born within the computer vision field, CNNs have been generalized to 1-D inputs (such as sound waves in *speech recognition* tasks) and 3-D inputs (such as video frames in *object tracking* tasks).

2.7 Deep Generative Models

Many ML tasks require to learn the underlying probability distribution of the data. This is an inherently unsupervised problem, since to know this distribution, the model must capture the underlying structure of the data space, such as regions of data points with high probability. The two main operations that knowing the data distribution enables are:

- **inference**, that is, computing the density of arbitrary data points;
- **sampling**, that is, generating new data by drawing samples from the learned distribution.

Broadly speaking, a Deep Generative Model (DGM) is a model that uses deep NNs to approximate a probability distribution over random variables. In the case of learning the data distribution, the random variable is a training set $\mathbb{D}_n = \{\mathbf{x}^{(i)}\}_{i=1}^n$. DGMs are trained to minimize the KLD between the true data distribution p_{data} and a model $p_{\boldsymbol{\theta}}$, a distribution parameterized by a deep NN:

$$\arg\min_{\boldsymbol{\theta}} \text{KLD}(p_{\text{data}} \parallel p_{\boldsymbol{\theta}}) \overset{\text{def}}{=} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \left[\log p_{\text{data}}(\mathbf{x}) - \log p_{\boldsymbol{\theta}}(\mathbf{x})\right].$$

The KLD is a measure of discrepancy between distributions; it is asymmetric, meaning that $\text{KLD}(p_{\text{data}} \parallel p_{\boldsymbol{\theta}}) \neq \text{KLD}(p_{\boldsymbol{\theta}} \parallel p_{\text{data}})$, and it evaluates to 0 if and ony if $p_{\text{data}} = p_{\boldsymbol{\theta}}$. Since the term $\log p_{\text{data}}(\mathbf{x})$ does not depend on the parameters $\boldsymbol{\theta}$, we can equivalently optimize the following quantity, which has the same minimizer:

$$\arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \left[-\log p_{\boldsymbol{\theta}}(\mathbf{x})\right].$$

In practice, using MLE on the dataset $\mathbb{D}_n$, DGMs minimize the negative log-likelihood of the observed data using SGD:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{\mathbf{x}\in\mathbb{D}_n} -\log p_{\boldsymbol{\theta}}(\mathbf{x}).$$

The above objective can be minimized explicitly or implicitly. We distinguish two main categories of DGMs based on this observation:

- *explicit* DGMs minimize the log-likelihood directly. Thus, they can be used straightforwardly for inference and sampling;
- *implicit* DGMs do not minimize a log-likelihood, but a surrogate objective function that allows sampling from the data distribution as a byproduct.

In this thesis, we only deal with two specific kinds of explicit DGMs, which are described in the following. Other explicit DGMs include Sigmoid Belief Networks and Generative Flows; among implicit DGMs, we mention Generative Adversarial Networks and Generative Stochastic Networks.

2.7.1 Autoregressive Models

An Autoregressive (AR) model is an explicit DGM where the training objective is decomposed in a product of conditionals according the chain rule of probability:

$$-\log p_{\boldsymbol{\theta}}(\mathbf{x}) = -\log\left(\prod_i p_{\boldsymbol{\theta}_i}(x_i \,|\, x_{<i})\right) = \sum_i -\log p_{\boldsymbol{\theta}_i}(x_i \,|\, x_{<i}),$$

where x_i are random variables, and $x_{<i} = (x_1, x_2, \dots, x_{i-1})$. Basically, each conditional is approximated by a deep NN $p_{\boldsymbol{\theta}_i}$ that takes as input $x_{<i}$ (which for now we assume to be a fixed-size vector for simplicity) and outputs a distribution over x_i. Inference in AR models is achieved by forward propagating the input through each network in the order established by the decomposition, and summing up the intermediate probabilities. To generate a data point according to the learned distribution, it is sufficient to sample each conditional in the same order. One problem with the AR formulation is that it requires one deep network for each conditional, which is very likely to overfit the training data. This problem can be dealt with in two ways. The first is to use one single network $p_{\boldsymbol{\theta}}$ with shared parameters to learn all the conditionals; these approach is used by models such as recurrent neural networks, which we shall describe in detail in Section 3.3. Another strategy, which we do not cover this work, is to use *masking* strategies, either coupled with AEs **?** or with convolutional layers **?**, to constrain the outputs of the network to follow the order of the conditional decomposition.

2.7.2 Variational Auto-Encoders

A Variational Auto-Encoder (VAE) is a *latent variable model* where a set of latent variables $\mathbf{z} \in \mathbb{R}^z$, also called explaining factors, is incorporated to the data distribution by marginalization as follows:

$$p_{\text{data}}(\mathbf{x}) = \int p(\mathbf{x} \,|\, \mathbf{z})\, p(\mathbf{z})\, d\mathbf{z}.$$

In the above formula, $p(\mathbf{x} \,|\, \mathbf{z})$ is a *decoding distribution* and $p(\mathbf{z})$ is a prior over the latent variables, usually chosen to be a tractable distribution such as a Gaussian. The generative process expressed by a latent variable model consists in sampling from the prior a "specification" of the data, provided by the latent variables, which is used to condition the decoding distribution. Since the latent variables are not known in general, VAEs introduce an *encoding distribution* $q(\mathbf{z} \,|\, \mathbf{x})$ to produce latent variables given a data point. Instead of the data log-likelihood, VAEs work with a related quantity, called Evidence Lower Bound (ELBO) of the true log-likelihood, defined as follows:

$$\text{ELBO}(\mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim q(\mathbf{z} \,|\, \mathbf{x})} \log p(\mathbf{x} \,|\, \mathbf{z}) - \text{KLD}(q(\mathbf{z} \,|\, \mathbf{x}) \parallel p(\mathbf{z})),$$

and such that $\log p_{\text{data}}(\mathbf{x}) \geq \text{ELBO}(\mathbf{x})$. By maximizing the ELBO, the true log-likelihood of the data can be recovered. Intuitively, this can be achieved by maximizing the expected log-likelihood of the decoding distribution $p(\mathbf{x} \,|\, \mathbf{z})$ under the encoding distribution (the first term), while making the encoding distribution $q(\mathbf{z} \,|\, \mathbf{x})$ close to the prior distribution $p(\mathbf{z})$ at the same time (as specified by the KLD term). In practice, $p(\mathbf{z})$ is chosen to be a standard Gaussian $\mathbb{N}(0, \mathbf{I})$ with unit covariance matrix $\mathbf{I}$, and $q(\mathbf{z} \,|\, \mathbf{x})$ is a Gaussian distribution $q_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{z} \,|\, \mathbf{x}) = \mathbb{N}(\mu_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x}), \sigma^2_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x}))$, whose parameters are approximated by a deep NN with parameters $\boldsymbol{\theta}_{\text{enc}}$. This choice of distributions ensures that the KLD term of the ELBO can be calculated in closed form. The decoding distribution $p(\mathbf{x} \,|\, \mathbf{z})$ is implemented as another deep NN with parameters $\boldsymbol{\theta}_{\text{dec}}$. The loss function minimized by a VAE is thus the following:

$$\mathcal{L}(\boldsymbol{\theta}, \mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim q_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{z} \,|\, \mathbf{x})} - \log p_{\boldsymbol{\theta}_{\text{dec}}}(\mathbf{x} \,|\, \mathbf{z}) + \text{KLD}(q_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{z} \,|\, \mathbf{x}) \parallel \mathbb{N}(0, \mathbf{I})),$$

where the first term is a reconstruction loss, and the second term regularizes the encoding distribution by making it similar to the prior. The forward propagation of a VAE is specified as follows: first, the input $\mathbf{x}$ is mapped to the mean and variance of the encoding distribution by the encoder network. The two parameters are used to sample a latent vector $\mathbf{z}$. This is turn is given to the decoder network, which outputs a reconstruction $\mathbf{r}$. One major issue with this formulation is that this model cannot be trained with SGD, since the gradient of a stochastic operation (the sampling from the encoder distribution) is not defined. Thus, the sampling process is reparameterized as $\mathbb{N}(\mu_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x}), \sigma^2_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x})) = \mu_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x}) + \boldsymbol{\varepsilon}\sigma^2_{\boldsymbol{\theta}_{\text{enc}}}(\mathbf{x})$, with $\boldsymbol{\varepsilon} \sim \mathbb{N}(0, \mathbf{I})$. This way, the stochastic operation is independent of the input, and gradient backpropagation through the encoder becomes deterministic. The reparameterized model is shown in Figure 2.6. The VAE has several interesting properties: firstly, the latent space of a trained VAE is approximately normally distributed with 0 mean and unit variance. This means that it is compact and smooth around the mean, which in turn enables the user to seamlessly interpolate between latent representations. Secondly, it allows two possible generative modalities: an unconstrained one, which can be achieved by discarding the encoder network and starting the generative process by sampling from the prior $\mathbb{N}(0, \mathbf{I})$; and a conditional one, which is obtained by running an input $\mathbf{x}$ through the entire network. This last modality is generative in the sense that the network outputs

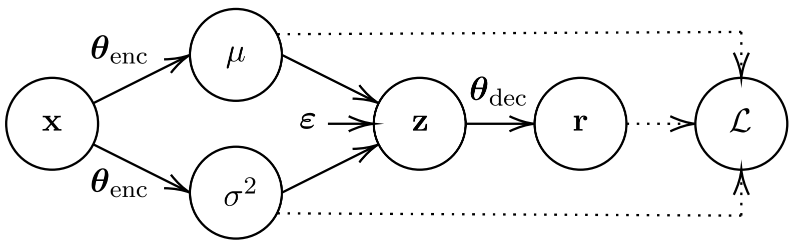

$\boldsymbol{\theta}_{\text{enc}}$
μ
$\mathbf{x}$
$\boldsymbol{\varepsilon}$
$\mathbf{z}$
$\boldsymbol{\theta}_{\text{dec}}$
$\mathbf{r}$
$\mathcal{L}$
$\boldsymbol{\theta}_{\text{enc}}$
σ^2

FIGURE 2.6: A Variational Autoencoder.

a variation (not an identical copy) of the input, due to the stochasticity induced by sampling the learned encoding distribution.

Chapter 3

Deep Learning in Structured Domains

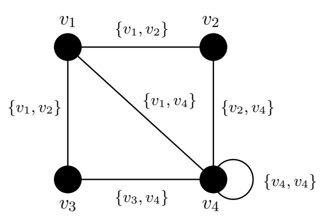
A **structured domain** is a data domain whose elements are formed by a set of atomic *entities*, and the *relations* between them. Structured data is common in several fields, such as biology, chemistry, finance, social networks, and many more. Typical examples are sequences such as time-series data, or graphs representing molecular structures. One distinctive characteristic of structured data is that it has **variable size**, meaning that the number of entities composing the datum is not fixed in general. This constitutes a serious limitation for traditional ML models, which are designed to work with "flat" data, *i.e.* collections of fixed-size vectors. In principle, they can be adapted to work with variable-sized data by incorporating the structure of the data to the input vectors as additional features. While useful to some extent, this approach requires to decide *a priori* which features are needed to solve a task. This, in turn, requires a level of domain expertise that is not always available for many interesting problems. In contrast, NNs (and Deep Learning models more so) are able to learn which features are useful to solve a task adaptively from data, without the need of feature engineering. Thus, the general idea is to provide the structured data directly as an input to the network, which automatically learns the needed features and the task, guided by the learning process. In this chapter, we present a class of NNs that are able to handle variable-sized inputs for learning in structured domains.

3.1 Graphs

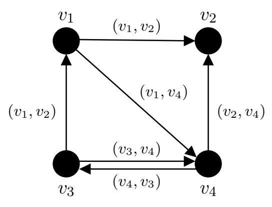
The elements of structured domains can be described in a compact and convenient notation using the general formalism of **graphs**. Informally, a graph is a collection of *vertices* (the entities) connected through a collection of *edges* (the relations). In literature, vertices are sometimes called *nodes*, while edges are also referred to as *arcs* or *links*. Formally, a graph with n vertices a pair

$$G = \langle \mathcal{V}_G, \mathcal{E}_G \rangle,$$

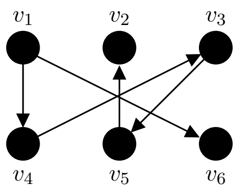
where $\mathcal{V}_G = \{v_1, v_2, \dots, v_n\}$ is its set of vertices, and $\mathcal{E}_G = \{\{u, v\} \mid u, v \in \mathcal{V}_G\}$ is its set of edges. In a graph, $\mathcal{E}_G$ specifies the graph *structure*, that is, the way



(A) An undirected graph.



(B) A directed graph.



(C) A bipartite graph.

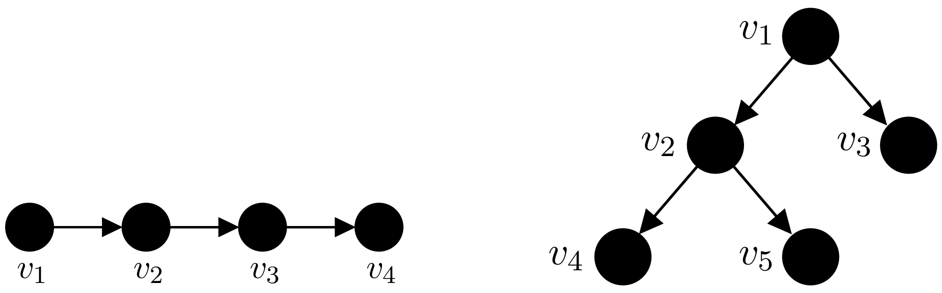
FIGURE 3.1: Three examples of graphs.

vertices and edges are interconnected. Notice that the pair $\{u, v\}$ is unordered: in this case, the graph is called **undirected**. Figure 3.1a shows a visual representation of an undirected graph. Given an edge $\{u, v\} \in \mathcal{E}_G$, u and v are called its *endpoints*, and are said to be *adjacent*. Alternatively, we say that $\{u, v\}$ is *incident* to u and v. Edges of the form $\{v, v\}$ that connect a vertex to itself are called *self-loops*. Graphs where it is possible to have more than one edge between a pair of vertices are called *multigraphs*. In this work, we restrict ourselves to the case where there is at most one possible edge between two vertices.

Directed Graphs A **directed graph** is one where he edges are ordered pairs of vertices, or equivalently one where $\mathcal{E}_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$. A directed edge is written as (u, v), meaning that it goes from vertex u to vertex v. An example of directed graph is shown in Figure 3.1b. Given a directed graph G and one of its vertices v, the set of all vertices from which an edge reaches v is called *predecessors* set, and is defined as $\mathcal{P}(v) = \{u \in \mathcal{V} \mid (u, v) \in \mathcal{E}\}$. The cardinality of the predecessors set is called the *in-degree* of the vertex, and we indicate it as $\text{degree}_{in}(v)$. Analogously, the set of all vertices reached by an edge from v is called the *successors* set, and is defined as $\mathcal{S}(v) = \{u \in \mathcal{V}_G \mid (v, u) \in \mathcal{E}_G\}$. Its cardinality is called the *out-degree* of the vertex, and indicated as $\text{degree}_{out}(v)$. The *neighborhood* (or *adjacency set*) of a vertex v is the union of the predecessors and successors sets: $\mathcal{N}(v) = \mathcal{P}(v) \bigcup \mathcal{S}(v)$. Alternatively, one can view the neighborhood as a function $\mathcal{N} : \mathcal{V}_G \to 2^{\mathcal{V}_G}$ from vertices to sets of vertices. The cardinality of the neighborhood is called the **degree** of the vertex, indicated as $\text{degree}(v)$. In this work, we consider all graphs directed unless otherwise specified. Undirected graphs are thus implicitly transformed into directed graphs with the same vertices, where the set of edges contains the edges (v, u) and (u, v) if and only if $\{u, v\}$ is an edge of the undirected graph.

Bipartite Graphs A graph G is called **bipartite** if we can split $\mathcal{V}_G$ in two disjoint subsets $\mathcal{V}_G^+$ and $\mathcal{V}_G^-$, such that $(u, v) \in \mathcal{E}_G$ if and only if either $u \in \mathcal{V}_G^+$ and $v \in \mathcal{V}_G^-$, or $v \in \mathcal{V}_G^+$ and $u \in \mathcal{V}_G^-$. Figure 3.1c shows an example of bipartite graph, where $\mathcal{V}_G^+ = \{v_1, v_2, v_3\}$ and $\mathcal{V}_G^- = \{v_4, v_5, v_6\}$.

Walks, Paths, and Cycles Let G be a graph. A *walk* of length l is any sequence of l vertices $(v_1, v_2, \ldots, v_l)$, where each pair of consecutive vertices is adjacent,



(A) A sequence. (B) A Tree.

FIGURE 3.2: Special classes of graphs.

i.e. $(v_i, v_{i+1}) \in \mathcal{E}_G$, $\forall i = 1, \dots, i-1$. A *path* of length l from vertex u to vertex v is a walk such that $v_1 = u$ and $v_l = v$, where each vertex appears exactly once in the sequence. If, given two vertices $u, v \in \mathcal{V}_G$ such that $u \neq v$, there exists a path between them, we say they are *connected*, or that v is reachable from u. Otherwise, we say they are *disconnected*, or that v is unreachable from u. A graph is called *connected* if every vertex is connected to any other vertex (ignoring the direction of the edges); otherwise it is called *disconnected*. A *cycle*, or *loop*, of length l is a walk where $v_1 = v_l$, and all the other vertices appear once in the sequence. Graphs that do not contain cycles are called *acyclic*.

Trees and Sequences A graph T is called a **tree** if its set of edges defines a *partial order* over the set of vertices, implying that it is also connected and acyclic. The vertices of a tree are called *nodes*. Given an edge $(u, v) \in \mathcal{E}_T$, we call u the *parent* of v and v the *child* of u. The set of children of a node v is indicated with the notation $\mathrm{ch}(v)$. In a tree, every node has exactly one parent, with the exception of a node called *root* or *supersource*, which has no parent node. A tree is *positional* if we can distinguish among the children of a node, *i.e.* if there exist a consistent ordering between them. Trees have a recursive structure: every node $v \in \mathcal{V}_T$ is itself the root of a tree, called *sub-tree of T rooted at v*, and indicated as $S(v)$. If $S(v)$ contains only v, v is called a *leaf*. Trees encode *hierarchical* relationships among nodes; an example of tree with five nodes is shown in Figure 3.2b.

A graph S with n vertices is called a *sequential graph*, or **sequence** of length n, if its set of edges defines a *total order* over the set of vertices, which allows us to represent the set of vertices in an ordered fashion as $\mathcal{V}_S = (v_1, v_2, \dots, v_n)$. In a sequence, the vertices are usually called *elements*. A sequence can be viewed as a special case of tree with only one leaf. Sequences are useful to encode *sequential* relationships among elements; Figure 3.2a shows an example of a sequence of four elements.

Subgraphs and Induced Subgraphs A **subgraph** $H = \langle \mathcal{V}_H, \mathcal{E}_H \rangle$ of a graph G is any graph for which $\mathcal{V}_H \subseteq \mathcal{V}_G$ and $\mathcal{E}_H \subseteq \mathcal{E}_G$. If $\mathcal{V}_H$ contains only vertices that are endpoints in $\mathcal{E}_H$, the resulting subgraph is called **induced subgraph**, or the subgraph induced by $\mathcal{V}_H$ in G.

3.1.1 Attributed Graphs

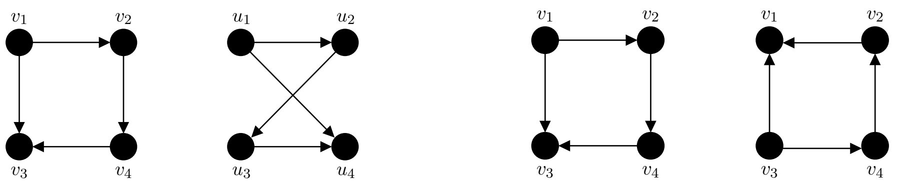
Real-world instances of graphs usually carry out other information besides structure, generally attached to their vertices or edges. As an example, consider the graph representation of a molecule, in which vertices are usually annotated with an atom type, and edges are annotated with a chemical bond type. Given a graph G with n vertices and m edges, we define the associated graph with additional information content, and we call it an **attributed graph**, as a triplet:

$$\langle G, \psi, \zeta \rangle,$$

where $\psi : \mathcal{V}_G \to \mathbb{R}^d$ is a mapping from the space of vertices to a space of d-dimensional *vertex features*, and $\zeta : \mathcal{E}_G \to \mathbb{R}^e$, is a mapping from the space of edges to a space of e-dimensional *edge features*. The values of these features can be either discrete (in which case the features are called *labels* and encoded as one-hot vectors) or continuous vectors. Sometimes, we omit to define ψ and ζ explicitly, and provide the vertex and edge features directly as sets, *e.g.* $\mathbf{x}_G = \{\mathbf{x}_{[v]} \in \mathbb{R}^d \mid v \in \mathcal{V}_G\}$ for the vertex features, and $\mathbf{e}_G = \{\mathbf{e}_{[u,v]} \in \mathbb{R}^e \mid (u,v) \in \mathcal{E}_G\}$ for the edge features. If some ordering of the vertices and edges is assumed, we can represent equivalently ψ as a matrix $\mathbf{X}_G \in \mathbb{R}^{n\times d}$ where the i-th row contains the vertex features of the i-th vertex; analogously, we can define ζ as a matrix of edge features $\mathbf{E}_G \in \mathbb{R}^{m\times e}$.

3.1.2 Isomorphisms, Automorphisms, and Canonization

An **isomorphism** between two graphs G and H is a bijection $\phi : \mathcal{V}_G \to \mathcal{V}_H$ such that $(u,v) \in \mathcal{E}_G$ if and only if $(\phi(u), \phi(v)) \in \mathcal{E}_H$. Intuitively, graph isomorphism formalizes the notion of *structural equivalence* between graphs, in the sense that two isomorphic graphs are structurally equivalent, regardless of the information they contain. Figure 3.3a shows two isomorphic graphs and their corresponding ϕ bijection. An **automorphism** $\pi : \mathcal{V}_G \to \mathcal{V}_G$ is an isomorphism between G and itself. Since π is essentially a permutation of the vertex set, it follows that a graph always has at most $n!$ possible automorphisms. Intuitively, and similarly to graph isomorphism, graph automorphisms convey the notion that the structure of a graph is invariant to permutation of the vertices and edges. An automorphism π on an example graph is shown in Figure 3.3b. Related to isomorphisms and automorphisms is the problem of **graph canonization**, where a canonical ordering (or form) of the graph vertices is sought, such that every graph H isomorph to a given graph G has the same canonical form. As we shall see, (approximate) graph canonization plays a role in the usage of graph within practical contexts; conversely, many techniques described in this work try to avoid representing graphs in canonical form, in favor of permutation-invariant representations.



$\phi(v_1) = u_1$, $\phi(v_2) = u_2$, $\phi(v_3) = u_4$, $\phi(v_4) = u_3$

(A) Isomorphism.

$\pi(v_1) = v_3$, $\pi(v_2) = v_4$, $\pi(v_3) = v_1$, $\pi(v_4) = v_2$

(B) Automorphism.

FIGURE 3.3: An example of isomorphism and automorphism.

3.1.3 Graphs as Matrices

One compact way to represent the structure of a graph is through its **adjacency matrix**. Given a graph G with n vertices and m edges, the entries of its corresponding adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ are defined as follows:

$$\mathbf{A}_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in \mathcal{E}_G \\ 0 & \text{otherwise.} \end{cases}$$

Note that the diagonal entries $\mathbf{A}_{ii}$ of the adjacency matrix specify the presence (or absence) of self-loops. Another interesting property of the adjacency matrix is that it is symmetric for undirected graphs, which implies $\mathbf{A}_{ij} = \mathbf{A}_{ji}, \forall i, j = 1, \ldots, n$. Adjacency matrices make some calculations of graph properties particularly convenient: for example, the in-degree and out-degree of a vertex $v_j \in G$ can be obtained by performing row-wise and column-wise sums on $\mathbf{A}$:

$$\text{degree}_{in}(v_j) = \sum_{i=1}^{n} \mathbf{A}_{ji} \qquad \text{degree}_{out}(v_j) = \sum_{i=1}^{n} \mathbf{A}_{ij}.$$

Adjacency matrices are also useful to understand concepts such as graph automorphisms: in fact, an automorphism of G corresponds to a permutation of the columns or rows of the adjacency matrix (but not both). Other useful matrices to represent properties graphs are the *Laplacian matrix* $\mathbf{L} \in \mathbb{R}^{n \times n} = \mathbf{D} - \mathbf{A}$, and the *symmetric normalized Laplacian matrix* $\tilde{\mathbf{L}} \in \mathbb{R}^{n \times n} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$. In both definitions, the matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ is the *degree matrix*, where all entries are zero except the diagonal entries, for which $\mathbf{D}_{ii} = \text{degree}(v_i)$. These matrices provide information about the graph connectivity through their eigenvalues and eigenvectors.

3.2 The Adaptive Processing of Structured Data

The processing of structured data for learning purposes is carried out by a **structural transduction**, namely a function $\mathbb{T} : \mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are structured domains. When the structural transduction is implemented by a (deep) NN, it is *adaptive*, *i.e.* it is learned from data. A structural transduction can be decomposed as $\mathbb{T} = \mathbb{T}_{\text{enc}} \circ \mathbb{T}_{\text{out}}$, where:

- $\mathbb{T}_{\text{enc}}$ is called *encoding function* or *state transition function* that is applied separately to each element of the structure. The output of the encoding function is a structure isomorphic to that in input, where the elements are now **state vectors**. Intuitively, a state vector encodes the information of the element and of the elements it depends on;
- $\mathbb{T}_{\text{out}}$ is called *output function*, which computes an output from the state vectors.

The output function of the structural transduction is task-dependent. Considering a supervised setting and a generic graph dataset $\mathbb{D}_n$ consisting of n training pairs, we distinguish two learning problems:

- in *structure-to-structure* tasks, the dataset has the form $\mathbb{D}_n = \{(\mathbf{x}_G^{(i)}, \mathbf{y}_H^{(i)})\}_{i=1}^n$, and the output function maps each element of the structured datum to an output. More specifically, a training pair is defined as $\mathcal{S} = (\mathbf{x}_G, \mathbf{y}_H)$, where $\mathbf{x}_G = \{\mathbf{x}_{[v]} \mid v \in \mathcal{V}_G\}$ and $\mathbf{y}_H = \{\mathbf{y}_{[\phi(v)]} \mid v \in \mathcal{V}_G\}$, with G isomorphic to H under a bijection ϕ. The MLE objective function minimized in these tasks is the following:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{\mathcal{S} \in \mathbb{D}_n} -\log p_{\boldsymbol{\theta}}(\mathbf{y}_H \mid \mathbf{x}_G) = \frac{1}{n} \sum_{\mathcal{S} \in \mathbb{D}_n} \sum_{v \in \mathcal{V}_G} -\log p_{\boldsymbol{\theta}}(\mathbf{y}_{[\phi(v)]} \mid \mathbf{x}_{[v]}),$$

  where $p_{\boldsymbol{\theta}}$ is a neural network with parameters $\boldsymbol{\theta}$ that learns an approximation of the true conditional.

- in *structure-to-element* tasks, the dataset has the form $\mathbb{D}_n = \{(\mathbf{x}_G^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^n$, and the output function maps the whole structure to a single output vector (or scalar). More specifically, a training pair is defined as $\mathcal{S} = (\mathbf{x}_G, \mathbf{y})$, where $\mathbf{y} \in \mathbb{R}^y$. The MLE objective function minimized in these tasks is the following:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{\mathcal{S} \in \mathbb{D}_n} -\log p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}_G).$$

  To learn structure-to-element tasks, the output function must compress the states of each element of the structure into a global output vector representing the entire structure, which is compared to the target $\mathbf{y}$. To do so, there are several strategies; in general, one could pick a single state vector as a representative for the whole structure, or compute a summary of the entire structure using all the available state vectors. The function that implements the latter strategy is usually termed **readout**.

As anticipated, one important issue that structural transductions need to address is how to deal with variable-sized inputs. The solution is to apply the same state transition function (that is, with the same adaptive parameters) *locally* to every element in the structure, rather than to apply it one time to the overall structure. This process is similar to the localized processing of images performed by CNNs **?**, which works by considering a single pixel at a time, and combining it with some finite set of nearby pixels. This local property of the structural transduction is often referred to as *stationarity*. An interesting byproduct of using stationary transductions is that they require a smaller number of parameters with respect to non-stationary ones, since the network weights are shared across the structure. At the same time, using stationary transductions also requires additional mechanisms to learn from the global structure of the datum (such as readouts in the case of structure-to-element tasks), rather than only locally.

In the following sections, we present three specific NN architectures that implement transductions over structured data: recurrent neural networks, which process data represented as sequences; recursive neural networks, which process hierarchical data such as trees; and deep graph networks, which process general graphs.

3.3 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a NN architecture able to process sequences. Let S be a sequence of length m whose set of elements is $\mathcal{V}_S = (v_1, v_2, \dots, v_m)$, and let $S_{\mathbf{x}} = \big(\mathbf{x}_{[1]}, \mathbf{x}_{[2]}, \dots, \mathbf{x}_{[m]}\big)$ be its element features. Here, we slightly abuse the notation $\mathbf{h}_{[v_t]}$ in favor of $\mathbf{h}_{[t]}$ since sequence elements are ordered. The state transition function of a RNN, applied locally to each sequence element, has the following general form:

$$\mathbf{h}_{[t]} = \mathbb{T}_{\text{enc}}(\mathbf{x}_{[t]}, \mathbf{h}_{[t-1]}),$$

where $\mathbf{h}_{[t]} \in \mathbb{R}^h$ is a state vector, also known as **hidden state**. The calculation of the hidden state performed by the state transition function $\mathbb{T}_{\text{enc}}$ is *recursive*: to compute a hidden state for the t-th element of the sequence, the hidden state of the previous element must be known in advance. Thus, the state computation is a sequential process, where the input sequence is traversed in order one element at a time, and the hidden state is updated as a function of the current sequence element and the hidden state at the previous step. To avoid infinite recursion, the hidden state is initialized with a vector $\mathbf{h}_{[0]}$. As the sequence is traversed, the hidden state maintains a *memory* of the past elements of the sequence. The presence of a memory mechanism makes RNNs very powerful: in fact, it has been proved that finite-size RNNs can compute any function computable with a Turing machine **?**. As with CNNs, the development of RNNs started in the early '90s, and they have recently been rediscovered within the DL framework after their success, especially in NLP-related tasks.

3.3.1 Training

Given a sequence S with features $S_{\mathbf{x}}$, the original implementation of the state transition function of a RNN is defined as follows[1]:

$$\mathbf{h}_{[t]} = \tanh\left(\mathbf{W}^{\intercal}\mathbf{x}_{[t]} + \mathbf{U}^{\intercal}\mathbf{h}_{[t-1]}\right), \ \forall t = 1, \dots, m.$$

The above is also called **recurrent layer**. The weight matrices $\mathbf{W} \in \mathbb{R}^{d\times h}$ and $\mathbf{U} \in \mathbb{R}^{h\times h}$, are shared among the sequence elements according to the stationarity property. For this reason, it is often said that the network is *unrolled* over the sequence. In structure-to-structure tasks, once the states of the elements are calculated, an element-wise output is computed as:

$$\mathbf{o}_{[t]} = g_{\text{out}}(\mathbf{h}_{[t]}), \ \forall t = 1, \dots, m,$$

where g_{out} can be any neural network such as one simple output layer or a more complex downstream network. Similarly, in structure-to-element tasks, a single output is computed from the last hidden state of the sequence:

$$\mathbf{o} = g_{\text{out}}(\mathbf{h}_{[m]}).$$

Figure 3.4b shows a RNN in compact form, as well as unrolled over a sequence of length m for a structure-to-structure task. The error of the network during training is computed by comparing the output of the network for each sequence element $\mathbf{o}_{[t]}$ to the corresponding sequence element $\mathbf{y}_{[t]}$ in the target sequence with the loss function $\mathcal{L}$, which is summed up over all the elements in the sequence. Notice that it is possible to stack multiple recurrent layers and create deep RNNs by feeding the hidden state produced the recurrent layer to a subsequent recurrent layer, other than to the next step of the recurrence. In this cases, the output is computed after the last recurrent layer. RNNs can be also adapted to learn structure-to-structure distributions of the kind $p(S'_{\mathbf{y}} \mid S_{\mathbf{x}})$, where S' and S are not isomorphic, *i.e.* when the lengths of the input and target sequence do not match. The usual way to proceed in this case is to use two RNNs: one acts as an encoder, computing a fixed-size representation of the input $S_{\mathbf{x}}$ (for example, its last hidden state as seen above); the other acts as a decoder of the target sequence $S'_{\mathbf{y}}$, conditioned on the input representation. The conditioning is achieved by initializing the hidden state of the decoder RNN with the encoding of the input computed by the encoder RNN. These types of architectures are called Sequence-to-Sequence (seq2seq) models.

RNNs are usually trained with Backpropagation Through Time (BPTT), a variant of vanilla backpropagation that propagates the gradient both from the output layer to the recurrent layer, and backwards along the sequence elements. One BPTT update requires $O(mb)$ computation, where m is the sequence length and b is the size of the mini-batch given to the optimizer. This can become computationally inconvenient for

[1]For the rest of this chapter, biases are omitted for readability.

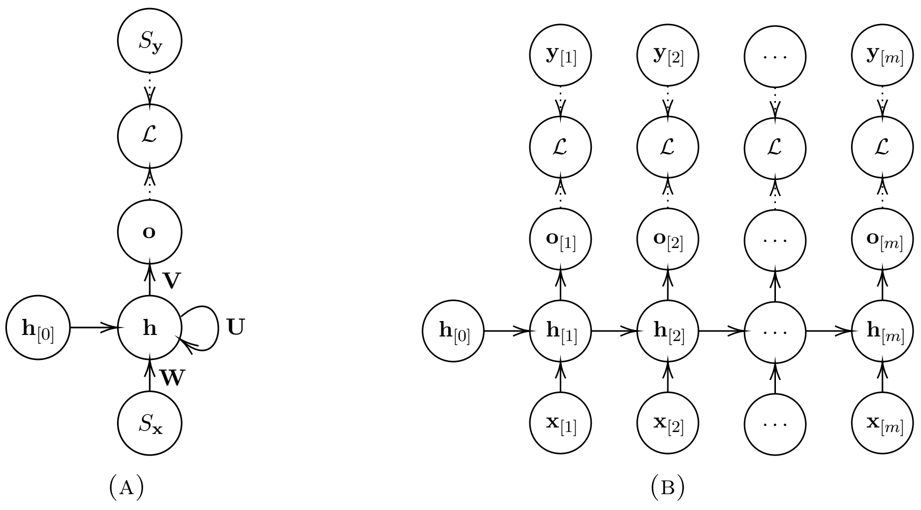

$S_{\mathbf{y}}$
$\mathcal{L}$
$\mathbf{o}$
$\mathbf{V}$
$\mathbf{h}_{[0]}$
$\mathbf{h}$
$\mathbf{U}$
$\mathbf{W}$
$S_{\mathbf{x}}$
(A)
$\mathbf{y}_{[1]}$ $\mathbf{y}_{[2]}$ $\cdots$ $\mathbf{y}_{[m]}$
$\mathcal{L}$
$\mathbf{o}_{[1]}$ $\mathbf{o}_{[2]}$ $\cdots$ $\mathbf{o}_{[m]}$
$\mathbf{h}_{[0]}$ $\mathbf{h}_{[1]}$ $\mathbf{h}_{[2]}$ $\cdots$ $\mathbf{h}_{[m]}$
$\mathbf{x}_{[1]}$ $\mathbf{x}_{[2]}$ $\cdots$ $\mathbf{x}_{[m]}$
(B)

FIGURE 3.4: (A): An example of recurrent neural network that can learn a structure-to-structure task. (B): the same network unfolded over a training pair of sequences of length m.

long sequences, and can lead to instabilities like gradient vanishing. Thus, in practical settings, faster BPTT variants are often used, such as truncated BPTT **?**.

3.3.2 Gated Recurrent Neural Networks

Vanilla RNNs struggle to learn with long sequences. This issue has been documented several times in the literature (see *e.g.* **?**), and is mostly due to the gradient vanishing or exploding problems. While gradient exploding can be dealt with gradient clipping, gradient vanishing is more hard to tackle. Several workarounds have been proposed to overcome such limitation; the most adopted in practical settings exploits a form of information *gating*. Specifically, gating mechanisms in RNNs are used to control the information flow inside the recurrent layer. In particular, it might be useful for the network to *forget* useless information, or to *reset* the hidden state when some kind of knowledge has already been processed. Gated mechanisms fulfill this purpose adaptively, driven by data. The most used RNN variant that implements gating mechanisms is the Long Short-Term Memory (LSTM). An LSTM is composed of a *cell* $\mathbf{c} \in \mathbb{R}^h$, an *input gate* $\mathbf{i} \in \mathbb{R}^h$, a *forget gate* $\mathbf{f} \in \mathbb{R}^h$, and an *output gate* $\mathbf{g} \in \mathbb{R}^h$. Assuming an input sequence element $\mathbf{x}_{[t]} \in \mathbb{R}^d$, the hidden state $\mathbf{h}_{[t]} \in \mathbb{R}^h$ of a LSTM is computed as follows:

$$\begin{aligned}
\mathbf{f}_{[t]} &= \sigma\left(\mathbf{W}_1^\intercal \mathbf{x}_{[t]} + \mathbf{U}_1^\intercal \mathbf{h}_{[t-1]}\right)\\
\mathbf{i}_{[t]} &= \sigma\left(\mathbf{W}_2^\intercal \mathbf{x}_{[t]} + \mathbf{U}_2^\intercal \mathbf{h}_{[t-1]}\right)\\
\mathbf{g}_{[t]} &= \sigma\left(\mathbf{W}_3^\intercal \mathbf{x}_{[t]} + \mathbf{U}_3^\intercal \mathbf{h}_{[t-1]}\right)\\
\tilde{\mathbf{c}}_{(t)} &= \tanh\left(\mathbf{W}_4^\intercal \mathbf{x}_{[t]} + \mathbf{U}_4^\intercal \mathbf{h}_{[t-1]}\right)\\
\mathbf{c}_{[t]} &= \mathbf{f}_{[t]} \odot \mathbf{c}_{[t-1]} + \mathbf{i}_{[t]} \odot \tilde{\mathbf{c}}_{(t)}\\
\mathbf{h}_{[t]} &= \mathbf{g}_{[t]} \odot \tanh(\mathbf{c}_{[t]}),
\end{aligned}$$

where $\odot$ is the Hadamard (element-wise) product between matrices. Notice that the weight matrices $\mathbf{W}_i \in \mathbb{R}^{d\times h}$ and $\mathbf{U}_i \in \mathbb{R}^{h\times h}$ with $i = 1, \ldots, 4$ are all different. In short, the input gate controls how much of the input is kept, the forget gate controls how much information about previous elements is kept, and the output gate controls how much of the two should be used to compute the hidden state. While powerful, a single LSTM requires eight weight matrices; thus, it is computationally expensive to train. The Gated Recurrent Unit (GRU) gating mechanism is a lightweight alternative to LSTM which uses less parameters, though it is slightly less powerful **?**. A GRU uses two gates, an *update* gate $\mathbf{u} \in \mathbb{R}^h$ and a *reset* gate $\mathbf{r} \in \mathbb{R}^h$, and computes the hidden state as follows:

$$\begin{aligned}
\mathbf{u}_{[t]} &= \sigma\left(\mathbf{W}_1^\intercal \mathbf{x}_{[t]} + \mathbf{U}_1^\intercal \mathbf{h}_{[t-1]}\right)\\
\mathbf{r}_{[t]} &= \sigma\left(\mathbf{W}_2^\intercal \mathbf{x}_{[t]} + \mathbf{U}_2^\intercal \mathbf{h}_{[t-1]}\right)\\
\tilde{\mathbf{h}}_{(t)} &= \tanh\left(\mathbf{W}_3^\intercal \mathbf{x}_{[t]} + \mathbf{U}_3^\intercal (\mathbf{r}_{[t]} \odot \mathbf{h}_{[t-1]})\right)\\
\mathbf{h}_{[t]} &= (\mathbf{1} - \mathbf{u}_{[t]}) \odot \mathbf{h}_{[t-1]} + \mathbf{u}_{[t]} \odot \tilde{\mathbf{h}}_{(t)},
\end{aligned}$$

where $\mathbf{1} \in \mathbb{R}^h$ is a vector of all ones. In practice, the reset gate controls how much information from previous sequence elements should be kept, and the hidden state is computed as a convex combination of this quantity and the previous hidden state, controlled by the update gate.

3.3.3 Recurrent Neural Networks as Autoregressive Models

RNNs can be used as autoregressive generators of sequences. In fact, a probability distribution over sequences admits a decomposition as a product of probability distributions over sequence elements, conditioned on the previous elements. More specifically, given a random variable $X = (x_1, x_2, \ldots)$ over sequences, where x_i are random variables over the sequence elements, the following decomposition:

$$p(X) = \prod_i p(x_i \,|\, x_{<i})$$

can be approximated autoregressively by an RNN trained on a dataset of sequences $\mathbb{D}_n = \{\mathbf{x}_G^{(i)}\}_{i=1}^n$. Figure 3.5a shows how the training can be achieved. In words, at a given step, the output of the network is fed as input to the next step of the recurrence, and the loss of the network is calculated between the output and the expected input sequence element. If target information is available, *i.e.* if the dataset of sequences has the form $\mathbb{D}_n = \{(\mathbf{x}_G^{(i)}, \mathbf{y}_H^{(i)})\}_{i=1}^n$, a different training strategy, called *teacher forcing* **?**, is also possible. With teacher forcing, the target information is used for the loss calculation and given as input to the next step of the recurrence (instead of the output of the network), as shown in Figure 3.5b. Both strategies have advantages and disadvantages: teacher forcing learns faster initially, but does not expose the network to its own errors, thus it can be less precise at generation time. Often, a combination of the two is used. Once the network is trained, novel sequences can be generated

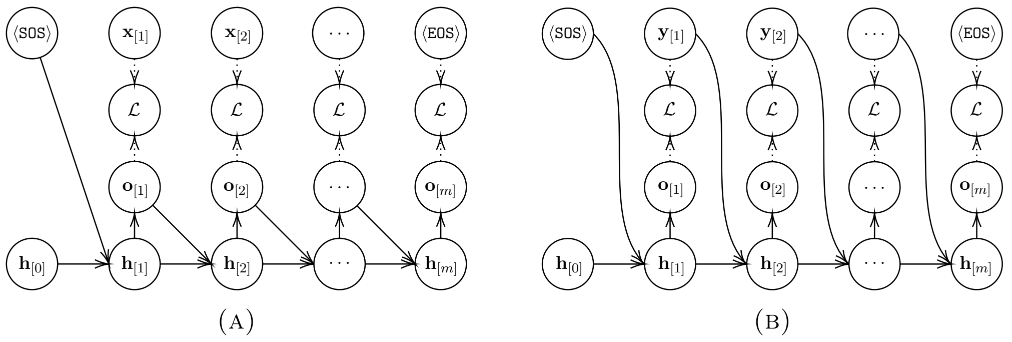


FIGURE 3.5: An example of training a recurrent neural network for learning an auto-regressive distribution. (A): standard training procedure. (B): training with teacher forcing.

according to the following scheme:

- start the generation process feeding the initial hidden state $\mathbf{h}_{[0]}$ and a *start of sequence* token ⟨SOS⟩ as input to the network;
- the output $\mathbf{o}_{[1]}$ of the network is a distribution over sequence elements. Sample a new sequence element $\tilde{\mathbf{x}}_{[1]} \sim \mathbf{o}_{[1]}$;
- feed the updated hidden state $\mathbf{h}_{[1]}$ and the sampled element $\tilde{\mathbf{x}}_{[1]}$ as input to the network, and repeat the whole process until an *end of sequence* token ⟨EOS⟩ is sampled;
- the generated sequence is $\tilde{S} = \big(\tilde{\mathbf{x}}_{[1]}, \tilde{\mathbf{x}}_{[2]}, \ldots\big)$.

3.4 Recursive Neural Networks

A Recursive Neural Network (RecNN) is a NN architecture that can adaptively process hierarchical data. Using trees as an example, let T be a tree whose set of nodes is $\mathcal{V}_T = \{v_1, v_2, \ldots, v_m\}$, and let $T_\mathbf{x} = \{\mathbf{x}_{[v]} \in \mathbb{R}^d \mid v \in \mathcal{V}_T\}$ be its set node of features.

The state transition function of a RecNN, applied locally to each node, is the following:

$$\mathbf{h}_{[v]} = \mathbb{T}_{\mathrm{enc}}(\mathbf{x}_{[v]}, \mathbf{h}_{[S(v)]}),\ \forall v \in \mathcal{V}_T$$

where $\mathbf{h}_{[S(v)]} \in \mathbb{R}^h$ is the hidden state of the sub-tree rooted at v. As with RNNs, the state transition function $\mathbb{T}_{\mathrm{enc}}$ is recursive, but this time the recursion is defined over the tree structure. Specifically, to compute the hidden state of a node, the hidden state of all its children must be known in advance. The state computation starts at the leaves of the tree (where the state is initialized beforehand to make the recursion well-defined), and proceeds bottom-up until the root node is reached. The development of RecNNs started in the middle '90s **?**, with the introduction of the notion of generalized recursive neuron **?** and the development of a general framework for learning with tree-structured data **?**, which was later extended to more expressive classes of structures such as Directed Acyclic Graphs (DAGs) and Directed and Positional Acyclic Graphs (DPAGs). Since then, they have applied fruitfully in several fields, including among others cheminformatics **?**, sentiment analysis **?** and scene parsing **?**. Interestingly, RecNNs are also backed up by strong theoretical results, which support the generality of the structural transduction and characterize the kind of functions they can learn. Specifically, universal approximation theorems showing that RecNNs can approximate arbitrarily well any function from labeled trees to real values **?**, and from labelled DPAGs to real values **?** have been proved.

3.4.1 Training

Using the binary tree of Figure 3.2b and a structure-to-element task as an example, one possible implementation of the state transition function of a RecNN is the following:

$$\mathbf{h}_{[v]} = g\left(\mathbf{W}^{\mathsf{T}}\mathbf{x}_{[v]} + \mathbf{U}_{\mathrm{l}}^{\mathsf{T}}\mathbf{h}_{[\mathrm{l}(v)]} + \mathbf{U}_{\mathrm{r}}^{\mathsf{T}}\mathbf{h}_{[\mathrm{r}(v)]}\right),\ \forall v \in \mathcal{V}_T.$$

In the above formula, g can be any hidden activation function, and $\mathbf{W} \in \mathbb{R}^{d\times h}$, $\mathbf{U}_{\mathrm{l}}$, and $\mathbf{U}_{\mathrm{r}} \in \mathbb{R}^{h\times h}$, are weight matrices shared across the structure. Notice that the two weight matrices on the node children are positional, meaning that they are applied to a certain node according to its position. In the example case of a binary tree, the two functions $\mathrm{l}(v)$ and $\mathrm{r}(v)$ select the left and right child of a node v, respectively, if they exist. Figure 3.6 shows the unfolded RecNN over the tree, where the final output of the entire structure is obtained using the hidden state of the root node v_1 as:

$$\mathbf{o} = g_{\mathrm{out}}\left(\mathbf{h}_{[v_1]}\right),$$

where $\mathbf{o} \in \mathbb{R}^y$ is the output of the network and g_{out} can be any downstream network such as a simple output layer, or a more complex neural network. For structure-to-structure tasks, the output is calculated node-wise as follows:

$$\mathbf{o}_{[v]} = g_{\mathrm{out}}\left(\mathbf{h}_{[v]}\right),\ \forall v \in \mathcal{V}_T.$$

Notably, the order by which the hidden states need to be calculated (the numbers at the left of the hidden states in the figure) must be respected to ensure that the recursive process is well-defined. The states of nodes with the same ordering can be calculated in parallel according to the tree structure, which makes RecNNs more efficient than RNNs when compared on structures with the same number of elements. More in general, RecNNs are analogous to RNNs as to how they can be trained with

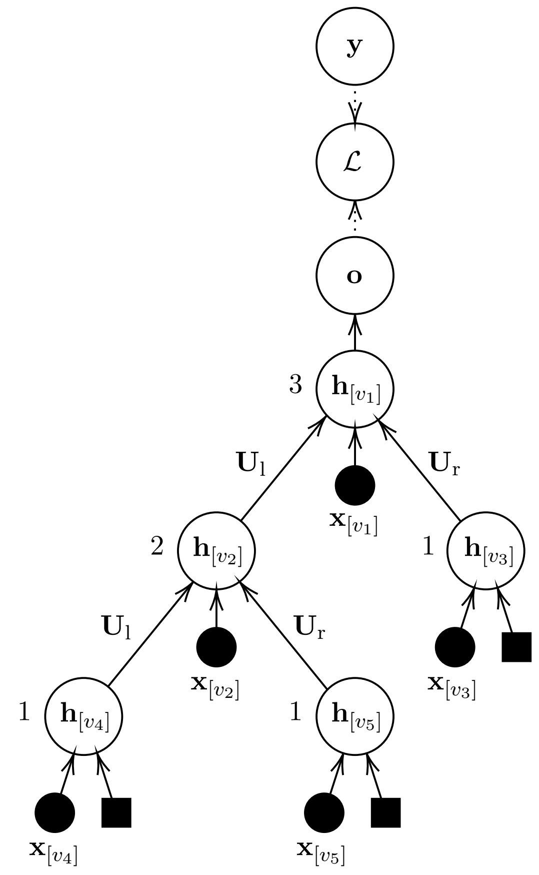


FIGURE 3.6: A recursive neural network unfolded over the tree of Figure 3.2b for a structure-to-element task. The number at the left of the hidden states indicates the order in which they are calculated. The black boxes represent "null" state vectors used to initialize the process at the leaves.

MLE, and as what kinds of conditional distributions they can learn (even though in practical cases the structure-to-element scenario is more common).

Part II

Deep Learning on Graphs with Applications to Computational Biology

Chapter 4

Deep Graph Networks

The RNN and RecNN models presented in Chapter 3 share the idea that the state transition function is applied locally and recursively on the structure to compute the state vectors. Extending it to arbitrary graphs (which can have cycles) would require to apply the state transition function recursively to the neighbors of a vertex. However, this approach is not applicable to general graphs. In fact, the presence of cycles creates *mutual dependencies*, which are difficult to model recursively and may lead to infinite loops when computing the states of vertices in parallel. While this issue can be overcome by resorting to canonization techniques that provide an ordering between the vertices, it is not feasible in many practical cases. Deep Graph Networks (DGNs) are a class of NNs which can process arbitrary graphs, even in presence of cycles. The solution adopted by DGNs to the problem of modelling mutual dependencies is to update the state of the vertices according to an *iterative* scheme. Specifically, the hidden state of a vertex is updated as a function of the hidden state of the same vertex at the previous iteration. Given a graph G with vertex features $\mathbf{x}_G = \{\mathbf{x}_{[v]} \mid v \in \mathcal{V}_G\}$, the state transition function computed by a DGN, applied locally to each vertex of G, has the following form:

$$\mathbf{h}^{\ell}_{[v]} = \mathbb{T}_{\text{enc}}(\mathbf{x}_{[v]}, \mathbf{h}^{\ell-1}_{[v]}),\ \forall v \in \mathcal{V}_G$$

where $\mathbf{h}^{\ell}_{[v]} \in \mathbb{R}^h$ is now the state of vertex v at iteration ℓ. Notice how the value of the state vector does not depend on the value of the neighboring state vectors, but to the same state vector at the previous iteration. Following, we slightly change terminology and refer to the vertices of a graph as nodes, in accordance to the terminology currently used in the literature. For the same reason, we shall use the following terminology as regards the supervised tasks that can be learned with DGNs:

- structure-to-structure tasks shall now be termed **node classification** tasks if the targets are discrete node labels, or **node regression** tasks if the targets associated to the nodes are continuous vectors or scalars. We further distinguish among *inductive* node classification (respectively, regression) tasks, if the prediction concerns unseen graphs; and *transductive* node classification (respectively, regression) tasks, if the structure of the graph is fixed (*i.e.* , the dataset is composed of one single graph), and the task is to predict from a subset of nodes for whose target is not known. The transductive setting is often referred to as semi-supervised node classification (respectively, regression);

- structure-to-element tasks shall now be termed **graph classification** tasks if the target associated to the graph is a discrete label, or **graph regression** tasks if the targets are continuous vectors (or scalars).

4.1 Contextual Processing of Graph Information

Besides solving the problem of mutual dependencies in the state computations, the iterative scheme has another important purpose, that of propagating the local information of a node to the other nodes of the graph. This process is known under several names, such as **context diffusion**. Informally, the context of a node is the set of nodes that directly or indirectly contribute to determine its hidden state; for a more formal characterization of the context, see **?**. Context diffusion in a graph is obtained through **message passing**, *i.e.* by repeatedly applying the following procedures:

- each node constructs a *message* vector using its hidden state, which is sent to the immediate neighbors according to the graph structure;
- each node receives messages from its neighbors, which are used to update its current hidden state through the state transition function.

Message passing is bootstrapped by initializing the hidden state of the nodes appropriately, so that an initial message can be created. Usually, this initial message is the vector of node features. Using the example graph of Figure 4.1 as reference, we now explain how the context flows through the nodes as message passing is iterated. At iteration $\ell = 2$, the vertex v receives a single message from its only neighbor, u. The incoming message was constructed using information about the state of u at $\ell = 1$, which in turn was obtained through the state of neighbors of u at $\ell = 0$ (including v itself). Thus, the context of v at iteration $\ell = 2$ includes u as well as the neighbors of u. It is clear that, for this particular case, at iteration $\ell = 3$ the context of v would include all the nodes in the graph. Clearly, by iterating message passing, the nodes are able to acquire information from other nodes farther away in the graph. In the literature, we distinguish three different approaches by which iterative context diffusion is implemented in practice, which we describe in the following.

4.1.1 Recursive Approaches

In the recursive approach to context diffusion, message passing is formulated as a dynamical system, which is run indefinitely until the values of the hidden states converge. Some well-known representatives of this paradigm are the Graph Neural Network **?**, the Graph Echo State Network **?**, and the more recent Fast and Deep Graph Neural Network (**?**). To ensure convergence, these approaches impose contractive dynamics

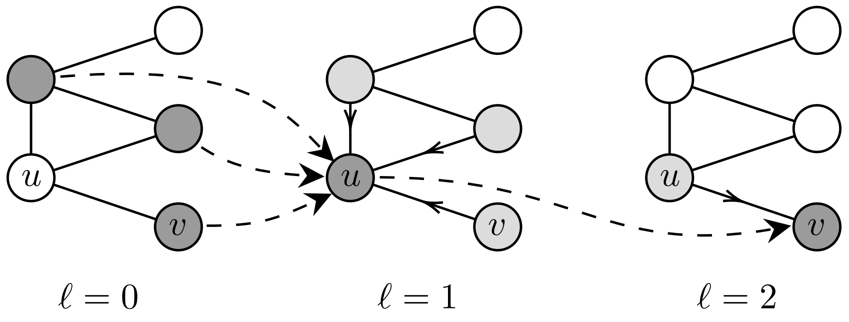


FIGURE 4.1: Context diffusion through message passing. Directed edges represent messages (*e.g.* from node u to v at iteration $\ell = 2$). Dashed arrows represent the implicit contextual information received by a node (in dark grey) through the messages from its neighbors (in light grey). Focusing on node v, its context at iteration $\ell = 2$ is composed all the dark grey nodes (including v itself).

on the state transition function. While the Graph Neural Network enforces such constraints in the (supervised) loss function, the other two inherit convergence from the contractivity of (untrained) reservoir dynamics. Another example is the Gated Graph Neural Network **?**, where, differently from **?**, the number of iterations is fixed *a priori* regardless of whether convergence is reached or not. Another approach based on *collective inference*, which adopts the same strategy but does not rely on any particular convergence criteria, has been introduced in **?**. Practically speaking, the mutual dependencies between hidden states are modeled with a single recurrent layer. In this case, the symbol ℓ represents an iteration of the recurrent state transition function.

4.1.2 Feed-Forward Approaches

The feed-forward approach is based on stacking multiple layers to compose the local context learned at each message passing iteration. As a result, the mutual dependencies between the hidden states are handled separately via differently parameterized layers, without the need of constraints to ensure the convergence of the state transition function. Thus, in the feed-forward case, the symbol ℓ indicates the layer that handles the corresponding message passing iteration. The effectiveness of the compositionality induced by the introduction of layers has been demonstrated in **?**, where it is shown formally that the context of a node increases as a function of the network depth, up to including all the other nodes in the graph. Feed-forward approaches are nowadays the main paradigm to design DGNs, due to their simplicity, efficiency, and performance on many different tasks. However, deep networks for graphs suffer from the same gradient-related problems as other deep NNs, especially when associated with an "end-to-end" learning process running through the whole architecture (**?**). For the rest of this thesis, all the DGNs used shall be feed-forward.

4.1.3 Constructive Approaches

Constructive approaches are a special case of feed-forward models, in which training is performed layer-wise. The major benefit of constructive architectures is that deep networks do not incur the vanishing/exploding gradient problem by design. In supervised scenarios, the constructive technique can learn the number of layers needed to solve a task **?**. In other words, constructive DGNs can determine automatically how much context is most beneficial to perform well, according to the specific task at hand. Another feature of constructive models is that they solve a problem in a *divide-et-impera* fashion, rather than using "end-to-end" training, by incrementally splitting the task into manageable sub-tasks. Each layer solves its own sub-problem, and subsequent layers use their results to improve further on their own, addressing the global task progressively. Among the constructive approaches, we mention the Neural Network for Graphs **?**, which was the first to propose a feed-forward architecture for graphs. Among recent models, another related approach which tackles the problem from a probabilistic point of view is the Contextual Graph Markov Model **?**.

4.2 Building Blocks of Deep Graph Networks

DGNs are built from several architectural components, which we cover in detail in this section. In short, a DGN can be decomposed into a collection of layers that process the graph structure, and a downstream predictor (either a classifier or a regressor) that computes a task-dependent output. The whole network is trained in an end-to-end fashion. In this section, we focus on the former components, the ones whose role is to carry out the processing of an input graph.

4.2.1 Graph Convolutional Layers

A Graph Convolutional Layer (GCL) is essentially a neural network layer that performs message passing. The term "convolutional" is used to remark that the local processing performed by the state transition function is a generalization of the convolutional layer for images to graph domains with variable-size neighborhoods. Given a graph G with n nodes, and its node attributes $\mathbf{x}_G = \{\mathbf{x}_{[v]} \mid v \in \mathcal{V}_G\}$, one general formulation of a GCL is the following:

$$\mathbf{h}^{\ell}_{[v]} = \mathrm{U}^{\ell}\left(\mathbf{h}^{\ell-1}_{[v]},\ \mathrm{A}(\{\mathrm{T}^{\ell}(\mathbf{h}^{\ell-1}_{[u]}) \mid u \in \mathcal{N}(v)\})\right),\ \forall v \in \mathcal{V}_G, \tag{4.1}$$

where $\mathbf{h}^{\ell}_{[v]} \in \mathbb{R}^{h^{\ell}}$ is the hidden state of the node at layer ℓ, $\mathbf{h}^{\ell-1}_{[v]} \in \mathbb{R}^{h^{\ell-1}}$ is the hidden state of the node at the previous layer $\ell-1$, and by convention $\mathbf{h}^{0}_{[v]} = \mathbf{x}_{[v]}$. Notice that the neighborhood function $\mathcal{N}$ is also implicitly passed as input of the layer, so that the connectivity of each node is known. We can identify three key functions in the above formulation:

- $\mathrm{T}^{\ell} : \mathbb{R}^{h^{\ell-1}} \rightarrow \mathbb{R}^{u}$ is a layer-wise *transform* function that applies some transformation to the hidden states of neighbors of node v at layer $\ell-1$, mapping them to some arbitrary vector space $\mathbb{R}^{u}$;
- $\mathrm{A} : (\mathbb{R}^{u} \times \mathbb{R}^{u} \times \ldots) \rightarrow \mathbb{R}^{h^{\ell}}$ is an *aggregation* function that maps a *multiset*[1] of transformed neighbors of v to a unique *neighborhood state vector*. In practice, A is a *permutation invariant* function, meaning that its output does not change upon reordering of the arguments. For this reason, the computation of the neighborhood state vector is often referred to as *neighborhood aggregation*;
- $\mathrm{U}^{\ell} : (\mathbb{R}^{h^{\ell}} \times \mathbb{R}^{h^{\ell}}) \rightarrow \mathbb{R}^{h^{\ell}}$ is a layer-wise *update* function that takes the hidden state of a node at layer $\ell-1$ and the aggregated vector, and combines them to produce the new hidden state of the node at layer ℓ.

The usage of a permutation invariant function to compute the state of the neighbors is crucial, as it allows to acquire information from nearby nodes in a non-positional fashion, which is often the case with real-world graphs. From this general formulation, several implementations can be realized. As an example, we report the well-known formulation in **?**, corresponding to the Graph Convolutional Network (GCN) model:

$$\mathbf{h}^{\ell}_{[v]} = \sigma\left(\mathbf{W}^{\mathsf{T}}_{\ell} \sum_{u \in \mathcal{N}(v)} \tilde{\mathbf{L}}_{uv} \mathbf{h}^{\ell-1}_{[v]}\right), \; \forall v \in \mathcal{V}_G, \tag{4.2}$$

where, assuming $\mathbf{h}^{\ell-1}_{[v]} \in \mathbb{R}^{h^{\ell}}$ for simplicity, $\mathbf{W}_{\ell} \in \mathbb{R}^{h^{\ell} \times h^{\ell}}$ is a layer-wise weight matrix, $\tilde{\mathbf{L}}_{uv}$ is the entry of the symmetric normalized graph Laplacian $\tilde{\mathbf{L}}$ related to nodes u and v, and:

$$\mathrm{T}^{\ell}(\mathbf{h}^{\ell-1}_{[v]}) = \mathbf{t}^{\ell}_{[v]} = \tilde{\mathbf{L}}_{uv} \mathbf{h}^{\ell-1}_{[v]} \tag{4.3}$$

$$\mathrm{A}(\{\mathbf{t}^{\ell}_{[u]} \mid u \in \mathcal{N}(v)\}) = \mathbf{n}^{\ell}_{[v]} = \sum_{u \in \mathcal{N}(v)} \mathbf{t}^{\ell}_{[v]} \tag{4.4}$$

$$\mathrm{U}^{\ell}(\mathbf{h}^{\ell-1}_{[v]}, \mathbf{n}^{\ell}_{[v]}) = \mathbf{h}^{\ell}_{[v]} = \sigma\left(\mathbf{W}^{\mathsf{T}}_{\ell} \mathbf{n}^{\ell}_{[v]}\right). \tag{4.5}$$

In this case, the aggregation function is the sum function. Other examples of permutation invariant functions used in practical contexts are the mean, the max, or other general functions which work on multisets **?**. Notice that a GCL can be applied simultaneously to all the nodes in the graph, corresponding to visiting the graph nodes in parallel, with no predefined ordering. This contrasts with RNNs and RecNN, where parallelism in the state calculations is not possible or limited, respectively.

The generic GCL can be rewritten in matrix form as some variation of the following:

$$\mathbf{H}_{\ell} = \mathrm{GCL}(\mathbf{A}, \mathbf{H}_{\ell-1}) = g\left(\mathbf{A}\mathbf{H}_{\ell-1}\mathbf{W}_{\ell}\right) \in \mathbb{R}^{n \times h^{\ell}},$$

[1]Given a set S, a multiset $\mathbb{M}(S)$ is a tuple $\langle S, \varrho \rangle$ where $\varrho : S \rightarrow \mathbb{N}_{+}$ gives the multiplicity of each element in S.

where g is a generic activation function, $\mathbf{A} \in \mathbb{R}^{n \times n}$ is the adjacency matrix of the graph, $\mathbf{H}_{\ell-1} \in \mathbb{R}^{n \times h}$ are the hidden states computed at layer $\ell-1$ where by convention $\mathbf{H}_0 = \mathbf{X}_G \in \mathbb{R}^{n \times d}$ is the matrix of node features, and $\mathbf{W}_\ell \in \mathbb{R}^{h^{\ell-1} \times h^\ell}$ is the matrix of trainable layer-wise weights. Here, the adjacency matrix substitutes the neighborhood function $\mathcal{N}$, and the node adjacency is inferred by its rows and columns. With this formulation, the GCL can be vectorized, which allows to run the state computation in fast hardware such as GPUs.

Handling Edges In certain tasks, including information about the edge features to the message passing algorithm can be beneficial to performances. Here, we describe how this can be achieved, focusing on the case where the edge features are discrete values out of a set of k possible choices. Specifically, given a graph G, we assume a set of edge features of the form $\mathbf{e}_G = \{\mathbf{e}_{[u,v]} \in \mathcal{C} \mid (u,v) \in \mathcal{E}_G\}$, with $\mathcal{C} = \{c_i\}_{i=1}^k$. To account for different edge types, two modifications to the message passing algorithm are required. One is to replace the standard neighborhood function in the aggregation function with the following *edge-aware* neighborhood function of a node v:

$$\mathcal{N}_c(v) = \{u \in \mathcal{N}(v) \mid \mathbb{I}[\mathbf{e}_{[u,v]} = c]\},$$

which selects only neighbors of v with edge type c. The other modification requires to change the update function for handling the different edge types. Taking again the GCN implementation as an example, Eq. 4.5 is modified as follows:

$$\mathrm{U}^\ell(\mathbf{h}_{[v]}^{\ell-1}, \mathbf{n}_{[v]}^\ell) = \sigma\left(\sum_{c \in \mathcal{C}} \mathbf{W}_{c,\ell}^\top \mathbf{n}_{[v]}^\ell\right),$$

where the weight matrices $\mathbf{W}_{c,\ell}$ are now edge-specific, so that the contributions of the different edge types are weighted adaptively. In practice, the above procedure corresponds to performing k different aggregations weighted separately to compute the state of the node. Other approaches to include edge information in the message passing scheme require to extend the transform function, such that the edges between the processed node and its neighbors are included in the transformation (for example, by concatenating the edge feature to the hidden state vector).

Node Attention Attention mechanisms **?** are a widely used tool in Deep Learning to get importance scores out of arbitrary sets of items. Thus, they are naturally applicable within the DGN framework to measure the contribution of the different nodes during neighborhood aggregation. Specifically, to introduce attention mechanisms in neighborhood aggregation, we weigh the contribution of the transformed nodes in the neighborhood by a scalar $a_{uv}^\ell \in \mathbb{R}$, called *attention score* as follows:

$$\mathrm{A}(\{a_{uv}^\ell \mathrm{T}^\ell(\mathbf{h}_{[u]}^{\ell-1}) \mid u \in \mathcal{N}(v)\}).$$

The attention scores are derived from *attention coefficients* w^{ℓ}_{vu}, which are essentially similarity scores between the neighbor and the current node, calculated as follows:

$$w^{\ell}_{vu} = c(\mathbf{h}^{\ell}_{[v]}, \mathbf{h}^{\ell}_{[u]}),$$

where c is an arbitrary neural network. Different attention mechanisms are defined based on how c is implemented. Finally, the coefficients are normalized into attention scores by a softmax function, effectively defining a probability distribution among them. The attention mechanism can be generalized to *multi-head* attention, where multiple attention scores for each node are calculated and concatenated together to obtain an attention vector, rather than a score. Figure 4.2 shows an example of attention computed on an example graph. We remark that node attention is unrelated to weighting the connection between nodes, which is an operation that involves the edge features. Here, similarity between nodes is calculated relying solely on the hidden states of the involved node and its neighbors.

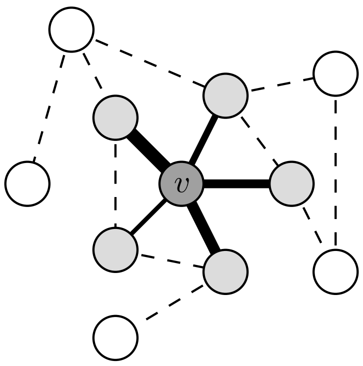


FIGURE 4.2: An example of node attention. Note that edge thickness is not related to the strength of the connection between node v (in dark grey) and its neighbors (in light grey), but it represents the degree of similarity between the node states, quantified in a probabilistic sense by the attention score. Dashed edges connect nodes that are not involved in the attention score computation.

Node Sampling Node sampling is a technique used when learning on large graphs to ensure computational efficiency. When the number of nodes in a graph is large, and nodes are densely connected among themselves, computing neighborhood aggregation may become very expensive or even intractable. The most straightforward method to address this issue is to randomly sample a predefined number of nodes to aggregate, rather than using the whole neighborhood. This basic strategy can be refined by using more sophisticated techniques such as important sampling **?**, or even extended to sampling a bounded number of nodes which are not necessarily in the immediate neighborhood of the current node **?**. The latter requires to add fictitious edges between the current node and nodes at farther distances, in order to treat them as standard neighbors. This way, global information about the graph can be incorporated more directly, as compared to message passing.

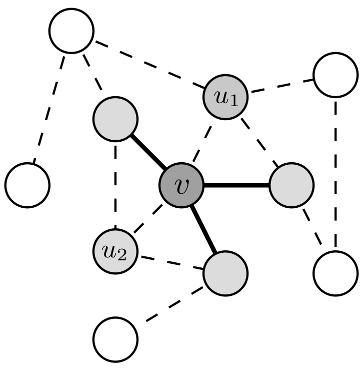


FIGURE 4.3: An example of node sampling. We focus on a node v (in dark grey), and its neighbors (in light grey). Dashed edges connect nodes that are not involved in the neighborhood aggregation; notice how nodes u_1 and u_2 are excluded from the neighborhood aggregation, even though they are effectively neighbors of v.

4.2.2 Readout Layers

As we have seen, the application of L DGN layers to a graph G yields L hidden states per node, each one composed with a progressively broad context. In node classification or regression tasks, these are combined by a *hidden state readout* function to obtain a unique hidden state to use as input of the output function, which emits a prediction for every node. Specifically, a hidden state readout function R' computes a **node representation** (or **node embeddings**) $\mathbf{h}^*_{[v]}$ for each node as follows:

$$\mathbf{h}^*_{[v]} = \mathrm{R}'(\{\mathbf{h}^\ell_{[v]}\}_{\ell=1}^L),\ \forall v \in \mathcal{V}_G.$$

Notice that, when aggregating hidden states, one can exploit the fact that the number of layers is fixed beforehand in feed-forward DGN architectures, and that the hidden states are ordered depth-wise. Thus, the aggregation need not to be permutation-invariant. Usual choices of R' include concatenation, weighted average (where the mixing weights can also be learned), RNNs, or just selecting the hidden state at the last layer. The node representations are then fed to an output layer or downstream network, which computes node-wise outputs:

$$\mathbf{o}_{[v]} = g_{\text{out}}(\mathbf{h}^*_{[v]}),\ \forall v \in \mathcal{V}_G,$$

where $\mathbf{o}_{[v]} \in \mathbb{R}^y$ and g_{out} can be any arbitrarily complex neural network as usual. A visual example of the process for a single node is shown in Figure 4.4. In graph classification or regression tasks, the node representations computed by a node readout function are aggregated once more by a *graph readout* function, to compute a **graph representation** (or **graph embedding**) $\mathbf{h}_G$, *i.e.* a vector representing the entire graph. Differently from the hidden state readout, the readout function must necessarily be permutation-invariant, since there are no guarantees about the number of graph nodes. Specifically, a graph readout function R computes the embedding of

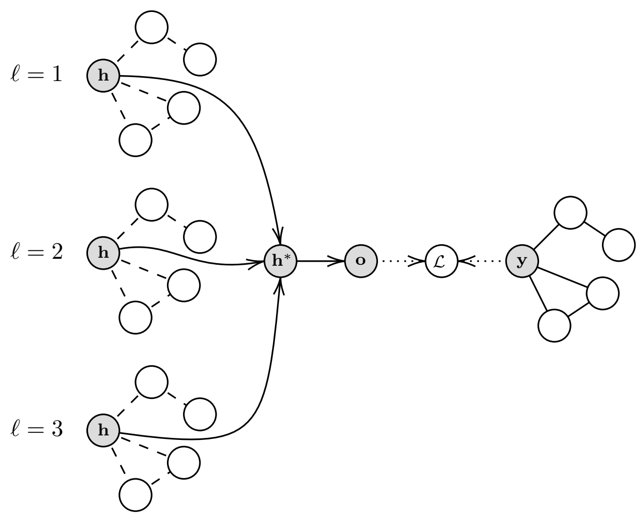


FIGURE 4.4: The role of hidden state readout function in a node classification/regression task. Here, we focus on a single node $\mathbf{h}^{\ell}_{[v]}$, where we drop the subscripts to avoid visual cluttering. Grey nodes replace the $[v]$ subscript. The hidden state readout function creates a node representation $\mathbf{h}^{*}_{[v]}$ by combining its three hidden states (one for each layer). Successively, the node representation is turned into an output by an output layer, and compared by the loss function to the same node $\mathbf{y}_{[v]}$ in the isomorphic target graph. This operation is repeated for every node in the graph.

graph G as follows:

$$\mathbf{h}_G = \mathrm{R}(\{\mathbf{h}^{*}_{[v]} \mid v \in \mathcal{V}_G\}).$$

Typical readouts for DGNs include simple functions such sum, mean, max, or more complex aggregators such as deep sets models **?**. Finally, the graph embedding is fed to an output layer or a downstream network to compute the associated output:

$$\mathbf{o} = g_{\text{out}}(\mathbf{h}_G).$$

An graph readout applied to an example graph is shown in Figure 4.5.

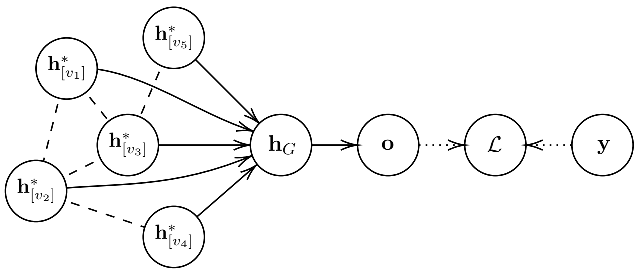


FIGURE 4.5: A graph readout on an example graph for a graph classification/regression task. Here, we assume the node representations $\mathbf{h}^{*}_{[v]}$ have already been obtained by a node readout (not shown).

4.2.3 Graph Pooling Layers

Similarly to the layer used by CNNs for computer vision, pooling is also applicable to DGNs for graph classification (or regression) tasks. In DGN architectures, pooling is usually placed after a graph convolutional layer, and serves a three-fold purpose: it is used to detect communities in the graph, *i.e.* clusters of nodes with very higher connectivity among themselves than the rest of the graph; to augment the information content of the hidden states with this knowledge; and to reduce the number of nodes (and consequently, the number of parameters) needed by the network in later stages of computation. An example of a graph pooling layer is shown in Figure 4.6, where nearby nodes are pooled into a single node in the reduced graph according to some strategy. Graph pooling methods are developed according to two strategies: *adaptive* and *tolopogical*. Adaptive methods pool nodes in a differentiable manner, so that the optimal clustering of the nodes for the task at hand is learned by the end-to-end network. One example of adaptive pooling is DiffPool, developed in **?**. Given a graph G with n nodes, and assuming the ℓ-th GCL has been applied to the hidden states, DiffPool computes two matrices:

$$\mathbf{Z}_{\ell-1} = (\mathrm{DGN}_e(\mathbf{A}_{\ell-1}, \mathbf{H}_{\ell-1})) \in \mathbb{R}^{n\times h} \quad \text{and} \quad \mathbf{S}_{\ell-1} = \tau\,(\mathrm{DGN}_p(\mathbf{A}_{\ell-1}, \mathbf{H}_{\ell-1})) \in \mathbb{R}^{n\times k},$$

where DGN_e and DGN_p is a stack of graph convolutional layers. The matrix $\mathbf{S}$ computes a soft-assignment to each node to one of k clusters with a softmax output function. These two matrices are then combined with the current hidden states to produce a novel adjacency matrix and its corresponding matrix of hidden states as follows:

$$\mathbf{H}^{\ell} = \mathbf{S}_{\ell-1}^{\intercal}\mathbf{Z}_{\ell-1} \in \mathbb{R}^{k\times h}$$
$$\mathbf{A}_{\ell} = \mathbf{S}_{\ell-1}^{\intercal}\mathbf{A}_{\ell-1}\mathbf{S}_{\ell-1} \in \mathbb{R}^{k\times k}$$

where $\mathbf{A}^0 = \mathbf{A}$ and $\mathbf{H}^0 = \mathbf{X}_G$. Thus, after applying the DiffPool layer, the size of the graph is reduced progressively from n to k nodes.

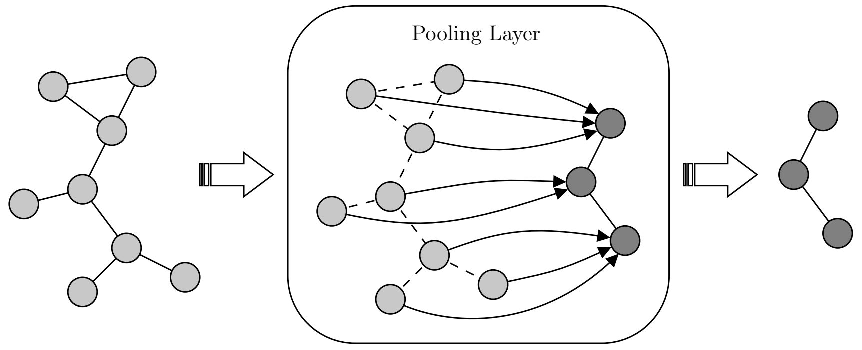


FIGURE 4.6: A visual example of a graph pooling layer.

Topological pooling uses not-differentiable methods which leverage the global structure of the graph, and the communities beneath it. cluster the nodes according to well known graph theory tools, such as spectral clustering **?**.

4.2.4 Regularization

DGNs are trained with regular losses, such as CE for classification and MSE for regression. Besides standard regularization techniques, the objective function is often regularized through unsupervised loss functions, which impose priors on which kinds of structures the network should preferably learn. The regularized objective function for supervised tasks[2] has the form:

$$\arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, (\mathbf{x}_G, y)) + \lambda \sum_{\ell=1}^{L} \Psi(\mathbf{H}^{\ell}_{G}),$$

were Ψ is a regularization function weighted by a regularization coefficient λ, that is applied at each layer ℓ to the set of hidden states of the nodes, represented as a matrix $\mathbf{H}^{\ell}_{G} \in \mathbb{R}^{n\times h}$. An example of regularization widely employed in practical settings is the *link prediction* unsupervised loss, defined as:

$$\Psi(\mathbf{H}^{\ell}_{G}) = \sum_{(u,v)} \left\| \mathbf{h}^{\ell}_{[v]} - \mathbf{h}^{\ell}_{[u]} \right\|_2, \tag{4.6}$$

where the notation (u, v) is an abbreviation to indicate all possible combinations of nodes. Basically, when this loss is minimized, it biases the network towards producing node representations that are more similar for nodes connected by an edge. Notably, this loss can be also used in isolation to tackle link prediction tasks, *i.e.* tasks where the downstream network must predict unseen links between the nodes.

4.3 The Evaluation of Deep Graph Networks

Over the years, DGNs have yielded strong performances on several predictive tasks, becoming the *de facto* learning tool for graph-related problems. Given their appeal, several DGN architectures have been developed recently. These architectures require a thorough evaluation to understand which one is better suited for a certain task. The evaluation requires both an extensive model selection phase, to select appropriate hyper-parameters, as well as a model evaluation phase to obtain an estimation of the generalization ability of the network. In the literature, the evaluation of DGNs is carried out on a variety of benchmark datasets, generally from the chemistry and social sciences domains, where graphs are used to represent molecules and social networks, respectively. However, as pointed by some researchers **?**, the papers that introduce novel architectures often adopt not reproducible or unfair experimental setups, which make the comparisons among models unreliable. In this section, we present three contributions related to address this important issue. Specifically, we:

[2]We use a generic target y to imply that this formulation is task-independent.

- provide a rigorous evaluation of existing DGNs models in the context of graph classification, using a standardized and reproducible experimental environment. Specifically, we perform a large number of experiments within a rigorous model selection and assessment framework, in which all models are compared using the same node features and data splits;

- investigate if and to what extent current DGN models can effectively exploit graph structure on the evaluation benchmarks. To this end, we also evaluate two domain-specific structure-agnostic baselines, whose purpose is to disentangle the contribution of structural information from node features;

- study the effect of node degrees as features in social datasets. We show that adding node degrees to the node features can be beneficial, and it has implications as to how many convolutional layers are needed to obtain good performances.

4.3.1 Datasets

All graph datasets are publicly available **?** and represent a relevant subset of those most frequently used in literature to compare DGNs. Some collect molecular graphs, while others contain social graphs. Specifically, we use the following chemical datasets:

- D&D **?** is a graph dataset in which nodes are amino acids, and there is an edge between two nodes if they are they are neighbors in the amino-acid sequence or in 3D space. The task is a binary classification one, where the objective is to determine whether a graph represents an enzyme or non-enzyme;

- PROTEINS **?** is a subset of D&D where the largest graphs have been removed;

- NCI1 **?** is a dataset made of chemical compounds screened for ability to suppress or inhibit the growth of a panel of human tumor cell lines. The task is a binary classification one, where the objective is to determine if a chemical compound acts as suppressor or inhibitor;

- ENZYMES **?** is a dataset of enzymes from the BRENDA enzyme database **?**. In this case, the task is to correctly assign each enzyme to one out of 6 Enzyme Commission (EC) numbers.

All As regards datasets containing social graphs, we use the following:

- IMDB-BINARY and IMDB-MULTI **?** are movie collaboration datasets. Each graph is an ego-network where nodes are actors or actresses, and edges connect two actors/actresses which star in the same movie. Each graph has been extracted from a pre-specified genre of movies, and the task is to classify the genre graph the ego-network is derived from.

- REDDIT-5K **?** is a dataset where each graph represents an online thread on the Reddit platform, and nodes correspond to users. Two nodes are connected by an edge if at least one of the two users commented each other on the thread. The task is to classify each graph to a corresponding community (a sub-reddit);
- COLLAB **?** is a scientific collaboration dataset. Each graph is an ego-network of different researchers from some research field. The task is to classify each ego-network to the corresponding field of research.

All node features are discrete (we shall refer to them as node labels equivalently). The only exception is the ENZYMES dataset, which also has an additional 18 continuous features. The social datasets do not have node features. In this case, we use either an uninformative label for all nodes, or the node degree. Specifically, social datasets are used to understand whether the models are able to learn structural features on their own or not. The statistics of the datasets, which include number of graphs, number of target classes, average number of nodes per graph, average number of nodes per graph, and number of node labels (if any) are reported in Table 4.1.

TABLE 4.1: Dataset Statistics. Note that, when node labels are not present, we either assigned the same feature of 1 or the degree to all nodes in the dataset.

		Graphs	**Classes**	**Avg. Nodes**	**Avg. Edges**	**Labels**
CHEM.	D&D	1178	2	284.32	715.66	89
	ENZYMES	600	6	32.63	64.14	3
	NCI1	4110	2	29.87	32.30	37
	PROTEINS	1113	2	39.06	72.82	3
SOCIAL	COLLAB	5000	3	74.49	2457.78	-
	IMDB-BINARY	1000	2	19.77	96.53	-
	IMDB-MULTI	1500	3	13.00	65.94	-
	REDDIT-BINARY	2000	2	429.63	497.75	-
	REDDIT-5K	4999	5	508.82	594.87	-

4.3.2 Architectures

In total, we choose five different DGN architectures. The high-level structure of the DGN comprises an input layer, a stack of one or more GCLs, a readout layer and a final MLP classifier, which maps the graph embedding to the dataset-dependent output. Following, we describe in detail the kinds of GCL used by the DGNs, and the specific hyper-parameters optimized for each of them.

Graph Isomorphism Network The Graph Isomorphism Network (GIN) convolutional layer, proposed by **?**, is implemented as follows:

$$\mathbf{h}^{\ell}_{[v]} = \mathrm{MLP}^{\ell}\left(\left(1+\epsilon^{\ell}\right) \mathbf{h}^{\ell-1}_{[v]} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}^{\ell-1}_{[u]} \right),$$

where $\epsilon \in \mathbb{R}$ is a trainable parameter, and MLP is a neural network with 2 hidden layers, each one consisting of a linear transformation, followed by a BatchNorm layer and a ReLU non-linearity. The first layer of the DGN is not a GCL, but a graph readout that sums the node features, and passes them through an MLP identical to the one just described. The graph embedding obtained by this layer is then summed to the graph embeddings computed by the GCLs. For this architecture, we choose the following hyper-parameters: number of layers, size of the hidden state at each layer, and type of readout function (for the first non-convolutional layer as well).

GraphSAGE The Graph SAmple and aggreGatE (GraphSAGE) convolutional layer, proposed by **?**, is implemented as follows:

$$\mathbf{h}^{\ell}_{[v]} = \sigma\left(\mathbf{W}^{\top}_{\ell} \mathbf{h}^{\ell-1}_{[u]} + \frac{1}{|\mathcal{N}(v)|} \mathbf{U}^{\top}_{\ell} \sum_{u \in \mathcal{N}(v)} \mathbf{h}^{\ell-1}_{[u]} \right).$$

Notice that the original formulation also implements a form of node sampling, since it is applied to large graphs. Here, since the size of the graph we deal with is manageable, we do not implement node sampling.

GraphSAGE + DiffPool In order to have a representative DGN with pooling, we tested a variant of GraphSAGE, where a DiffPool layer is added to the network after every GraphSAGE convolution. Internally, the DiffPool layer is implemented as in Section 4.2.3, where DGN_e and DGN_p are a stack of 3 GraphSAGE layers respectively. Thus, one DiffPool layer requires to compute 6 graph convolutions. The number of clusters k is deterministic, and is obtained as αM, where M is the maximum number of nodes among the graphs in the dataset, and α is a coarsening factor which is 0.1 if only one DiffPool layer is applied, and 0.25 otherwise.

ECC The Edge-Conditioned Convolutional layer, presented in **?**, is implemented as follows:

$$\mathbf{h}^{\ell}_{[v]} = \sigma\left(\frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} \mathrm{MLP}^{\ell}(\mathbf{e}_{uv})^{\top} \mathbf{h}^{\ell-1}_{u} \right),$$

where MLP is a neural network that computes a weight between the hidden state of the current node $\mathbf{h}^{\ell-1}_{[v]}$ and its generic neighbor $\mathbf{h}^{\ell-1}_{[u]}$, using the edge feature vector as input, composed of one hidden layer with ReLU non-linearities and a subsequent linear output layer. Each convolutional layer of the corresponding DGN is composed of a stack of three ECC layers.

DGCNN The graph convolutional layer of the Deep Graph Convolutional Neural Network (DGCNN), presented in **?**, is the following:

$$\mathbf{h}^{\ell}_{[v]} = \tanh\left(\mathbf{W}^{\intercal}_{\ell} \sum_{u \in \mathcal{N}(v)} \mathbf{D}^{-1}_{vu} \mathbf{h}^{\ell-1}_{[u]}\right),$$

where $\mathbf{D}^{-1}$ is the inverse degree matrix. After all the convolutional layers are applied, the computation is followed by SortPool layer **?**, which performs graph pooling, and a final 1-D CNN.

4.3.3 Baselines

We compare the proposed architectures with two structure-agnostic baselines, one for molecular graphs, and one for social graphs. For molecular graphs, with the exception of graphs in the ENZYMES dataset, we use the model proposed by **?**, applied as follows to a generic graph G:

$$\mathbf{h}_G = \text{MLP}\left(\sum_{v \in \mathcal{V}_G} \mathbf{x}_{[v]}\right).$$

This architecture corresponds to summing the node features together (which are one-hot encoded vectors representing the atom types), and applying an MLP with 2 hidden layers with ReLU non-linearities on top of them. In practice, the network counts the atom occurrences for each atom type, and computes non-linear transformation of this sum. Notice that the graph connectivity is not taken into account by the model. For the social graphs, and for the graphs in the ENZYMES dataset (which uses 18 additional node features with respect to the other molecular datasets), we use the following architecture, applied to a generic graph G as follows:

$$\mathbf{h}_G = \text{MLP}_2\left(\sum_{v \in \mathcal{V}_G} \text{MLP}_1(\mathbf{x}_{[v]})\right),$$

where MLP_1 is a linear layer plus a ReLU non-linearity, and MLP_2 is a two hidden layer MLP with ReLU non linearities. This model is also known as Deep Sets in the literature **?**. Notice that even in this case we do not take into account the graph connectivity. The role of these baselines is to provide feedback on the effectiveness of DGNs on a specific dataset. Specifically, if a DGN performs similarly to a structure-agnostic baseline, one can draw two possible conclusions: either the task does not need structural information to be effectively solved, or the DGN is not exploiting graph structure adequately. While the former can be verified through domain-specific human expertise, the second is more difficult to assess, as multiple factors come into play such as the size of the training dataset, the structural inductive bias imposed by the architecture and the selected hyper-parameters. Nevertheless, a significant improvement with respect to a baseline is a strong indicator that graph structure has been exploited.

4.3.4 Experimental Setup

In all our experiments, we use classification accuracy (*i.e.* the percentage of correctly classified graphs out of the total number of predictions) as performance metric. Our evaluation pipeline consists in an outer 10-fold CV for model assessment, and an inner holdout technique with a 90%/10% training/validation split for model selection. After *each* model selection, we train the winning model three times on the whole training fold, holding out a random fraction (10%) of the data to perform early stopping. We do this in order to contrast the effect of unfavorable random weight initialization on test performances. The final score for each test fold is obtained as the average of these three runs. Importantly, all data splits have been precomputed, so that models are selected and evaluated on the same data partitions; this guarantees consistency in the evaluation. For the same reason, we also stratify all the class labels, so that the classes proportions are preserved inside each 10-fold split, as well as in the internal holdout splits. With respect to the general DGN architecture, we optimize learning rate and early stopping patience for every considered DGN. All models are trained with the Adam **?** optimizer with learning rate decay.

Hyper-Parameters For all the DGN architectures, we tune the size of the hidden state of the convolutional layers and the number of layers. We keep the number of configurations roughly equal across all the tested models. The baselines are composed of a single layer, hence we only tune the hidden size. Besides architecture-specific hyper-parameters, we also tune others that are shared across all models, related to the training procedure. Specifically, for each model under evaluation, we optimize learning rate and learning rate decay. For GIN, we also optimize batch size. For the two baselines, we also optimize the L2 regularization factor. To be consistent with the literature, we implement early stopping with patience parameter s, where training stops if s epochs have passed without improvement on the validation set. A high value of s can favor model selection by making it less sensitive to fluctuations in the validation score at the cost of additional computation. The patience hyper-parameter is optimized for all the considered models. A table showing the complete grid of hyper-parameters we used for each DGN under evaluation is reported in Appendix A.

Computational Considerations The experiments required an extensive computational effort. For all models, the sizes of the hyper-parameter grids range from 32 to 72 possible configurations, depending on the number of hyper-parameters to choose from. The total number of single training runs to complete model assessment exceeds 47000. Such a large number requires an extensive use of parallelism, both in CPU and GPU, to conduct the experiments in a reasonable amount of time. In some cases (e.g. ECC in social datasets), training on a *single* hyper-parameter configuration required more than 72 hours; consequently, the sequential exploration of one single grid would last months. For this reason, we limit the time to complete a single training to 72 hours.

4.3.5 Results

The experimental results are reported in Table 4.2 (for the chemical datasets) and Table 4.3 (for the social datasets). We notice an interesting trend on the D&D, PROTEINS and ENZYMES datasets, where none of the DGNs are able to improve over the baseline. Conversely, on the NCI1 dataset, the baseline is clearly outperformed: this result suggests that on this dataset, these DGNs architectures are suited to exploit the structural information of the training graphs. This result is reinforced from empirical evidence: in fact, we observed in preliminary trials (not reported here) that an overly-parameterized baseline with 10000 hidden units and no regularization is not able to overfit the NCI1 training data completely, reaching around 67% training accuracy, while a model such as GIN can easily overfit (≈ 100% accuracy) the training data. This indicates that, at least for the NCI1 dataset, the structural information hugely affects the ability to fit the training set. On social datasets, GIN seems to be the most performant model, reaching the best accuracy in three out of five datasets. However, in both chemical and social scenarios, the standard deviations are so large that all judgements about which model is better are speculative. Following, we summarize other relevant findings of our study.

TABLE 4.2: Results on chemical datasets with mean accuracy and standard deviation are reported. Best performances are highlighted in bold.

	D&D	**NCI1**	**PROTEINS**	**ENZYMES**
Baseline	**78.4** ± 4.5	69.8 ± 2.2	**75.8** ± 3.7	**65.2** ± 6.4
DGCNN	76.6 ± 4.3	76.4 ± 1.7	72.9 ± 3.5	38.9 ± 5.7
DiffPool	75.0 ± 3.5	76.9 ± 1.9	73.7 ± 3.5	59.5 ± 5.6
ECC	72.6 ± 4.1	76.2 ± 1.4	72.3 ± 3.4	29.5 ± 8.2
GIN	75.3 ± 2.9	**80.0** ± 1.4	73.3 ± 4.0	59.6 ± 4.5
GraphSAGE	72.9 ± 2.0	76.0 ± 1.8	73.0 ± 4.5	58.2 ± 6.0

The Importance of Baselines Our results confirm that structure-agnostic baselines are an essential tool to evaluate DGNs under a clear perspective, as well as to extract useful insights on whether structure has been exploited. As an example, consider how none of the DGNs surpasses the baseline on D&D, PROTEINS and ENZYMES; based on this result, we argue that the state-of-the-art DGNs we analyzed are not able to fully exploit the structure on such datasets yet. This contrasts with the current literature of chemistry, where structural properties of the molecular graph are strongly believed to correlate with molecular properties (**?**). For this reason, we suggest not to over-emphasize small performance gains on these datasets. Currently, it is more likely that small fluctuations in performances are likely to be caused by other factors, such as random initializations, rather than a successful exploitation of the structure. In conclusion, we warmly recommend DGN practitioners to include baseline comparisons in future works, in order to better characterize the extent of their contributions.

TABLE 4.3: Results on social datasets with mean accuracy and standard deviation are reported. Best performances are highlighted in bold. OOR means Out of Resources, either time (> 72 hours for a single training) or GPU memory.

		IMDB-B	IMDB-M	REDDIT-B	REDDIT-5K	COLLAB
No Features	Baseline	50.7 ± 2.4	36.1 ± 3.0	72.1 ± 7.8	35.1 ± 1.4	55.0 ± 1.9
	DGCNN	53.3 ± 5.0	38.6 ± 2.2	77.1 ± 2.9	35.7 ± 1.8	57.4 ± 1.9
	DiffPool	68.3 ± 6.1	45.1 ± 3.2	76.6 ± 2.4	34.6 ± 2.0	67.7 ± 1.9
	ECC	67.8 ± 4.8	44.8 ± 3.1	OOR	OOR	OOR
	GIN	66.8 ± 3.9	42.2 ± 4.6	$\mathbf{87.0} \pm 4.4$	$\mathbf{53.8} \pm 5.9$	$\mathbf{75.9} \pm 1.9$
	GraphSAGE	$\mathbf{69.9} \pm 4.6$	$\mathbf{47.2} \pm 3.6$	86.1 ± 2.0	49.9 ± 1.7	71.6 ± 1.5
With Degree	Baseline	70.8 ± 5.0	$\mathbf{49.1} \pm 3.5$	82.2 ± 3.0	52.2 ± 1.5	70.2 ± 1.5
	DGCNN	69.2 ± 3.0	45.6 ± 3.4	87.8 ± 2.5	49.2 ± 1.2	71.2 ± 1.9
	DiffPool	68.4 ± 3.3	45.6 ± 3.4	89.1 ± 1.6	53.8 ± 1.4	68.9 ± 2.0
	ECC	67.7 ± 2.8	43.5 ± 3.1	OOR	OOR	OOR
	GIN	$\mathbf{71.2} \pm 3.9$	48.5 ± 3.3	$\mathbf{89.9} \pm 1.9$	$\mathbf{56.1} \pm 1.7$	$\mathbf{75.6} \pm 2.3$
	GraphSAGE	68.8 ± 4.5	47.6 ± 3.5	84.3 ± 1.9	50.0 ± 1.3	73.9 ± 1.7

The Effect of Node Degree Based on our results, adding the node degree to the input features almost always results in a performance improvement, sometimes very strong, on social datasets. For example, adding the degree information to the baseline improves performances of $\approx 15\%$ across all datasets, up to being competitive with the examined DGNs. In particular, the baseline achieves the best performance on IMDB-MULTI, and performs very close to the best model (GIN) on IMDB-BINARY. In contrast, the addition of degree information is less impacting for most DGNs. This result is somewhat expected, since they are supposed to automatically extract the degree information from the structure. One notable exception to this trend is DGCNN, which explicitly needs the addition of node degrees to perform well across all datasets. We observe that the ranking of the models, after the addition of the degrees, drastically changes; this raises the question about the impact of other structural features (such as clustering coefficient) on performances, which we leave to future works. In a further experiment, we reason about whether the degree has an influence on the number of layers that are necessary to solve the task. We investigate the matter by computing the median number of layers of the winning model in each of the 10 different folds. These results are shown in Table 4.4. Given the benefit given by the addition of the node degree feature, we hypothesize that models such as DGCNN learn features correlated to the node degrees in the very first layers; this learned information helps to perform well in the tasks using fewer convolutional layers.

TABLE 4.4: The table displays the median number of selcted layers in relation to the addition of node degrees as input features on all social datasets. 1 indicates that an uninformative feature is used as node label.

	IMDB-B		**IMDB-M**		**REDDIT-B**		**REDDIT-M**		**COLLAB**	
	1	**DEG**	**1**	**DEG**	**1**	**DEG**	**1**	**DEG**	**1**	**DEG**
DGCNN	3	3	3.5	3	4	3	3	2	4	2
DiffPool	1	2	2	1	2	2	2	1	2	1.5
ECC	1	2	1	1	-	-	-	-	-	-
GIN	3	2	4	2	4	4	4	3	4	4
GraphSAGE	4	3	5	4	3	4	3	5	3	5

Comparison with Published Results Figure 4.7 compares the average values of our test results (shown with a square marker) to those reported in the literature (shown with a triangle marker). In addition, we plot the average of our validation results across the 10 different model selections (shown with a circle marker). From the plot, it is clear that our results are in most cases different from published results, and the gap between the two estimates is usually consistent. Moreover, and differently from results in the literature, our average validation accuracies are consistently similar to the test accuracies, which indicates that our estimates are less biased in general. We emphasize that our results are *i*) obtained within the rigorous model selection and evaluation framework; *ii*) fair in terms of how the data was split, and which features have been used for all competitors; *iii*) reproducible[3].

4.3.6 Conclusions

In this section, we showed how a rigorous empirical evaluation of DGNs can help to design better experiments, and to draw more informed conclusions as regards the potential impact of novel architectures. This has been possible by introducing a clear and reproducible environment for benchmarking current and future DGN architectures, as well as with reliable and reproducible results to which DGN practitioners can test novel architectures. This work will hopefully prove useful to researchers and practitioners that want to compare DGNs in a more rigorous way.

[3]Code, hyper-parameters and data splits are available at `https://github.com/diningphil/gnn-comparison`

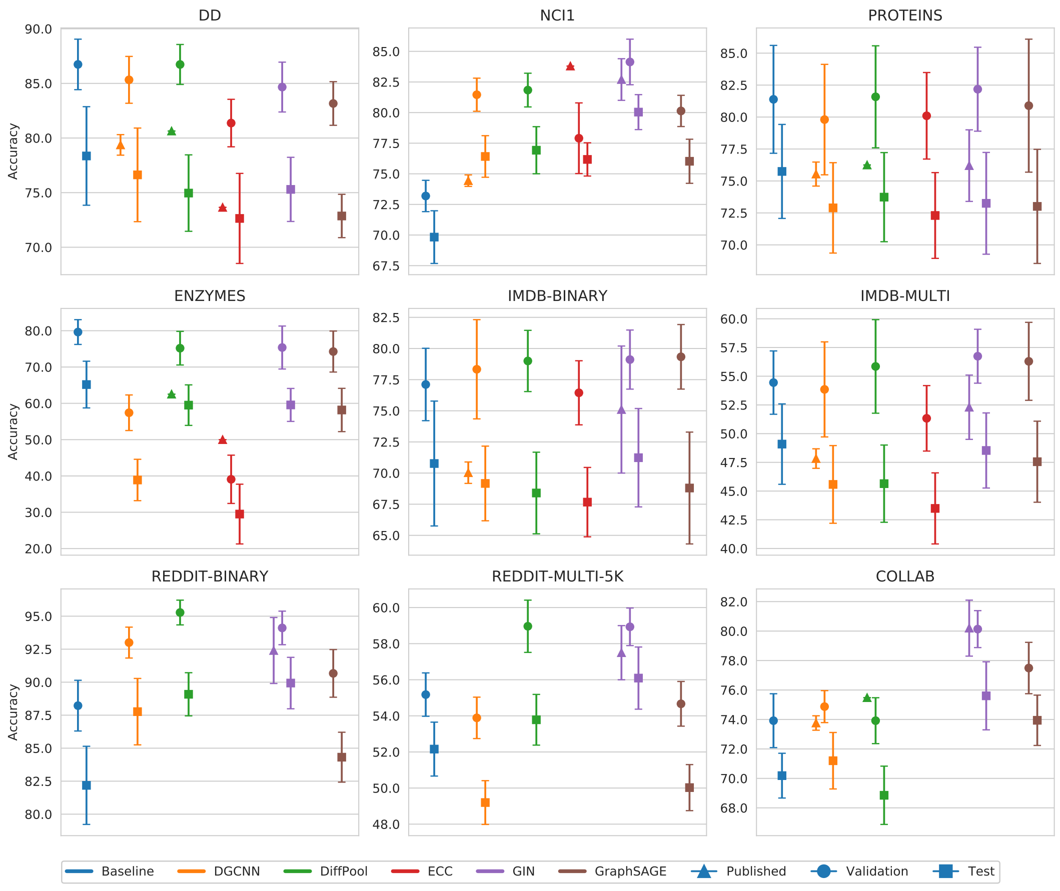


FIGURE 4.7: Results.

Chapter 5

Case Study: Prediction of Dynamical Properties of Biochemical Pathways using Deep Graph Networks

In this chapter, we present an application of Deep Learning techniques on graphs to a life sciences problem related to computational biology. Specifically, we apply Deep Graph Networks to process biochemical pathways, *i.e.* dynamical systems that model the complex interactions between molecules at the biochemical level. Biochemical pathways can be represented as a particular form of bipartite graphs known as Petri networks, which allow to study several properties of such systems. Here, we focus on the property of concentration robustness. To be measured, concentration robustness requires to perform time-expensive simulations. Here, we opt for an approximate but reliable solution, which is orders of magnitude faster to compute. Through processing of the Petri network associated to the biochemical pathway with Deep Graph Networks, we show experimentally that it is possible to build a model that predicts concentration robustness rapidly and with a satisfactory level of accuracy.

5.1 Introduction and Motivation

In order to understand the mechanisms underlying the functioning of living cells, it is necessary to analyze their activities at the biochemical level. Biochemical pathways (or pathways, in short) are complex dynamical systems in which molecules interact with each other through chemical reactions. In these reactions, molecules can take the role of reactant, product, promoter or inhibitor. The dynamics of a pathway are determined by the variation over time of the concentration of its molecules. To study these dynamics, two methodologies are traditionally employed. One consists in modelling the pathway as a system of Ordinary Differential Equations (ODEs), derived from the application of chemical kinetics laws such as the law of mass action. In cases where pathways involve molecules available in small concentrations, which make the dynamics of reactions sensitive to random events, stochastic modelling and simulation approaches are preferred. These are usually variants of the well-known Gillespie's simulation algorithm ?. The use of these modelling tools allows to investigate dynamical properties of biochemical pathways such as the reachability of steady states, the occurrence of oscillatory behaviors, causalities between molecular species, and robustness. However, quantitatively measuring these properties often requires to execute a large number of numerical or stochastic simulation, which in turn are time-consuming and computationally intensive.

Given their nature, one widely used formalism to represent biochemical pathways is that of graphs. Many different graphical notations of pathways exist in the literature (see, e.g., ?), most of which represent molecules as nodes, and reactions as multi-edges or as additional nodes. Using graphs to represent pathways is convenient for three main reasons. Firstly, they provide a quite natural visual representation of the reactions occurring in the pathway. Secondly, they enable the study of the pathway dynamics through methods such as network and structural analysis. Thirdly, graphs can easily be transformed into ODEs or stochastic models, to apply standard numerical simulation techniques.

In this study, we investigate whether predicting dynamical properties of biochemical pathways from the structure of their associated graphs is possible; and if so, to what extent. In other words, our main assumption is that the dynamics of the biological system modeled by the pathway can be correlated to the structural properties of the graph by which it is represented. If the assumption is correct, the positive implications are two-fold: on one hand, a good predictive model of desired biochemical properties could, in principle, replace numerical or stochastic simulations whenever time and computational budgets are limited. On the other hand, it could allow to predict the properties even in cases where the quantitative information is not available, for example whenever numerical or stochastic simulation methods cannot be applied.

The main idea behind this work is to use of Deep Graph Networks to learn structural features of pathways represented as Petri networks (or Petri nets, in short), which are used to predict a property of interest. Here, we focus on the assessment of the dynamical property of robustness, defined as the the ability of a pathway to preserve its dynamics despite the perturbation of some parameters or initial conditions. More specifically, given a pathway and a pair of molecular species (called *input* and *output* species), the robustness measures how much the concentration of the output species at the steady state is influenced by perturbations of the initial concentration of the input species. This is a notion of *concentration robustness* (?) which is to some extent correlated with the notion of global sensitivity (?). Robustness makes up for a perfect candidate to test our approach, as its assessment is time-consuming and computationally intensive, requiring a huge number of simulations to explore the parameters space.

The initial part of this work focuses on the creation of a dataset suited to train the DGN. We start from collecting 706 curated pathway models in SBML format from the BioModels[1] database (?), which were initially converted into Petri nets. For every pathway in this initial dataset, the robustness of every possible pair of input and output species has been computed through ODE-based simulations. Then, these robustness values have been transformed into binary indicators of whether robustness holds for a given pathway and input/output species. Lastly, for each pathway and for each input/output species in that pathway, the induced subgraphs containing the input and output nodes (as well as other nodes that influence the pathway dynamics) have been extracted. To summarize, the final dataset obtained with this preparatory phase consists of a set of subgraphs, each associated to a pair of input/output molecular species, and their respective robustness indicator. The predictive task is thus one of binary classification: specifically, given a subgraph and two nodes corresponding to the input and output species, the model should correctly classify them as robust or not.

We model the task with a DGN to learn structural features from the subgraphs and compute a graph embedding that is passed to a MLP classifier. The performances of the model are assessed according to a rigorous framework similar to the one developed in 4.3.4. Our experimental results show that we are indeed able to predict robustness with reasonable accuracy. We also conduct a follow-up investigation of how the architectural choices, such as type of graph convolutional layer and number of layer, impact performances. The analysis suggests that the depth of the DGN, in terms of number of layers, plays an important role in capturing the right features that correlate the subgraph structure to the robustness, and that deep DGNs perform better at this task.

To our knowledge, this is the first work that addresses the problem of predicting dynamical, properties of pathways on a large scale using Deep Learning. In contrast, other approaches in the literature mainly focus on inferring the parameters of a single pathway, or the relationships between its species. We believe this work has great potential in helping understand the functioning of living cells, by serving as a fast, and computationally friendlier, alternative to performing expensive simulations in the assessment of pathway properties.

5.2 Background

In this section, we provide the necessary formal background to understand the modeling of biochemical pathways with Petri nets, and the dynamical property of concentration robustness.

5.2.1 Pathway Petri Nets

Biochemical pathways are essentially sets of chemical reactions of the form:

$$c_1 S_1 + c_2 S_2 + \ldots \xrightarrow{k} c'_1 P_1 + c'_2 P_2 \ldots,$$

[1] BioModels: https://www.ebi.ac.uk/biomodels/

Reaction	Modifiers	Kinetics
$A + B \to 2B$		$r1 = k_1 AB$
$B \to A$		$r2 = k_2 B$
$C + D \to E$	A	$r3 = k_3 CDA$
$E \to F$		$r4 = k_4 E$
$F \to E$		$r5 = k_5 F$
$G \to H$	F	$r6 = \frac{k_6 G}{1+2F}$
$H \to G$		$r7 = k_7 H$

(A) Reactions

$$\begin{aligned}
\tfrac{dA}{dt} &= -k_1 AB + k_2 B \\
\tfrac{dB}{dt} &= k_1 AB - k_2 B \\
\tfrac{dC}{dt} &= -k_3 CDA \\
\tfrac{dD}{dt} &= -k_3 CDA \\
\tfrac{dE}{dt} &= k_3 CDA - k_4 E + k_5 F \\
\tfrac{dF}{dt} &= k_4 E - k_5 F \\
\tfrac{dG}{dt} &= -\tfrac{k_6 G}{1+2F} + k_7 H \\
\tfrac{dH}{dt} &= \tfrac{k_6 G}{1+2F} - k_7 H
\end{aligned}$$

(B) ODEs

FIGURE 5.1: An example of biochemical pathway. (A): list of reactions with information on modifiers and kinetic formulas. (B): the corresponding system of ODEs.

where S_i, P_i are molecules (*reactants* and *products*, respectively), $c_i, c'_i \in \mathbb{N}$ are *stoichiometric coefficients* expressing the multiplicities of reactants and products involved in the reaction, and $k \in \mathbb{R}_{\geq 0}$ is the *kinetic constant*, used to compute the reaction rate according to standard chemical kinetic laws such as the law of mass action. Besides reactants and products, the reactions of a biochemical pathway often include in their description other molecules, called *modifiers*. These are not consumed nor produced by the reaction, but act either as *promoters* or as *inhibitors*, meaning that they can increase or decrease the reaction rate, respectively. Although these molecules are not listed among reactants and products, they do have a role in the kinetic formula, which no longer follows the mass action principle in this case. For example, in the SBML language (?), a standard XML-based modeling language for biochemical pathways, reactions can be associated with a number of modifiers, whose concentration is used in the kinetic formula of the reaction. In Figure 5.1a we show a table describing the set of reactions describing a biochemical pathway (first column), some of which include a modifier (second column), namely A for the third reaction, and F for the sixth. Each reaction is associated with its kinetic formula (third column), that, for simplicity, we reference through an alias of the form ri with $i = 1, \dots, 7$. Using the kinetic formulas of the two reactions with modifiers as an example, it is clear that A acts as a promoter (meaning that the reaction rate is proportional to the concentration of A) and that F acts as inhibitor (meaning that the reaction rate is inversely proportional to the concentration of F). Kinetic formulas can then be used to construct a system of ODEs as shown in Figure 5.1b.

A common way to represent biochemical pathways is through Petri nets ?. The formalism of Petri nets have been originally proposed for the description and analysis of concurrent systems ?, but has been later adopted to model other kinds of systems, such as biological ones. Several variants of Petri nets have been proposed in the literature. In this work, we consider a version of *continuous* Petri nets ? with promotion and inhibition edges and general kinetic functions. We call this biologically inspired variant Pathway Petri Network (PPN). A PPN is essentially a bipartite graph with two types of nodes and three types of labelled edges. According to standard Petri nets terminology, the two types of nodes are called *places* and *transitions*. The semantics of a PPN in a continuous setting are described by a system of ODEs, with one equation for each place. In the case of pathways, such system corresponds exactly to the one obtained from the chemical reactions shown in Figure 5.1b. The state of a PPN (called *marking*) is then defined as an assignment of positive real values to the variables of the ODEs. We denote with $\mathcal{M}$ the set of all possible markings.

More formally, a PPN can be defined as a tuple $\wp = \langle \mathcal{P}, \mathcal{T}, \mathcal{A}_S, \mathcal{A}_P, \mathcal{A}_I, \vartheta, \varsigma, M_0 \rangle$ where:

- $\mathcal{P}$ and $\mathcal{T}$ are finite, non empty disjoint sets of places and transitions, respectively;
- $\mathcal{A}_S = ((\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a set of standard directed edges;
- $\mathcal{A}_P \subseteq (\mathcal{P} \times \mathcal{T})$ is the set of promotion edges;
- $\mathcal{A}_I \subseteq (\mathcal{P} \times \mathcal{T})$ is the set of inhibition edges;
- $\varsigma : \mathcal{A}_S \to \mathbb{N}^{\geq 0}$ weights every standard edge by non-negative integer values;
- $\varsigma : \mathcal{T} \to (\mathcal{M} \to \mathbb{R}^{\geq 0})$, is a function that assigns, to each transition, a function that computes a kinetic formula to every possible marking $M \in \mathcal{M}$;
- $M_0 \in \mathcal{M}$ is the initial marking.

A visual representation of the PPN corresponding to the pathway in Figure 5.1a is shown in Figure 5.2. The sets of places $\mathcal{P}$ and transitions $\mathcal{T}$ of a pathway Petri net represent molecular species and reactants, and are displayed as circles and rectangles, respectively. In the figure, places are labeled with the name of the corresponding molecule. The directed edges, depicted as standard arrows, connect reactants to reactions and reactions to products. The weights of the edges (omitted if equal to one) correspond to the stoichiometric coefficients of reactant/product pairs. The sets of promotion and inhibition edges, $\mathcal{A}_P$ and $\mathcal{A}_I$, connect molecules to the reactions they promote or inhibit, respectively, and they are displayed as dotted or T-shaped arrows, respectively. The kinetic formulas of reactions (or rather, their aliases defined as in Figure 5.2), are shown inside the rectangles of the corresponding transitions. As explained previously, molecules connected through promotion edges give a positive contribution to the value of the kinetic formula, while molecules connected through inhibition edges give a negative (inversely proportional) contribution. Finally, the initial marking M_0 is not shown in the figure, and it has to be described separately.

In this work, we use a variant of PPNs where all the information unrelated to the structure of the pathway is discarded. Specifically, we ignore information about:

- kinetic formulas;
- multiplicities of reactants and products (*i.e.* edge labels);

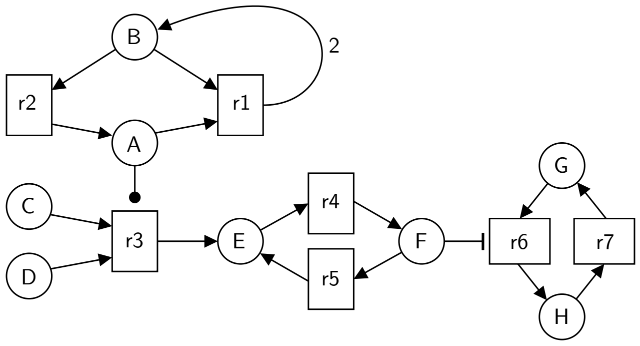


FIGURE 5.2: The Pathway Petri Net corresponding to the reactions described in Figure 5.1a

- the initial marking M_0.

Basically, for our purposes, a PPN is a tuple $\wp = \langle \mathcal{P}, \mathcal{T}, \mathcal{A}_S, \mathcal{A}_P, \mathcal{A}_I \rangle$ where the irrelevant components have been omitted. We rewrite this object, according to the graph notation introduced in Section 3.1, into a **pathway graph** $G = \langle \mathcal{V}_G, \mathcal{E}_G \rangle$. To do so, we first define the following nodes and edges subsets:

- $\mathcal{V}_G^{\text{mol}} = \mathcal{P}$ is the set of molecules;
- $\mathcal{V}_G^{\text{rx}} = \mathcal{T}$ is the set of reactions;
- $\mathcal{E}_G^{\text{std}} = \mathcal{A}_S$ is the set of standard edges;
- $\mathcal{E}_G^{\text{pro}} = \mathcal{A}_P$ is the set of promoter edges;
- $\mathcal{E}_G^{\text{inh}} = \mathcal{A}_I$ is the set of inhibitor edges,

where $\mathcal{V}_G^{\text{mol}} \bigcap \mathcal{V}_G^{\text{rx}} = \emptyset$ and $\mathcal{E}_G^{\text{std}} \bigcap \mathcal{E}_G^{\text{pro}} \bigcap \mathcal{E}_G^{\text{inh}} = \emptyset$. Then, we simply set $\mathcal{V}_G = \mathcal{V}_G^{\text{mol}} \bigcup \mathcal{V}_G^{\text{rx}}$ and $\mathcal{E}_G = \mathcal{E}_G^{\text{std}} \bigcup \mathcal{E}_G^{\text{pro}} \bigcup \mathcal{E}_G^{\text{inh}}$. Using the biochemical pathway of Figure 5.1 as reference, its associated pathway graph is shown in Figure 5.3. More explicitly,

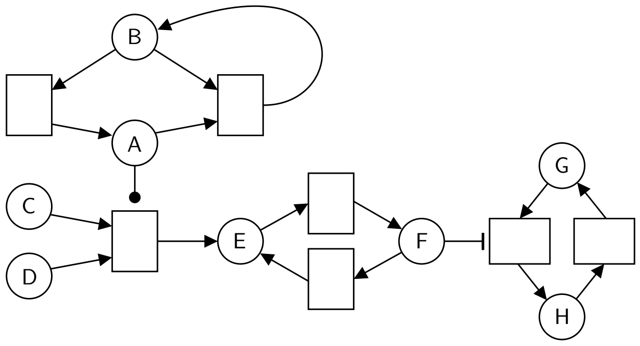


FIGURE 5.3: The pathway graph obtained by omitting kinetic formulas and edge labels from the one in Figure 5.2. Notice that the names of the molecules are displayed for visual aid, but are not included of the pathway graph.

the set of nodes of the pathway graph contains both molecules and reactions. The set of edges of the graph contains an edge (of any type) from a molecule to a reaction if and only if a perturbation in the concentration of the molecule determines a change in the reaction rate (that could in principle be computed through the omitted kinetic formula). Similarly, it contains an edge from a reaction to a molecule if and only if a perturbation in the reaction rate determines a change in the dynamics of the concentration of that molecule. This is intuitive for those molecular species that are products, as the dynamics of the product accumulation is determined by the reaction rate. By construction, a pathway graph is bipartite.

5.2.2 Concentration Robustness

Robustness is defined as the ability of a system to maintain its functionalities again external and internal perturbations **?**, a property observed in many biological systems. A general notion of robustness has been formalized by Kitano in **?**. This formalization considers a specific functionality of a system and the *viability* of such functionality, defined as the ability of the system (*e.g.* a cell) to carry it out. This could be expressed, for instance, in terms of the synthesis/degradation rate or concentration level of some target substance, in terms of cell growth rate, or in terms of other suitable quantitative indicators. More specifically, according to Kitano's definition, the robustness R of a system $\mathbb{S}$, with respect to a specific functionality a and against a set of perturbations P is expressed as:

$$R^{\mathbb{S}}_{a,P} = \int_p \Psi(p) D^{\mathbb{S}}_a(p)dp$$

In the above definition, $\Psi(p)$ is the probability a perturbation p, and $D_a(p)$ evaluates the functionality a of the system $\mathbb{S}$ under the perturbation p. More precisely, function $D_a(p)$ gives the viability of a under perturbation p, in relation to the viability of a under normal conditions. In the absence of perturbations, $D_a(p) = 1$, meaning that the functionality a is assumed to be carried out in an optimal way, or equivalently, that perturbations have irrelevant or no influence; conversely, $D_a(p) = 0$ if perturbations cause the system to fail completely in performing a, and $0 < D_a(p) < 1$ in the case of relevant perturbations.

An improvement to Kitano's formulation of robustness has been proposed in **?**, where functionalities to be maintained are described as linear temporal logic (LTL) formulas. In this formulation, the impact of perturbations is quantified through a notion of *violation degree*, which measures the distance between the dynamics of the perturbed system and the LTL formula. Many more specific definitions exist, differing either in the class of biological systems they apply to, or in the way the functionality to be maintained is expressed **?**.

In the case of biochemical pathways, a common formulation of robustness can be expressed in terms of maintenance of the concentration levels of some species. This definition can be reduced to both general formulations in **?** and **?**. In particular, the *absolute concentration robustness* proposed in **?** is based on the comparison of the concentration level of given species at the steady state, against perturbations (either in the kinetic parameters or in the initial concentrations) of some other species.

A generalization of absolute concentration robustness, called *α-robustness*, has been proposed in **?**, where concentration intervals are introduced both for the perturbed molecules (input species) and for the molecules whose concentration is maintained (output species). Informally speaking, a biochemical pathway is α-robust with respect to a given set of initial concentration intervals, if the concentration of a chosen output molecule at the steady state lies in the interval $[k - \alpha/2, k + \alpha/2]$ for some $k \in \mathbb{R}$. A relative version of α-robustness can be obtained simply by dividing α by k. This notion of α-robustness is related to the notion of global sensitivity **?**, which typically measures the average effect of a set of perturbations. Hereafter, we use the term robustness to specifically refer to *α-robustness* for brevity.

The assessment of robustness usually requires to perform exhaustive numerical simulations in parameter space **?**; **?** (where by parameters we intend mainly the kinetic parameters, or the initial concentrations). In some particular cases, one can exploit the biological network structure to avoid performing simulations altogether **?**. Moreover, in cases where the dynamics of the network are monotonic, the number of such simulations can be significantly reduced **?**.

5.3 Methods

Here, we provide details about how the raw biological pathways have been converted into the dataset of graphs on which the DGN has been trained.

5.3.1 Subgraphs Extraction

As explained in Section 5.2.2, concentration robustness is defined in terms of pathway, and a pair of input and output molecules. However, for a fixed choice of input and output, not all the nodes in a pathway graph contribute to the assessment, but only a specific subset corresponding to an induced subgraph. Given a pathway graph G and an input/output node pair $\mathsf{I}, \mathsf{O} \in \mathcal{V}_G^{\text{mol}}$ with $\mathsf{I} \neq \mathsf{O}$, we call this subgraph the *subgraph of G induced by the input/output pair* (I, O). In practice, this induced subgraph allows to focus on the graph portion that is relevant for the assessment of the property, by removing the nodes irrelevant for its computation. Before delving into details, let us first introduce a helper data structure which we call **enriched pathway graph**. Given a pathway graph G, its enriched version G' is defined as follows: initially, $\mathcal{V}_{G'} = \mathcal{V}_G$, $\mathcal{E}_{G'} = \mathcal{E}_G$. Then, for every standard edge $(u, v) \in \mathcal{E}_G^{\text{std}}$ where $u \in \mathcal{V}_G^{\text{mol}}$ and $v \in \mathcal{E}_G^{\text{rx}}$, we update the standard edges of G' with a reverse standard edge (v, u), setting $\mathcal{E}_{G'}^{\text{std}} = \mathcal{E}_{G'}^{\text{std}} \bigcup (v, u)$. Note that we do not reverse neither standard edges from reactions to molecules, nor promotion and inhibition edges. The enriched pathway graph obtained from the pathway graph of Figure 5.3 is shown in Figure 5.4, where the additional edges are drawn in solid black. Such graph represents influence relationships between molecules and reactions. The reversed edges encode the fact that a perturbation in the reaction rates determines a variation in the reactants consumption. Hence, the enriched pathway graph essentially corresponds to the *influence graph* that could be computed from the Jacobian matrix containing the partial derivatives of the system of ODEs associated to the pathway ?. The purpose of the enriched pathway graph is to

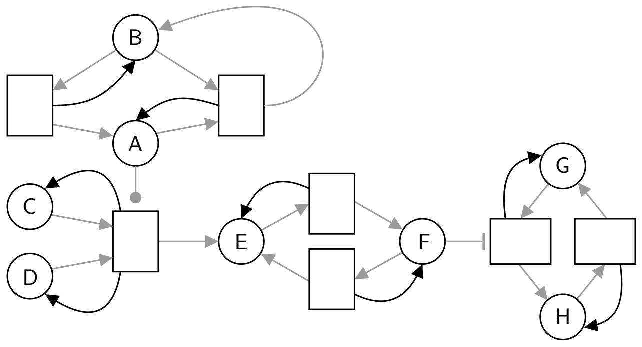


FIGURE 5.4: The enriched pathway graph obtained from the pathway graph of Figure 5.3. The edges added with respect to the original graph are shown in black.

determine which portion of the associated pathway graph is relevant for the assessment of the robustness. With the help of the enriched graph G', we can introduce the subgraph of G induced by an input pair (I, O) as a graph $S_{\mathsf{I,O}} = \langle \mathcal{V}_{S_{\mathsf{I,O}}}, \mathcal{E}_{S_{\mathsf{I,O}}} \rangle$, defined informally as follows: $S_{\mathsf{I,O}}$ is the smallest subgraph of G whose node set contains I, O, as well as nodes in every possible oriented path from I to O in G'. We remark that $S_{\mathsf{I,O}}$ is a subgraph of G, although its node set is computed on the basis of the paths in G'. Figure 5.5 shows some examples of induced subgraphs extracted from the graph in Figure 5.3.

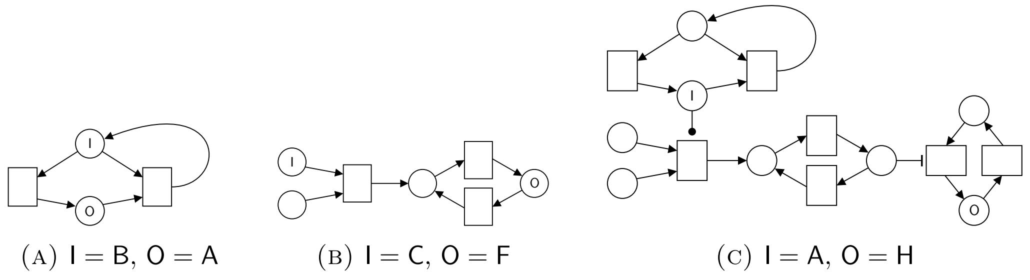


FIGURE 5.5: Examples of subgraphs of the pathway graph in Figure 5.3 induced by different input/output node pairs (I, O).

5.3.2 Robustness Computation

The robustness values used to label the induced subgraphs are computed following the relative α-robustness approach. The dynamics of each biochemical pathway has been simulated by applying a numerical solver (the `libRoadRunner` Python library) to its ODEs representation. Reference initial concentrations of involved molecules have been obtained from the original SBML model of each pathway. Moreover, 100 simulations have been performed for each molecule, by perturbing its initial concentration in the range $[-20\%, +20\%]$. The termination of each simulation has been set to the achievement of the steady state, with a timeout of 250 simulated time units.[2] Given a subgraph $S_{\mathsf{I},\mathsf{O}}$, we computed the width α of the range of concentrations reached by the output molecules O by varying the input I (as per definition of α-robustness). A relative robustness $\overline{\alpha}$ has then been obtained by dividing α by the concentration reached by the output when the initial concentration of the input is the reference one (no perturbation). The final robustness value $r_{\mathsf{I},\mathsf{O}} \in [0, 1]$ has been computed by comparing $\overline{\alpha}$ (a relative representation of the output range) with 0.4 (a relative representation of the initial input range, that is 40%) as follows:

$$r_{\mathsf{I},\mathsf{O}} = 1 - min(1, \frac{\overline{\alpha}}{0.4})$$

5.3.3 Data Preprocessing

Our initial data collection consisted of 706 SBML models of biochemical pathways, downloaded from the BioModels database **?**. Specifically, the data correspond to the complete set of manually curated models present in the database at the time we started the construction of the dataset[3]. We removed empty models (not containing any node) and discarded duplicates, reducing the number of SBML models to 484. These models were first transformed into PPNs representations and saved in DOT format[4]. For the translation of the SMBL models into PPNs, we developed a Python script that, for each reaction in the SMBL model extracts reactants, products and modifiers. Furthermore, it also checks the kinetic formula in order to determine whether each modifier is either promoter or an inhibitor. With the conversion of each PPN into a pathway graph, we obtained a collection $\mathcal{G} = \{G_i\}_{i=1}^{484}$ of pathway graphs. Each of these pathway graphs has been transformed into a set containing one induced subgraph (whose size does not exceed 100 nodes, for computational reasons) for every possible input/output combination of pathway graph nodes representing molecules, according to the procedure detailed in Section 5.3.1. For each induced subgraph, the associated robustness has been calculated as explained in Section 5.3.2. The result of this preprocessing is a training dataset:

$$\mathbb{G} = \bigcup_{G \in \mathcal{G}} \{(S_{\mathsf{I},\mathsf{O}}, \overline{r}_{\mathsf{I},\mathsf{O}}) \mid \forall (\mathsf{I}, \mathsf{O}) \in \mathcal{V}_G^{\text{mol}}\},$$

where $\overline{r}_{\mathsf{I},\mathsf{O}} = \mathbb{I}[r_{\mathsf{I},\mathsf{O}} > 0.5]$ are robustness indicators used as the labels for the associated binary classification task. The total number of subgraphs in the preprocessed dataset is 44928.

[2]The concentration values obtained at the end of the simulation are considered as steady state values also in the cases in which a timeout has been reached.

[3]May 2019.

[4]The DOT graph description language specification, available at: `https://graphviz.gitlab.io/_pages/doc/info/lang.html`

5.3.4 Subgraph Features

We encoded information such as node type, edge type, and input/output pair of the induced subgraphs in their node and feature vectors. Specifically, for each subgraph node, we associate a binary 3-dimensional feature vector which encodes whether the node is a molecule or a reaction, whether the node is an input node or not, and whether a node. Given an induced subgraph $S_{\mathsf{I},\mathsf{O}}$ and one of its nodes $v \in \mathcal{V}_{S_{\mathsf{I},\mathsf{O}}}$, we define its 3-dimensional vector of node features $\mathbf{x}_{[v]} \in \mathbf{x}_{S_{\mathsf{I},\mathsf{O}}}$ component by component as follows:

$$\mathbf{x}_{[v,1]} = \begin{cases} 1 & \text{if } v \in \mathcal{V}_G^{\text{mol}} \\ 0 & \text{if } v \in \mathcal{V}_G^{\text{rx}}, \end{cases} \qquad \mathbf{x}_{[v,2]} = \begin{cases} 1 & \text{if } v = \mathsf{I} \\ 0 & \text{otherwise}, \end{cases} \qquad \mathbf{x}_{[v,3]} = \begin{cases} 1 & \text{if } v = \mathsf{O} \\ 0 & \text{otherwise}, \end{cases}$$

where the notation $\mathbf{x}_{[v,i]}$ indicates the i-th component of the node feature vector $\mathbf{x}_{[v]}$. Similarly, for each subgraph edge, we encode its edge type as a one-hot vector out of the three possibilities (standard, promoter or inhibitor). Given an edge $(u,v) \in \mathcal{E}_{S(\mathsf{I},\mathsf{O})}$, we define its 3-dimensional vector of edge features $\mathbf{e}_{[u,v]} \in \mathbf{e}_{S_{\mathsf{I},\mathsf{O}}}$ component by component as follows:

$$\mathbf{e}_{[u,v,1]} = \mathbb{I}[(u,v) \in \mathcal{E}_G^{\text{std}}], \quad \mathbf{e}_{[u,v,2]} = \mathbb{I}[(u,v) \in \mathcal{E}_G^{\text{pro}}], \quad \mathbf{e}_{[u,v,3]} = \mathbb{I}[(u,v) \in \mathcal{E}_G^{\text{inh}}],$$

where $\mathbb{I}$ is the indicator function, and the notation $\mathbf{e}_{[u,v,i]}$ identifies the i-th component of the edge feature vector $\mathbf{e}_{[u,v]}$. In practice, the edge features encode the edge type as a one-hot vector. Figure 5.6 shows the induced subgraph of Figure 5.5c with the corresponding feature vectors in place of its nodes and edges.

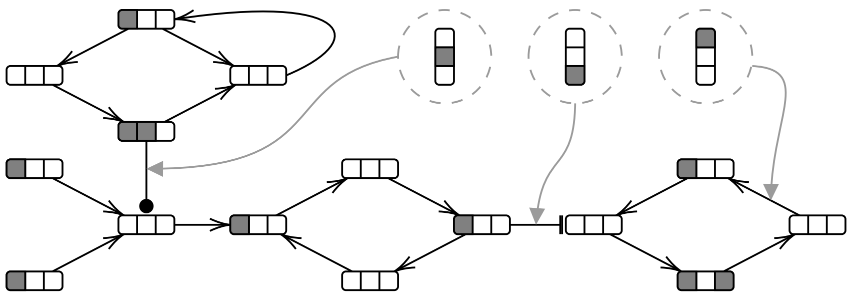


FIGURE 5.6: A representation of the node and edge feature vectors of the induced subgraph shown in Figure 5.5c. Each node is represented as a 3-dimensional binary vector, where 1 is indicated with dark grey and 0 with white. To avoid visual cluttering, we only show one example of edge feature vector per edge type (the column vectors inside the dashed circles).

5.4 Experiments

In this section, we provide all the necessary details concerning our experimental procedures. In particular, we describe the architecture of the DGN in detail, and discuss the assessment protocol by which we evaluated the proposed model on the predictive task.

The proposed DGN architecture for robustness prediction, shown at a high level in Figure 5.7, consists in a series of L graph convolutional layers, followed by a node readout that aggregates the hidden states at each layer into node representations by concatenation, a graph readout that aggregates the node representations into a graph representation, and a downstream MLP classifier which uses the graph representation to compute a probability of whether the graph is robust or not. Some

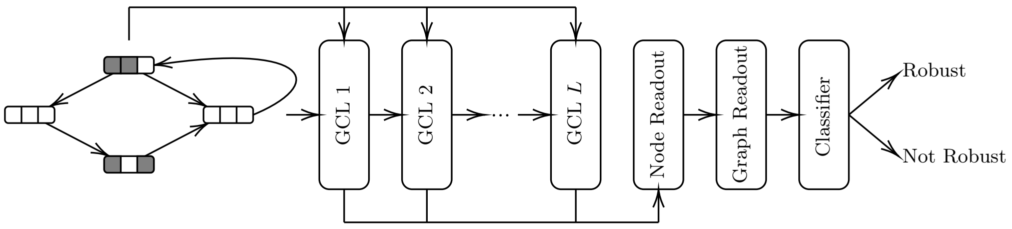


FIGURE 5.7: Model Architecture.

hyper-parameters of the network are fixed before-hand: the downstream MLP used for classification has two hidden layers of size 128 and 64, respectively, interleaved by ReLU non-linearities. As output layer, since our interest is to compute robustness probabilities, we used a sigmoid function that maps its input to the $(0, 1)$ range. All models are trained with MLE using SGD, by minimizing the BCE loss function as usual in binary classification tasks. For the minimization, we used the Adam optimizer, with a learning rate of 0.001 and scheduled learning rate annealing with a shrinking factor of 0.6 every 50 epochs. To prevent overfitting, we use two strategies: the hidden layers are regularized via Dropout, with drop probability of 0.1; and we used early stopping with a maximum number of epochs of 500, and patience parameter of 100 epochs.

Since this is the first time DGNs are applied to this task, we also report a baseline, which is simply a model that always predict the most frequent class (robust, in this case). Its purpose is to serve as reference point to understand whether the task is being learned to some extent or not.

Our experiments are divided in two sequential phases. The experimental protocol is detailed at a high level as follows:

- in a first phase, which we abbreviate as **E1**, we performed model assessment on a smaller dataset of induced subgraphs. The rationale of this phase is to get a sense of which architectural choices, *e.g.* with respect to the type of GCL, are most promising to carry out the full-fledged evaluation on the entire dataset;
- in the second phase, termed **E2**, we perform a final evaluation of the architecture on the full dataset, fixing some architectural components guided by the results obtained in **E1**.

In both experiments, the evaluation procedure consists of an external 5-fold CV for model evaluation, with an internal hold-out split of 90% training and 10% validation for model selection. After a set of hyper-parameters is selected by the inner model selection, the winning configuration is trained and evaluated 3 times in the corresponding test fold, to mitigate the effect of random initialization. The score of the model for that fold is given by the average of these 3 trials. Following, we detail about the hyper-parameters and the evaluation metrics used in both experiments.

5.4.1 E1 Setup

As explained before, the first evaluation of the model is carried out a subset of induced subgraphs, speficially those with number of nodes $<= 40$. This resulted in a dataset with a total of 7036 induced subgraphs, which we term $\mathbb{G}_{\text{small}}$. In this phase, we evaluate the model selecting among the following hyper-parameters:

- number of GCL layers L, choosing between 1 and 8;
- hidden state size h, which once chosen remains fixed across all the L layers, choosing between 128 and 64;
- type of GCL;
- whether the GCL handles the contribution of the different edge types or not;
- type of graph readout function, choosing between sum, mean and max.

Specifically to the third point, we evaluate the following GCL variants:

- Graph Convolutional Network (GCN), as described in Section 4.2.1;
- Graph Isomorphism Network (GIN), as described in Section 4.3.2;
- Weifeiler Lehman Graph Convolution (WLGCN), *i.e.* the GCL layer presented in **?**, whose implementation is the following:

  $$\mathbf{h}^{\ell}_{[v]} = \text{ReLU}\left(\mathbf{W}_{\ell}^{\top}\mathbf{h}^{\ell-1}_{[v]} + \mathbf{U}_{\ell}^{\top}\sum_{u\in\mathcal{N}(v)} e_{uv}\,\mathbf{h}^{\ell-1}_{[u]}\right),$$

  where $e_{uv} \in \mathbb{R}$ is a learned scalar that weighs the connections between the current node and its neighbors.

For each considered GCL, we evaluate a vanilla variant, which does not take into account the different edge types, as well as an edge-aware variant as described in Section 4.2.1, using three different weight matrices (one for each edge type) for the neighborhood aggregation. More specifically, given an induced subgraph G[5] and one of its nodes $v \in \mathcal{V}_G$, the edge-aware neighborhood function used to select neighbors of a certain type is the following:

$$\mathcal{N}_c(v) = \{u \in \mathcal{N}(v) \mid \mathbb{I}[(u, v) \in \mathcal{E}^c_G]\},$$

[5]For consistency, we slightly change notation and refer to induced subgraphs with the letter G instead of S from now on.

where $c \in \mathcal{C} = \{\text{std}, \text{pro}, \text{inh}\}$.

Evaluation Metrics As performances metric, we used accuracy on the predictions, defined as usual as the number of correct predictions out of the total number of predictions. We remark that the model outputs probabilities; hence, we round the prediction to the nearest integer (which is either 0 or 1) to compute a "hard" prediction, which is used in the accuracy calculations. Formally, if:

- TP is the number of true positives, *i.e.* the number of correctly predicted robust subgraphs;
- TN is the number of true negatives, *i.e.* the number of correctly predicted not robust subgraphs;
- FP is the number of false positive, *i.e.* the number of subgraphs wrongly predicted as robust;
- FN is the number of false negatives, *i.e.* the number of subgraphs wrongly predicted as not robust,

the accuracy is defined as:

$$\text{ACC} = \frac{TP + TN}{TP + TN + FP + FN}.$$

5.4.2 E2 Setup

In the second experiment, we fix the GCL (and the fact that it handles edge types or not) to the one that obtains the best evaluation score in **E1**. The other hyper-parameters evaluated are the same as in **E1**, though the selection is now performed on the full dataset $\mathbb{G}$.

Evaluation Metrics In this experiments, there is a high imbalance in favor of the positive class ($\approx$ 90%) with respect to the negative class in the dataset, which is not so dramatic for the small dataset ($\approx$ 70% in favor of the positive class). When the disproportion is so high, in fact, the accuracy can be misleading. Thus, besides accuracy, we evaluate the performances of the model using other metrics related to the accuracy such as:

- Sensitivity, also known as True Positive Rate (TPR), defined as $\frac{TP}{TP + FN}$, which intuitively measures how good is the classifier to detect the positive (robust) class;
- Specificity, defined as $\frac{TN}{TN + FP}$, which intuitively measures how good is the classifier to detect the negative (not robust) class.

Another metric we evaluated, which is not related to the accuracy, is the Area under the Receiver Operating Characteristics curve (AUROC), which quantifies the ability of the classifier to discriminate between negative and positive examples. The AUROC is computed as the integral of a ROC curve, which measures how the TPR and the False Positive Rate FPR $= 1 - \text{TPR}$ vary as one moves a threshold parameter β. It is drawn as a curve plot where the x-axis represents the false positive rate, and the y-axis represents the true positive rate, both with values between 0 and 1. A point of the curve has thus coordinates $(\text{FPR}(\beta), \text{TPR}(\beta))$, for a fixed value of β, with point (0; 0) being associated with $\beta = 1$ and point (1; 1) being associated with $\beta = 0$. The point (0; 1) is the optimum of the curve, since the false positive rate is minimized and the true positive rate is maximized. An example of ROC curve is shown in Figure 5.8.

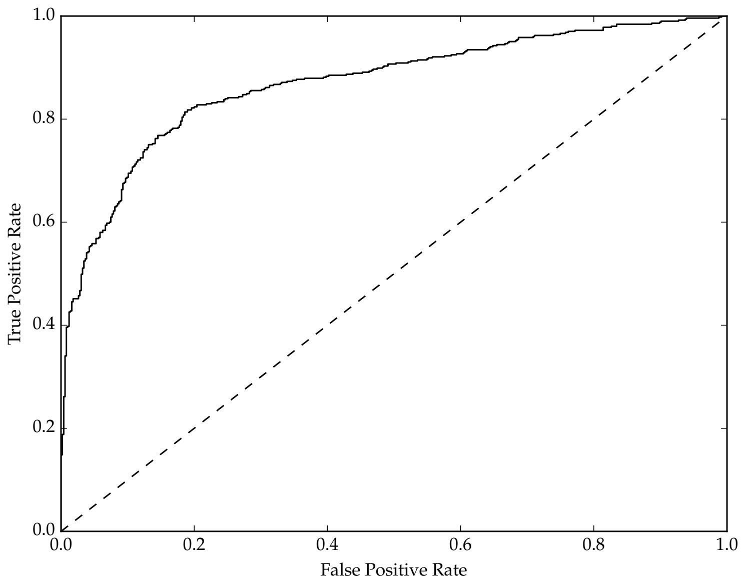


FIGURE 5.8: An example of ROC curve. The dashed line is the ROC curve of a random classifier.

To illustrate the behavior of the curve, it is useful to consider three extreme cases:

1. for a perfect classifier, which makes no errors, the curve goes from point $(0; 0)$, to point $(1; 0)$, to point $(1; 1)$;

2. for a completely random classifier, which whatever the threshold always produces the same number of false positives and true positives, the curve created is the line segment bounded by the points $(0; 0)$ and $(1; 1)$;

3. no reasonable classifier is assumed to produce a ROC curve whose points are located under the $(0; 0)$, $(1; 1)$ segment.

An AUROC value of 0.5 describes a completely random classifier (since it is equal to the area of the triangle described by the extreme case 2), while an AUROC of 1 describes a perfect classifier (since it is equal to the area of the square described by the extreme case 1). Good classifiers have an AUROC score generally equal or higher than 0.8. Besides being a metric independent of class proportions, the AUROC also offers a probabilistic interpretation of its value, since it is equals to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. For an extensive survey of ROC curves, see for example **?**.

5.5 Results

Here, we present the results obtained on the two experiments, **E1** and **E2**. Subsequently, we present an analysis of the performances obtained by the model on two use cases: one on synthetic data, and one on a real-world pathway.

5.5.1 Results on the E1 Experiment

Table 5.1 shows the average accuracy on the 5 test folds obtained by each of the examined architectures on the **E1** experiment. Even though these results are obtained on a dataset approximately 15% the size of the original one (consisting of 7036 induced subgraphs), we can already make some interesting observations about the task and the models:

- the task can be learned, as evidenced by the gap between the baseline and all the examined models, which is approximately 13% on average. This result verifies our initial assumption that it is indeed possible to predict robustness using only pathway structure;
- the comparison between the various GCLs highlights the fact that there is no clear winner between the three tested architectures, which obtain very similar results when assessed on the same experimental conditions (*i.e.* all models with/withour edge handling capabilities);
- as expected, adding edge handling to the GCLs is beneficial to improve performances. On average, the models obtain a 1.8% improvement in accuracy when edge handling is used.

In a second experiment, we assess whether depth (intendended as number of DGN layers) is a good inductive for this task.To do so, we perform a *post-hoc* study the validation scores, to see how they relate to number of DGN layers. Specifically, for each model, and for each number of layers from 1 to 8, we compute the average validation score obtained by all hyper-parameters configurations with an identical number of layers. Note that, although validation accuracy is in general an over-estimate of the true accuracy, the relative difference in performance as the number of layers change stays proportional independently of which specific data is used. In other words, we are not interested in the score by itself, but rather to the trend (increase or decrease) in performance in relation to the number of DGN layers. Figure 5.9 clearly shows that, in all cases, increasing the number of layers improves accuracy, up to a certain depth

TABLE 5.1: Results of the 5-fold CV evaluation of different DGN architectures on the **E1** experiment, as explained in Section 5.4. The suffix "-vanilla" indicates that the corresponding graph convolutional layer does not handle different edge contributions.

Model	**Test Accuracy**
Baseline	0.7322 ± 0.0000
GCN-vanilla	0.8573 ± 0.0087
GIN-vanilla	0.8567 ± 0.0137
WLGCN-vanilla	0.8624 ± 0.0088
GCN	0.8692 ± 0.0140
GIN	0.8684 ± 0.0078
WLGCN	0.8687 ± 0.0117

(around 4-5 layers on average) where performances start to plateau. This provides evidence that depth is a good inductive bias for DGNs on this task, up to some extent.

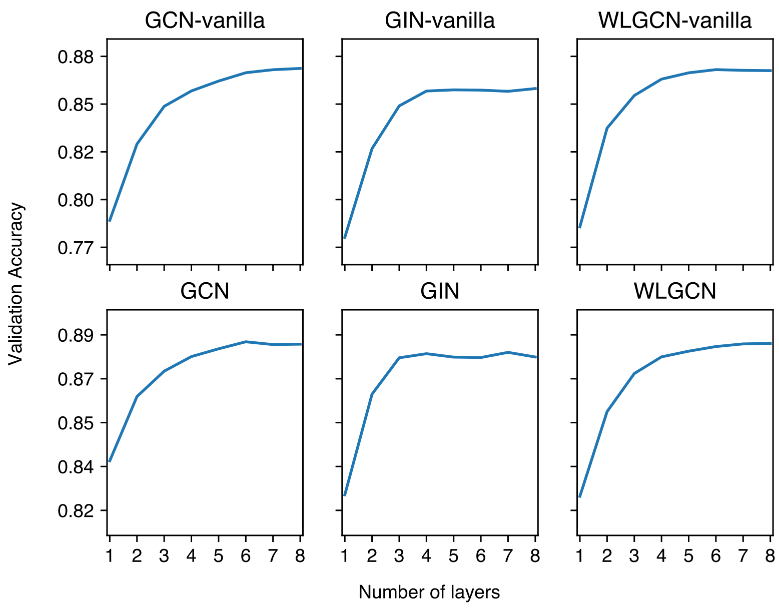


FIGURE 5.9: The effect of the number of DGN layers to the performances of the models.

5.5.2 Results on the E2 Experiments

For the second round of experiments, we fix the GCL to the GCN model with edge-handling capabilities, since it is the model that obtains the highest accuracy in **E1**. The results of our experiments are reported in Table 5.2, where we average the computed metrics across the 5 test folds as usual. In this case, we stratify the performances by number of nodes, in order to also analyze the model performances in relation to

TABLE 5.2: Results of the 5-fold CV evaluation on various performance metrics. We report the global results (on the last row) as well as the results stratified by number of nodes per subgraph. For each stratification, we also report the related support (*i.e.* the average number of graphs in the strata).

Strata	**Support**	**Accuracy**	**Sensitivity**	**Specificity**	**AUROC**
1-10	243 ± 19	0.729 ± 0.020	0.851 ± 0.073	0.526 ± 0.113	0.820 ± 0.034
11-20	711 ± 30	0.843 ± 0.006	0.919 ± 0.021	0.629 ± 0.050	0.892 ± 0.008
21-30	526 ± 19	0.921 ± 0.008	0.969 ± 0.010	0.740 ± 0.052	0.954 ± 0.015
31-40	967 ± 27	0.889 ± 0.012	0.937 ± 0.005	0.757 ± 0.040	0.950 ± 0.009
41-50	1512 ± 21	0.928 ± 0.004	0.970 ± 0.008	0.635 ± 0.064	0.944 ± 0.011
51-60	1679 ± 28	0.921 ± 0.004	0.971 ± 0.006	0.588 ± 0.038	0.950 ± 0.005
61-70	1439 ± 23	0.947 ± 0.005	0.982 ± 0.006	0.644 ± 0.058	0.967 ± 0.005
71-80	1159 ± 28	0.941 ± 0.006	0.980 ± 0.010	0.712 ± 0.087	0.972 ± 0.007
81-90	372 ± 20	0.957 ± 0.008	0.998 ± 0.002	0.037 ± 0.075	0.925 ± 0.010
91-100	378 ± 14	0.850 ± 0.017	0.964 ± 0.024	0.536 ± 0.061	0.888 ± 0.026
Overall	$\mathbf{8985} \pm 1$	$\mathbf{0.913} \pm 0.003$	$\mathbf{0.965} \pm 0.006$	$\mathbf{0.646} \pm 0.042$	$\mathbf{0.948} \pm 0.004$

the size of the input subgraph. From the results, one can immediately see, by looking at the last row, how the model accurately predicts robustness better than all models tested in the **E1** phase by a large margin: this is probably caused by the largest size of the dataset, which usually results in major improvements with any ML model, and in particular with DL models. Specifically, we report an overall accuracy of 0.913 ± 0.003, as well as an AUROC of 0.948 ± 0.004. The model shows very high sensitivity (0.965 ± 0.006) but a low specificity in comparison (0.646 ± 0.042); this indicates that it is “harder” for the model to predict induced subgraphs that are not robust. This effect is a probable consequence of the class misproportion between negative and positive examples, which is around 86% in favor of the positive class for this data sample. One result that is consistent across all measurements are the the very narrow standard deviations of the estimates, which indicate stable predictions regardless of the specific folds on which they are computed. To display this trend visually, we plot in Figure 5.11a the ROC curves obtained on the 5 test folds. Their similarity strongly indicates that the model performances are consistent across different test samples. The results of Table 5.2 show a good performance of the model under the several stratifications tested. In particular, it performs better when dealing on subgraphs with 21-80 nodes, reaching an average in that strata AUROC of over 0.955. To better visualize this trend, we plot in Figure 5.10 the rolling accuracy of the model, using a window size of 20, and averaging across the 5 test folds as usual. The plot clearly shows the improvement in accuracy as the number of nodes increases. Interestingly, when the size of the graph exceeds 80 nodes, the model performances start to decrease. This might be a consequence of the smaller sample sizes of subgraph with 81-100 nodes which occur in the dataset 3 times less on average than subgraphs with 21-80 nodes. The same trend can be noticed for smaller subgraphs, with up to

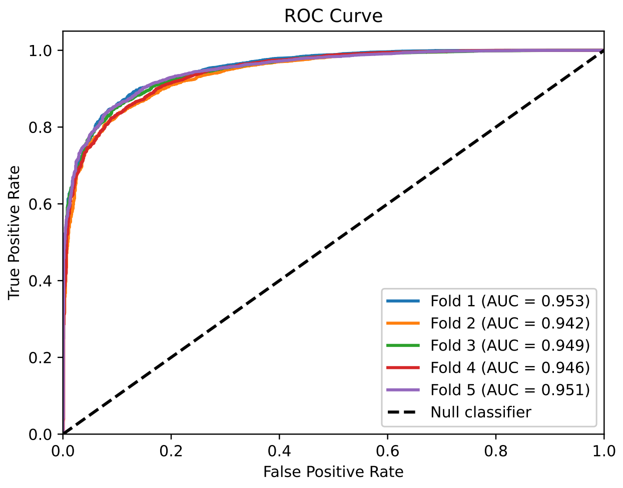
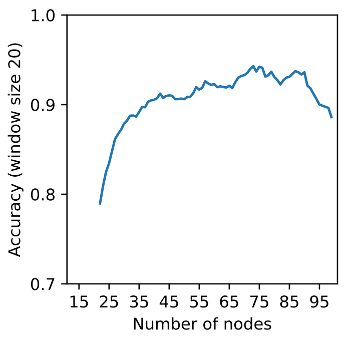
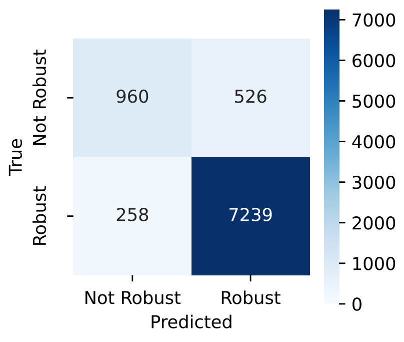


FIGURE 5.10: ROC curves for each of the five test folds. The black dashed line shows the performance of a baseline ("Null") classifier that always predicts one class.

20 nodes. Finally, Figure 5.11b shows the confusion matrix of the predictions computed by the model, where the entries are the averages computed across the five test folds. In the plot, we can visualize the good performances of the model as regards the number of correctly predicted subgraphs (on the diagonal) with respect to the cases where the model makes wrong predictions. Looking at the anti-diagonal the confusion matrix, we can also see that the model has a higher rate of false positives than false negative. Again, this is expected behaviour due to class misproportion in the dataset.



(A)



(B)

FIGURE 5.11: (A) Rolling mean accuracy (with a window of size 20), averaged over the 5 test folds, showing an increasing trend in performance as the number of graph nodes grows. (B) Confusion Matrix of the predictions computed by the model. Each entry of the matrix is the average of the corresponding entries in the five different confusion matrices of the test folds.

5.5.3 Use Case on Synthetic Data

The lower prediction accuracy in the case of small graphs (1-10 nodes) can be put into perspective, in light of the fact the fact we trained the model with subgraphs in which kinetic, stoichiometric and initial concentration parameters have been omitted (as explained in Section 5.2.1). In general, the smaller the graph, the higher the influence on its dynamics these parameters exert. As an example, let us consider the synthetic example of biochemical pathway introduced in Figure 5.1 and the corresponding pathway graph of Figure 5.3. Let us now consider the following kinetic and initial concentration (marking) parameters:

$$k1 = 1.0 \quad k3 = 0.01 \quad k5 = 0.01 \quad k7 = 0.3 \quad k2 = 5.0 \quad k4 = 0.1 \quad k6 = 5.0$$
$$m_0(A) = 50 \quad m_0(B) = 50 \quad m_0(C) = 100 \quad m_0(D) = 100$$
$$m_0(E) = 0 \quad m_0(F) = 0 \quad m_0(G) = 100 \quad m_0(H) = 0.$$

We used these parameters to simulate the ODEs of Figure 5.1b, calculating the corresponding robustness by varying the initial concentration of each molecule in the interval $[-20\%, +20\%]$. The robustness values obtained with the simulations are displayed in Table 5.3a. Analogously, in Table 5.3b we list the average and standard deviations obtained by the 5 different models evaluated in Section 5.5.2 (one trained on the respective CV fold), when tasked to predict the robustness probabilities of some input/output pairs of interest. We remark that values in these two tables are

TABLE 5.3: (A) Robustness values computed by numerical simulation of the ODEs in Figure 5.1. Input molecules with initial concentration equal to 0 are omitted. Output molecules with identical robustness values are merged. (B) Probabilities of robustness obtained from the model for some relevant input/output combinations.

Input	**Output**				
	A	**B**	**C/D**	**E/F**	**G/H**
A	1.00	0.73	0.99	1.00	1.00
B	1.00	0.73	0.99	1.00	1.00
C	1.00	1.00	0.00	0.99	0.99
D	1.00	1.00	0.00	0.99	0.99
G	1.00	1.00	1.00	1.00	0.50

(A)

In/Out	**Probability**
B/A	0.3798 ± 0.12
A/F	0.7254 ± 0.18
A/H	0.8835 ± 0.05
C/F	0.0793 ± 0.11
G/H	0.2351 ± 0.01

(B)

not directly comparable: those in Table 5.3a are exact robustness values, while those in Table 5.3b are probabilities of the robustness values to be greater than 0.5. In this specific case, the prediction turns out to be accurate in the case of input/output pairs corresponding to big induced subgraphs. This happens in the cases of the input/output pairs A/F and A/H, whose induced subgraphs are among the largest ones. In contrast, the prediction is incorrect for C/F: in this case, the models predicts a small robustness probability, while the simulations give 0.99. We observe that the robustness value of this input/output combination is sensitive to the perturbation of parameters that have been discarded when constructing the dataset. In particular, if the initial (omitted) concentration of C was 80 instead of 100, the robustness value of the pair C/F would become 0.5 rather than 0.99. Another case when the model prediction is wrong is that of the pair The prediction turns out to be wrong also in the case of input B/A. In this case, the probability given by the network is under 0.50, which contrasts to a value of 1 obtained by the ODEs simulation. Even in this case, we notice that the parameters that have been omitted in the dataset might have a strong influence on the robustness, such as kinetic formulas and the multiplicity of the edge directed to node B. Finally, in the case of the pair G/H, the prediction gives a small probability of robustness and indeed the actual measured value is borderline (0.50). More in general, we observe that smaller subgraphs are less frequent than medium-sized subgraphs in the dataset. Thus, it is possible that the model has learned to be more accurate on the latter subgraphs (to maximize the accuracy), at the expense of making more errors when predicting the former.

Part III

Deep Generative Learning on Graphs with Applications to Computational Chemistry

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

`\hypersetup{urlcolor=red}`, or

`\hypersetup{citecolor=green}`, or

`\hypersetup{allcolor=blue}`.

If you want to completely hide the links, you can use:

`\hypersetup{allcolors=.}`, or even better:

`\hypersetup{hidelinks}`.

If you want to have obvious links in the PDF but not the printed text, use:

`\hypersetup{colorlinks=false}`.