University of Pisa

Doctoral Thesis

Deep Learning on Graphs with Applications to the Life Sciences

Author:
Marco Podda

Supervisors:
Prof. Davide Bacciu
Prof. Alessio Micheli

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

in the

Computational Intelligence and Machine Learning Group
Department of Computer Science

April 30, 2021

UNIVERSITY OF PISA

Abstract

Department of Computer Science

Doctor of Philosophy

Deep Learning on Graphs with Applications to the Life Sciences

by Marco Podda

The application of Deep Learning models to complex biological problems has recently revolutionized the field of the life sciences, leading to advancements that could potentially change the quality and length of human life for the better. A significant contribution to this success comes from the ability of deep neural networks to well approximate non-linear biological functions and their 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.

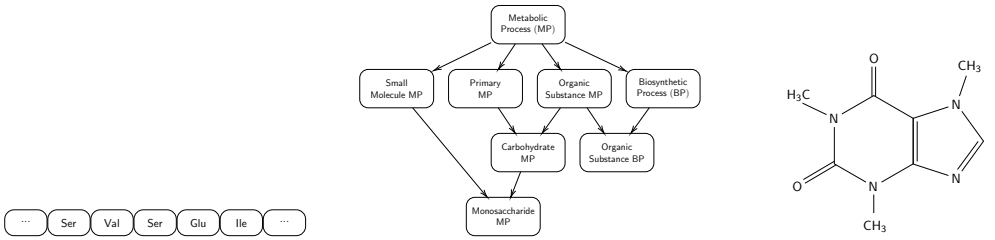
Contents

Chapter 1

Introduction

One of the long-standing goals of humanity is to understand the mechanisms that underlie biological life. The journey towards this end objective requires to solve several fascinating sub-problems: how do biological entities work at the cell level? Which processes determine a particular phenotype? How can we cure diseases? These broad questions require dedicated scientific studies, which necessarily entail analyzing large quantities of data to model complex phenomena. In this sense, the latest technological advancements in collecting, storing, and processing biological information have brought forth unprecedented progress. At the same time, an equal effort has been dedicated to developing computational methodologies that learn difficult biological tasks by extracting useful knowledge from these sheer amounts of data. Among the many, Deep Learning with neural networks [LBH15] has been undoubtedly one of the driving forces in these latest years. As we write this thesis, a Deep Learning model called *AlphaFold* has made a giant leap towards reliably predicting a protein's structure from its sequence of amino acids — the so-called *protein folding* problem [Sen+20]. With Deep Learning, we can now predict quantum properties of chemical structures within chemical accuracy, being 300k times faster than numerical simulations [Gil+17]. A Deep Learning method called *DeepVariant* can reconstruct a true genome sequence from High-Throughput Sequencing data with significantly greater accuracy than previous classical methods [Pop+18]. Lastly, Deep Learning has been successfully applied to counter antibiotic resistance by finding chemically different antibiotics from known ones but having the same bactericidal activity [Sto+20]. All these life sciences problems — and most life sciences problems more broadly — share apparent commonalities. One is that each of them concerns a poorly characterized phenomenon, meaning that the biological function that relates the data to the observed outcomes is only partially understood. In this case, Deep Learning is helpful because it shifts the burden of finding a suitable representation of the input data from the user to the modeling machine [BCV13]. In other words, instead of hand-crafting the knowledge required to solve a task, Deep Learning models infer task-specific knowledge from the data themselves. Another commonality regards the nature of the biological datum. Indeed, most biological data are *structured*, meaning that they can be decomposed into a set of entities and relations between them. Structured data in life sciences are ubiquitous: some examples are shown in Figure 1.1. Proteins can be thought of

as *sequences* of amino acids. Gene ontologies, which describe gene attributes across various species, are organized in *hierarchies*. Molecules are naturally represented as *graphs* where atoms are vertices, and chemical bonds are edges. Clearly, a biological datum taken as a whole has more informative content than its constituents taken in isolation. On the one hand, exploiting this superior expressiveness can be the key to design successful models of complex biological processes. On the other hand, working with structured data introduces additional challenges, such as representing them in a succinct way and preserving their relational nature at the same time — or in other words, how to keep their informative content intact while also being computationally efficient. Traditional Machine Learning models are designed to work with



(A) Protein sequence. (B) Gene ontology. (C) Molecular graph.

FIGURE 1.1: Examples of biological data.

non-relational real-valued vectors. Thus, they cannot be applied off-the-self to problems where data have a complex structure. In principle, it is possible to circumvent this complexity by hard-coding the relational information into vectors of structural descriptors. While useful to some extent, this approach builds upon a somewhat under-specified representation of knowledge. Moreover, it heavily relies on domain expertise to choose which structural information to retain and which to discard. In stark contrast, Deep Learning methodologies for structured domains allow to extract this relational knowledge directly and in a goal-oriented way. Neural approaches for the recursive processing of structured data have been studied long before the advent of Deep Learning [Elm90; HS97; SS97; FGS98] and were extended to general classes of graph data while modern Deep Learning methods were still under development [Mic09; Sca+09]. These research efforts culminated in three types of neural networks for structured data: Recurrent Neural Networks for sequence data, Recursive Neural Networks for tree/hierarchical data, and Deep Graph Networks for general graphs (*e.g.* cyclic/acyclic, directed/undirected). Recently, with the availability of large data sources and the increase of computational power, these models have been rediscovered and improved by the Deep Learning community. At the same time, their wide spectrum of potential applications attracted the attention of researchers coming from very different domains. When applied to challenging biological tasks, these models have demonstrated strong performances and great success [Bia+00b; LPB13; Duv+15; Bra+19].

1.1 Objective and Contributions

The high-level objective of this thesis is to give the reader broad perspective on the kinds of biological tasks that can be tackled with Deep Learning for structured data. We do so by focusing on two classes of learning problems: *predictive* and *generative*. Predictive problems deal with modeling an unknown input-output relationship of interest; generative problems deal with learning the data distribution to sample novel instances. This thesis present one technical contribution and one applied contribution to a specific life sciences problem for each of these two learning paradigms.

The first technical contribution is an empirical evaluation of different Deep Graph Networks proposed in the literature for graph classification tasks. The applied contribution pertains to the life sciences sub-domain of computational biology. Specifically, we develop a novel learning framework for the prediction of dynamical properties of biochemical pathways with Deep Graph Networks. In the context of generative learning, the technical contribution is a general-purpose, domain-agnostic generative model of unlabeled graphs. The applied contribution concerns an applies deep generative modeling to the life sciences sub-domain of computational chemistry. Specifically, we present a novel generative model of molecules for *de novo* drug design. Following, we discuss our contributions in detail.

An Empirical Evaluation of Deep Graph Networks for Graph Classification

We provide a systematic and fair evaluation of Deep Graph Networks for graph classification tasks. In the literature, these models' evaluation is often biased by poor experimental setups and unfair comparisons. We develop a unified framework under which these models can be evaluated coherently and their results adequately compared. This work lets us also shed light on what Deep Graph Networks learn differently from structure-agnostic baselines and the relation between structural features of the graphs (precisely, the vertex degree) and model performances. Finally, we draw useful insights on how Deep Graph Networks behave on certain tasks in relation to their inner components.

A Novel Learning Framework for the Prediction of Dynamical Properties of Biochemical Pathways

We address a newly formulated predictive problem in the context of biological pathways. A biochemical pathway is a dynamical system in which molecules interact with each other through chemical reactions. Each molecule can take different roles in a biochemical pathway: it can be a reactant, a product of some chemical reaction, and a promoter (or inhibitor) of other reactions. The pathway state at a given moment in time corresponds to some function being performed or not (for example, cell replication). Biochemical pathways can be conveniently represented as Petri networks, allowing researchers to study dynamical properties of the system such as reachability of steady states, causality between species, and robustness. Specifically, we focus

on the property of robustness, which can be described informally as the pathway's resilience to maintain its function against external perturbations. The standard way to assess robustness relies on an extensive number of numerical simulations, which in turn are expensive in terms of computational time. Our contribution is to show that, given pathways represented as Petri networks, we can reliably predict an indicator of their robustness using a Deep Graph Network instead. The assumption at the basis of this work is that the structure of the pathway, and not other factors such as kinetic and chemical constants, provides (in most cases) enough signal to let the model infer its robustness. We show experimentally that this is indeed possible to a reasonable extent, opening up to a future where costly numerical simulations can be bypassed altogether. Furthermore, we study how different architectural choices of the Deep Graph Network influence performances.

A Generative Model for Unlabeled Graph Generation

Graph generation is a challenging problem with applications in various research fields. We propose a general-purpose recurrent model to generate arbitrary unlabeled graphs, whose structure resembles that of training samples. Despite its conceptual simplicity, we experimentally demonstrate its effectiveness on a wide range of graph datasets, and that it is able to generalize both local and global properties of the training graphs. The results show that the proposed approach outperforms canonical baselines from graph theory and performs comparably to the current state of the art on the unlabeled graph generation task.

A Generative Model of Molecules for *de novo* Drug Design

We study the molecular generation problem in the context of *de novo* drug design. Currently, deep generative approaches on this task belong to two broad categories, differing in how molecules are represented. One approach is based on encoding molecular graphs as strings of text and learning a generative language model of such strings. Another, arguably more expressive, approach works directly with the molecular graph. Our work addresses two main limitations of the former generative framework: the generation of chemically invalid or duplicate molecules. We develop a language model for small molecular substructures called fragments, loosely inspired by the well-known paradigm of Fragment-Based Drug Design [EMO04] — in other words, we generate molecules *fragment-by-fragment*, rather than *atom-by-atom*. We show experimentally that this novel approach greatly improves the validity rate of the generated molecules. To avoid generating duplicate molecules, we present a frequency-based masking strategy that encourages the model to build molecules using infrequent fragments. Our experiments demonstrate that our simple strategy largely outperforms other language model-based competitors, reaching performances comparable to the state of the art in the task typical of graph-based approaches. Moreover, we show how the generated

samples are endowed similar molecular properties to those of training samples, even without explicitly being trained to do so.

1.2 Structure of the Thesis

This thesis is divided in four parts. In Part I, we present all the introductory notions to facilitate the understanding of this thesis' content. In the first part of Chapter 2, we provide a brief overview of the concepts of Machine Learning that are functional to our discussion. The rest of the chapter is dedicated to discuss the foundations of (Deep) Neural Networks for supervised and unsupervised learning. In Chapter 3, we present the class of Deep Learning models for the processing of structured data, namely Recurrent Neural Networks for sequence data, Recursive Neural Networks for tree/hierarchical data, and Deep Graph Networks for graph data. We conclude presenting the framework of Deep Generative Models for graph generation. In Part II, we present the two contributions pertaining predictive modeling using structured data and Deep Learning, and its application to computational biology. Specifically, in Chapter 4, we present a fair and rigorous evaluation of existing Deep Graph Networks; while in Chapter 5, we propose a novel application of Deep Graph Networks to the problem of predicting the dynamical properties of biological pathways. Part III of the thesis presents our two original contributions in the context deep generative learning of graphs, and its applications to computational chemistry. In Chapter 6, we discuss a general-purpose model capable of constructing unattributed graphs by sequentially generating their edges. In Chapter 7, we introduce a novel generative model of molecules in the context of drug discovery. Lastly, Part IV concerns concluding remarks and future works, which we discuss in Chapter 8.

1.3 Published Works

Several publications have contributed to the development of this thesis. Section 3.5 is based on the journal article:

D. Bacciu, F. Errica, A. Micheli, and M. Podda. "A gentle introduction to deep learning for graphs". In: *Neural Networks* 129 (2020), pp. 203–221,

where we re-elaborated the concepts behind Deep Graph Networks in a systematic introductory review.
Chapter 4 draws from our conference paper:

F. Errica, M. Podda, D. Bacciu, and A. Micheli. "A fair comparison of graph neural networks for graph classification". In: *Proceedings of the 8th International Conference on Learning Representations (ICLR)*. 2020,

which presented a fair and rigorous comparison of several state-of-the-art Deep Graph Networks.

Chapter 5 is a compendium of the work developed in two conference papers:

P. Bove., A. Micheli., P. Milazzo., and M. Podda. "Prediction of Dynamical Properties of Biochemical Pathways with Graph Neural Networks". In: *Proceedings of the 13th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS*. INSTICC. SciTePress, 2020, pp. 32–43

M. Podda, A. Micheli, D. Bacciu, and P. Milazzo. "Biochemical Pathway Robustness Prediction with Graph Neural Networks". In: *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. 2020.

The former work first formulated the predictive problem, receiving the Best Paper Award in the related conference. Later, it has been selected to be extended in a post-proceedings journal publication (in review).
Chapter 6 derives from our work on the conference paper:

D. Bacciu, A. Micheli, and M. Podda. "Graph generation by sequential edge prediction". In: *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. 2019,

which was later extended to a journal article in:

D. Bacciu, A. Micheli, and M. Podda. "Edge-based sequential graph generation with recurrent neural networks". In: *Neurocomputing* (2020).

Lastly, Chapter 7 concerns work published in the following conference paper:

M. Podda, D. Bacciu, and A. Micheli. "A Deep Generative Model for Fragment-Based Molecule Generation". In: *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (AISTATS)*. vol. 108. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2240–2250.

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 in Section 2.1. In Section 2.2, we lay out the foundations of how learning from data is possible. In Section 2.3, we discuss the more practical aspects of selecting and evaluating machine learning models. Section 2.4 briefly reviews the core concepts about neural networks, such as the procedure by which they are trained, the loss functions they optimize, and how they are regularized. Section 2.5 is devoted to autoencoders, a special class of neural networks for unsupervised learning. In Section 2.6, we introduce the main framework used in this thesis, that of deep learning. Finally, in Section 2.7, we present the class of deep learning models for generative tasks.

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 quantitatively or qualitatively measure 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 [Mit97], **learning** is defined as improving at some task with experience. We call *learner* a computer program that is capable of learning from data. To start off, let us first briefly describe the key components of a ML system in detail, introducing other useful notation, definitions, and concepts along the way:

- the **experience** 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 $\mathfrak{F}$ (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 *classification*, where $\mathfrak{F}$

is a function that assigns a categorical *label* to a data point; *regression*, where $\mathfrak{F}$ relates an example to a desired numerical quantity; and *density estimation*, where $\mathfrak{F}$ is the probability density (or mass, for the discrete case) function from which the examples are drawn;

- the **improvement** is measured by a so-called performance metric — a function that returns a quantitative measurement of how "well" the task is being learned. For example, in a classification task, one can measure improvement by computing the *accuracy* of the learner, *i.e.* the proportion of examples to which the learner assigns the correct label out of the total number of available examples.

ML can be broadly divided into 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 learn the relationship between the inputs and the outputs. Classification and regression are instances of 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** [SB18] is about instructing an *agent* to act in an environment by rewarding or penalizing its actions. This thesis exclusively deals with supervised and unsupervised learning.

Regardless of the learning paradigm, the central issue in ML is **generalization**; that is, the function inferred by the learner should be general enough to give the correct output even on data points not used during the learning phase. The concept of generalization is strictly related to that of *induction*, *i.e.* deriving a general rule by observing a subset of its specific instances. 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 desired target is found. A program that implements this search is called a **learning algorithm**.

The training process is driven by the data. For a given task, the learning algorithm has access to a **dataset** $\mathbb{D} = \{z^{(i)}\}_{i=1}^{n}$, 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 output space $\mathcal{Y}$ is tightly coupled to the task to be learned; for example, in classification tasks, $\mathcal{Y}$ is a discrete set, and its elements are called *labels*; in regression tasks, $\mathcal{Y} \subseteq \mathbb{R}$, and its elements are called *targets*. In unsupervised learning, $\mathcal{Z}$ is just the input space $\mathcal{X}$. For this reason, data used in unsupervised tasks is often

called *unlabeled*. For the moment, we assume that the input space $\mathcal{X}$ is a subspace of all d-dimensional real-valued vectors $\mathbb{R}^d$. We call vectors $\boldsymbol{x} \in \mathbb{R}^d$ *feature vectors* and their components **features**. Given a vector $\boldsymbol{x}$, we use the notation $\boldsymbol{x}_i$ 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 structured data.

The space of candidate functions explored by the learning algorithm is called **hypotheses space**. Formally, a hypotheses space is a set of functions $\mathcal{H}_\Theta = \{h_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \Theta\}$ whose members are called *hypotheses*. The hypotheses space is used indirectly by the learning algorithm, which instead operates in **parameter space** Θ. The elements of parameter space are called *parameters*; each parameter $\boldsymbol{\theta} \in \Theta$ uniquely identifies a hypothesis $h_{\boldsymbol{\theta}} \in \mathcal{H}_\Theta$. Usually, the parameters $\boldsymbol{\theta}$ are themselves real-valued vectors.

Besides the dataset, the learning algorithm is also given a **loss function** $\mathcal{L} : \Theta \times \mathcal{Z} \to \mathbb{R}_+$. Its role is to provide a point-wise, non-negative measurement of the error committed when approximating the unknown function $\mathfrak{F}$ with a hypothesis $h_{\boldsymbol{\theta}}$. The objective of the learning algorithm is thus to find the optimal hypothesis $h_{\boldsymbol{\theta}^*}$, whose error approximating $\mathfrak{F}$ is the lowest. Such an objective is formalized by the following optimization problem:

$$h_{\boldsymbol{\theta}^*} = \arg\min_{\boldsymbol{\theta} \in \Theta} \mathfrak{R}(h_{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \int \mathcal{L}(\boldsymbol{\theta}, z)\, dz,$$

where $\mathfrak{R}$ 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 $\mathbb{D}$, randomly sampled from $p(z)$. Therefore, a tractable optimization problem is used instead:

$$h_{\hat{\boldsymbol{\theta}}} = \arg\min_{\boldsymbol{\theta} \in \Theta} \mathfrak{R}_{\mathbb{D}}(h_{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{\theta}, z^{(i)}),$$

where $\mathfrak{R}_{\mathbb{D}}$ 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 $h_{\hat{\boldsymbol{\theta}}}$ given as output by the learning algorithm a **model**. The learning principle associated to this optimization problem is known as 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) [Rud16]. Briefly, SGD reduces the value of the loss function by "moving" the parameters in its negative gradient direction. The process requires initializing the parameters to some randomly chosen value $\boldsymbol{\theta}(0)$ and iterating over the

dataset several times. At each iteration t, the parameters are updated according to the following rule:

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

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

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

is an estimate of the true gradient of the loss function, averaged over a *mini-batch* of $b \ll n$ data points. SGD has several desirable properties as regards its convergence [RM51]: 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 surely. Over the years, different variants of SGD have been developed; a program that implements a specific variant is called an **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**, or *complexity*. 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 exactly approximate the noise of the dataset, which however is not a property of the general population the dataset was sampled from. When overfitting occurs, one of such hypotheses is wrongly chosen as model according to the ERM principle. To counter overfitting, one usually divides 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 used to get an estimate of the true generalization error. Overfitting can be detected by monitoring at which point 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 hypotheses space used by the learning algorithm. A principled form of regularization of the learning process derives from the

field of Statistical Learning Theory (SLT) [Vap00], 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 $\mathfrak{C}(\mathcal{H}_\Theta)$ of a hypotheses space is known[1], the following inequality:

$$\mathfrak{R}(h_{\boldsymbol{\theta}}) \leq \mathfrak{R}_{\mathbb{D}}(h_{\boldsymbol{\theta}}) + \varepsilon(n, \mathfrak{C}(\mathcal{H}_\Theta), \delta)$$

holds with probability of at least $1 - \delta$ provided that $n > \mathfrak{C}(\mathcal{H}_\Theta)$. 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 $\mathcal{H}_\Theta$. These two terms are tightly related: the training error can be decreased by increasing the capacity of the model, at the risk of incurring in overfitting. 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) [Vap00], 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 the out-of-sample performance of the model. In model evaluation, one is not necessarily interested to evaluating the generalization error; in most practical cases, other performance metrics are used. For example, in classification tasks, the accuracy of the model is often evaluated instead. The estimation of the performance metric is done on the test set; there exist different estimators, depending on the way the dataset is split into training and test partitions. The simplest is the **hold-out** estimator, which requires to split 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* [HTF09]. Informally, bias is related to how close (or how far) the estimation is 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 calculated on a single test set and might over-estimate (or under-estimate) the true out-of-sample performance just by chance. To reduce the variance of the estimation, k-fold Cross-Validation (CV) [AC10] can be used instead. k-fold CV requires to split the data in k disjoint partitions and repeat the evaluation k times. Each repetition is

[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.

a hold-out estimation where one partition in turn acts as test set, and the remaining $k-1$ are taken together as training set. The final estimator is the average of the k estimates. In practical cases, $k = 5$ or $k = 10$ are often used [HTF09; Koh+95].

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 regulated by other settings which differ from the usual model parameters. 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 a set of hyper-parameter configurations to be evaluated. A configuration is any assignment of the hyper-parameter values. Given a configuration, a model is instantiated with the corresponding hyper-parameters, trained on the training set and evaluated on some held-out data. Usually, the hyper-parameters set is specified as a *grid* where each hyper-parameter is associated to a 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 configurations. Other strategies for defining the set of hyper-parameters and the model selection algorithm are also possible [BB12b; BB12a]. 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 further holding out a **validation set** from the training set, to select a configuration of hyper-parameters that generalizes. Once such configuration is found, the out-of-sample performance is estimated on the test set as usual. From this perspective, model selection can be viewed as a nested assessment inside the outer model evaluation task, and can be tackled using the same kinds of estimators such as hold-out or k-fold CV.

2.4 Neural Networks

This work revolves around feed-forward Neural Networks (NNs) [Hay09; GBC16], a learning algorithm inspired by the mechanisms through which the neurons in our brain learn [Ros58]. 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 behavior 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 $\boldsymbol{x} \in \mathbb{R}^d$, the general function performed by a neuron can be expressed in the following form:

$$y = \sigma\left(\boldsymbol{w}^\mathsf{T}\boldsymbol{x} + b\right) = \sigma\left(\sum_{i=1}^{d} \boldsymbol{w}_i \boldsymbol{x}_i + b\right),$$

where $\boldsymbol{w} \in \mathbb{R}^d$ are called **weights** (or parameters), $b \in \mathbb{R}$ is called **bias** term, σ is called **activation function**, and $y \in \mathbb{R}$ is an output signal. An example of neuron is shown in Figure 2.1a. We can generalize a neuron to output a vector $\boldsymbol{y} \in \mathbb{R}^y$ as follows:

$$\boldsymbol{y} = \sigma\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right),$$

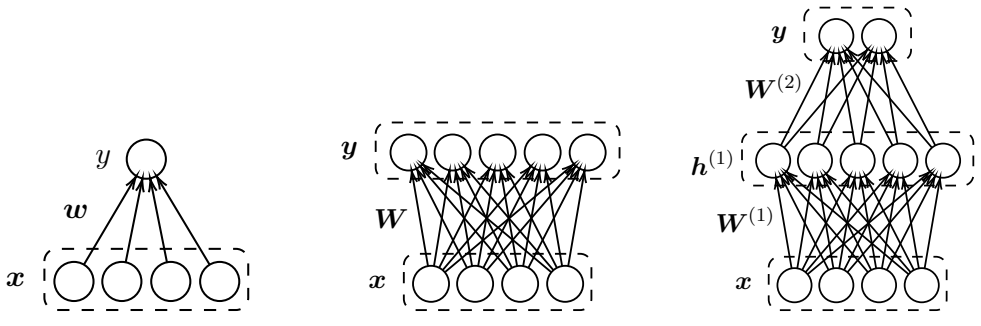
where $\boldsymbol{W} \in \mathbb{R}^{y \times d}$ is now a weight matrix, $\boldsymbol{b} \in \mathbb{R}^y$ is a bias vector, and the activation function is applied element-wise. Such construction is called a **layer**, and it is shown in Figure 2.1b. NNs compose multiple layers sequentially to implement complex functions as follows:

$$f(\boldsymbol{x}) = \left(f^{(1)} \circ f^{(2)} \circ \ldots \circ f^{(L)}\right)(\boldsymbol{x}),$$

where $f^{(l)}$ for $l = 1, \ldots, L-1$ are called **hidden layers** and $f^{(L)}$ is called **output layer**. In general, the l-th hidden layer of the network computes an **activation** $\boldsymbol{h}^{(l)} \in \mathbb{R}^{h_l}$ as follows:

$$\boldsymbol{h}^{(l)} = f^{(l)}(\boldsymbol{h}^{(l-1)}) = \sigma(\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}),$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{h_l \times h_{l-1}}$ is a weight matrix, $\boldsymbol{h}^{(l-1)} \in \mathbb{R}^{h_{l-1}}$ is the output of the previous layer (with $\boldsymbol{h}^{(0)} = \boldsymbol{x}$), and $\boldsymbol{b} \in \mathbb{R}^{h_l}$ is a bias vector. In other words, a neural network applies a series of parameterized transformations to an input vector to compute some desired output. The network parameters $\boldsymbol{\theta} = (\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{W}^{(L)}, \boldsymbol{b}^{(L)})$ are optimized with SGD so that the "end-to-end" function $f(\boldsymbol{x})$ computed by the network is a good approximation of the target function $\mathfrak{F}$. Notice that, to be able to use SGD, all the operations performed at the layer and neuron levels must be differentiable.



(A) A neuron. (B) A layer. (C) A MLP.

FIGURE 2.1: (A) Examples of neural networks (biases and activation functions not shown).

The number of layers and the dimension of their parameter matrices constitute

the *architecture* of the network. The network architecture, and the nature of the activation function of the hidden layers, play a major role in determining the kinds of functions a NN can learn. Specifically, NNs with at least one hidden layer with non-linear activation function, also called Multi-Layer Perceptrons (MLPs), are *universal approximators*, meaning that they can approximate arbitrarily well any computable function [Cyb89]. However, no guarantees as to which architecture is needed to attain the best approximation can be given, according to the "no free lunch theorem" [WM97]; thus, the network architecture is task-dependent and must be optimized with model selection. An example of MLP is shown in Figure 2.1c. As regards the activation functions, modern neural networks use one among three different kinds of non-linearities (or variations thereof), shown in Figure 2.2:

- the *sigmoid function*, defined as:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}};$$

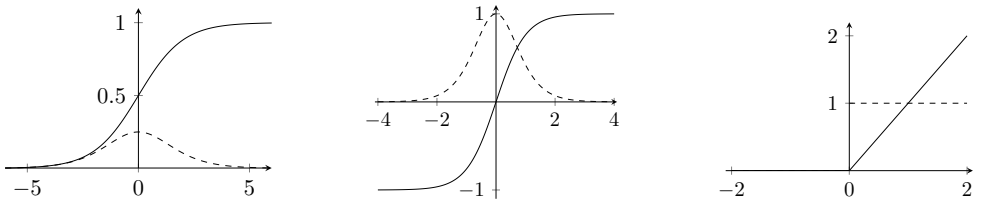
- the *hyperbolic tangent function*, defined as:

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

- the Rectified Linear Unit (ReLU) function [GBB11], defined as:

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

  Notice that, even though the ReLU function is not differentiable at $x = 0$, it can be used in practical settings with minimal adaptations.



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

FIGURE 2.2: Examples of activation functions for the hidden layers (solid line) and their derivatives (dashed line).

One distinctive advantage of NNs with respect to other learning algorithms is that they allow *automatic feature engineering*: in fact, the hidden layers of a NN extract features from the data without an explicit guidance from the end user. It is the network itself that decides which inner **representation** of the data is more appropriate to solve a given task by changing the values of the parameters during learning. In other words, the decision of which representation to use 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 one of the keys behind the success of NNs.

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, called *forward propagation*, training examples are fed to the network, which produces an output for each data point. At the end of forward propagation, the error between the output and the ground truth values is calculated through 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 **back-propagation** algorithm [RMH86], 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 modified according to the update rule implemented by the optimizer. Training NNs requires particular care to ensure the convergence of SGD. In particular, the initial values of the weight matrices should be initialized with small random values around 0 for *symmetry 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 [LeC+98].

2.4.2 Output Layers and Loss Functions

The output layer of a NN is chosen according to the task it should 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) [Bis06]. Specifically, one assumes that the unknown function $\mathfrak{F}$ is a conditional distribution $p(y \mid \boldsymbol{x})$[2], where $y \in \mathbb{R}$ for simplicity. Given a model $p_{\boldsymbol{\theta}}$ for p (in this case, the NN), the goal is to maximize the likelihood of the observed training data $\mathbb{D} = \left\{(\boldsymbol{x}^{(i)}, y^{(i)})\right\}_{i=1}^{n}$. In other words, we want to make the distribution $p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x})$ inferred by the model as similar as possible to the empirical distribution $\hat{p}(y \mid \boldsymbol{x})$ observed from the data. This corresponds to learn a set of parameters $\boldsymbol{\theta}$ such that the Kullback-Leibler Divergence (KLD) between the two distributions, defined as:

$$\arg\min_{\boldsymbol{\theta}} \mathrm{KLD}(\hat{p} \parallel p_{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}} \left[\log \hat{p}(y \mid \boldsymbol{x}) - \log p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x})\right],$$

is minimized. The KLD is a measure of dissimilarity between probability distributions; it is a proper loss function, since its output is always ≥ 0, and it evaluates to 0 if and only if $\hat{p} = p$. Since $\hat{p}$ does not depend on the parameters $\boldsymbol{\theta}$, to minimize the KLD it is sufficient to minimize the term $-\mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}} [\log p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x})]$, corresponding to the negative log-likelihood of the distribution inferred by the model using the training

[2] Here, we focus on the supervised case; later on, we shall see how MLE is applied in the unsupervised case.

data. Thus, minimizing the KLD with ERM is equivalent to minimizing the negative log-likelihood of the data:

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

Below, we review three of the most common output layers, and the loss functions by which the corresponding networks can be trained. Following, we assume a dataset $\mathbb{D}$ defined as above, and a neural network f with $L-1$ hidden layers and parameters $\boldsymbol{\theta}$ (which will be omitted from formulas for simplicity).

Linear Output Layer A linear output layer is used to solve regression tasks, *i.e.* tasks where the target function $\mathfrak{F}$ is continuous. It is defined as:

$$f^{(L)}(\boldsymbol{h}^{(L-1)}) = \boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}.$$

Notice that a linear output layer does not have an activation function; equivalently, one can think that the activation function is the identity. The model assumed for the true conditional is the following:

$$p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x}) \sim \mathbb{N}(y; f(\boldsymbol{x}), 1),$$

where $\mathbb{N}$(,i)s a Gaussian distribution. In other words, the targets are normally distributed with estimated mean $f(\boldsymbol{x})$ and unit variance. With this setup, the MLE solution is equivalent to minimizing the Mean Squared Error (MSE) loss function for each training example in the dataset:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, (\boldsymbol{x}, y)) = \text{MSE}(f(\boldsymbol{x}), y) \stackrel{\text{def}}{=} \|f(\boldsymbol{x}) - y\|_2.$$

Notice that minimizing the MSE corresponds to minimizing the Euclidean distance between the target y and the output of the network $f(\boldsymbol{x})$. A network with a single linear output layer and no hidden layers trained to minimize the MSE loss function is commonly known in the 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 target function $\mathfrak{F}$ takes values in the discrete set $\{0, 1\}$. The label 0 is usually called the *negative* class, while the label 1 is called the *positive* class. The layer is defined as follows:

$$f^{(L)}(\boldsymbol{h}^{(L-1)}) = \text{sigmoid}\left(\boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}\right).$$

Since the co-domain 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:

$$p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x}) \sim \text{Bernoulli}(y; f(\boldsymbol{x})),$$

In other words, the targets are assumed to be Bernoulli distributed with estimated probability $f(\boldsymbol{x})$. The MLE solution is equivalent to minimizing the Binary Cross-Entropy (BCE) loss function over the training examples in the dataset:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, (\boldsymbol{x}, y)) = \text{BCE}(f(\boldsymbol{x}), y) \stackrel{\text{def}}{=} y \log(f(\boldsymbol{x})) + (1 - y) \log(1 - f(\boldsymbol{x})).$$

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 target function $\mathfrak{F}$ takes values in a discrete set $\boldsymbol{\mathcal{C}} = \{c_1, \ldots, c_K\}$ of K mutually exclusive classes. In these cases, the targets y are expressed as *one-hot* vectors $\boldsymbol{y} \in \mathbb{R}^K$. In practice, each position of the one-hot vector corresponds to a specific class, and its entries are defined as follows:

$$\boldsymbol{y}_i = \begin{cases} 1 & \text{if } y = c_i \\ 0 & \text{otherwise.} \end{cases}$$

Specifically, a one-hot vector encodes the target as a discrete probability distribution over the possible classes, where all the mass is put on the class corresponding to the label y. The layer is defined as:

$$f^{(L)}(\boldsymbol{h}^{(L-1)}) = \text{softmax}\left(\boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)}\right),$$

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

$$\text{softmax}(\boldsymbol{s}_i) = \frac{e^{\boldsymbol{s}_i}}{\sum_{k=1}^{K} e^{\boldsymbol{s}_k}}.$$

More precisely, 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:

$$p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \sim \text{Multinoulli}(y; f(\boldsymbol{x}), k).$$

In other words, the targets are Multinoulli (categorically) distributed with estimated class probabilities $f(\boldsymbol{x})$. Minimizing the KLD in this setting is equivalent to minimizing the Cross-Entropy (CE) loss function, defined as:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, (\boldsymbol{x}, \boldsymbol{y})) = \text{CE}(f(\boldsymbol{x}), \boldsymbol{y}) \stackrel{\text{def}}{=} -\sum_{k=1}^{K} \boldsymbol{y}_k \log f(\boldsymbol{x})_k.$$

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 the 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 act on the network architecture, reducing the number of hidden layers and the number of units in each layer. This effectively constrains the number of functions learnable by the architecture. 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, and assuming a dataset $\mathbb{D} = \left\{(\boldsymbol{x}^{(i)}, y^{(i)})\right\}_{i=1}^{n}$ as usual, this can be achieved by augmenting the training objective with a *penalty term* as follows:

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

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

- L1 regularization, where the penalty is the absolute value of the weights ($\|.\|_1$). The bias of L1 regularization is to push the values of some weights to zero. This corresponds to training a NN with fewer parameters (parameters set to 0 do not contribute to the loss function);
- L2 regularization, where the penalty is equal to the square value of the weights ($\|.\|_2$). The bias induced by the L2 regularization is to penalize large values of the weights. Intuitively, restricting the range of the values that the weights can take limits the number of functions the network can approximate.

Notice that these are not the only penalties possible, although the ones described above are the most used in practical settings [HTF09].

Early Stopping Early stopping [Pre98] 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. The parameters that yielded the best validation performance 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 [Sri+14] 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 units in a layer by multiplying their output with a binary vector mask, whose entries are samples from a Bernoulli distribution with a hyper-parameter 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 set to zero after being multiplied with 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* [GG16].

2.5 Autoencoders

An Autoencoder (AE) [Bal12] 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 achieve a good reconstruction. Given a d-dimensional input vector $\boldsymbol{x} \in \mathbb{R}^d$, an autoencoder computes a function $f(\boldsymbol{x}) = (\text{enc}_{\boldsymbol{\psi}} \circ \text{dec}_{\boldsymbol{\phi}})(\boldsymbol{x})$ as follows:

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

where $\boldsymbol{h} \in \mathbb{R}^h$ is an h-dimensional *latent code*, $\boldsymbol{r} \in \mathbb{R}^d$ is a *reconstruction* of the input, $\text{enc}_{\boldsymbol{\psi}}$ is an encoding NN or **encoder** with parameters $\boldsymbol{\psi}$, $\text{dec}_{\boldsymbol{\phi}}$ is a decoding NN or **decoder**, with parameters $\boldsymbol{\phi}$. Despite being an unsupervised model, AEs can be trained in a supervised way, by implicitly using a dataset of the form $\mathbb{D} = \left\{(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(i)})\right\}_{i=1}^{n}$, *i.e.* one where the input itself is the target. The loss function of an AE is called *reconstruction loss*, and measures the error between the input $\boldsymbol{x}$ and the reconstructed output $f(\boldsymbol{x})$, as shown in Figure 2.3. For continuous inputs, the reconstruction is simply the MSE loss:

$$\mathcal{L}(\boldsymbol{\theta}, (\boldsymbol{x}, \boldsymbol{r})) = \|\boldsymbol{x} - \boldsymbol{r}\|_2 ,$$

where $\boldsymbol{\theta} = (\boldsymbol{\psi}, \boldsymbol{\phi})$. For discrete inputs, the CE loss function can be used instead.

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 $\boldsymbol{x} = \boldsymbol{r}$ everywhere by choosing

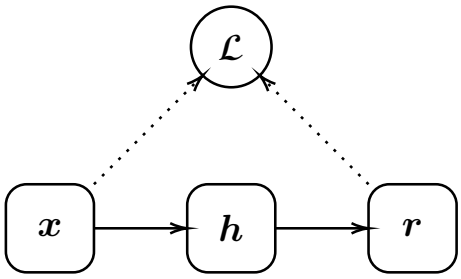


FIGURE 2.3: An Auto-Encoder.

f as the identity function. This is not very useful in practice, as the latent space of a network structured in this way would not learn anything useful about the data; 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, its latent space is a *bottleneck* that is forced to retain relevant features that allow to reconstruct 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 the set of integers. Clearly, this mapping would be uninformative of the true data distribution.

2.5.1 Regularized Autoencoders

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 loss function:

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

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 hidden 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 $\rho \in \mathbb{R}$ is a desired average activation and $\boldsymbol{h} \in \mathbb{R}^h$ are the actual hidden activations, the penalty is formulated as:

$$\Psi(\boldsymbol{h}) = \sum_{i=1}^{h} \rho \log \frac{\rho}{\boldsymbol{h}_i} (1 - \rho) + \log \frac{1 - \rho}{1 - \boldsymbol{h}_i}.$$

If ρ is chosen to be small, the penalty constrains most hidden units to have zero activation. Similarly, *contractive* AEs [Rif+11] 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:

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

which intuitively corresponds to penalizing hidden activations that have large variation for small variations of the input. Thus, the local structure of the latent space is forced to be smooth. Differently from the other variants, *denoising* AEs [Vin+10] achieve regularization acting on the training procedure, rather than via a penalty term. In a denoising AE, the input $\boldsymbol{x}$ is corrupted before being passed to the network (usually through Gaussian noise addition). Thus, the network uses the corrupted version $\tilde{\boldsymbol{x}}$ during forward propagation, while the loss is calculated on the original input, effectively training the model to remove the noise from $\boldsymbol{x}$. By doing so, the network is forced to learn meaningful patterns. The forward propagation phase of a denoising AE is shown in Figure 2.4.

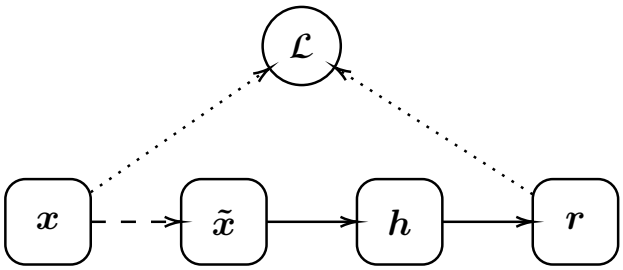


FIGURE 2.4: A denoising Auto-Encoder. The dashed arrow indicates the corruption process which transforms the input $\boldsymbol{x}$ into a noisy version $\tilde{\boldsymbol{x}}$, which is not directly part of the forward propagation.

2.6 Deep Learning

In Section 2.4, we have talked about the representational power of NNs. 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 to obtain 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 [Ben09]. This data-driven *representational bias* has proven effective in many practical domains, such as computer vision [KSH17] and Natural Language Processing (NLP) [Vas+17], establishing the success of deep NNs in many ML tasks. The term Deep Learning (DL) [GBC16], coined around 2006, is used to refer to NN architectures composed of many 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 surface of the loss functions optimized by deep networks is 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 optimization problem NNs try to solve [GVS15; JC17]. In parallel, improvements have been achieved by devising more effective optimizers [KB15; Rud16] and novel activation functions [GBB11]. 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 [Zha+20] to prevent gradient explosion, batch normalization [IS15] and residual connections [He+16] 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 [Aba+16; Pas+17] to speed up the training and inference processes of deep networks.

2.6.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) [LB+95] 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 [KSH17]. 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* 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 a feature map where each entry is defined as follows:

$$\boldsymbol{F}_{ij} = (\boldsymbol{M} * \boldsymbol{K})_{ij} = \sum_{i=1}^{w} \sum_{j=1}^{h} \boldsymbol{M}_{(i+s)(j+t)} \boldsymbol{K}_{st}.$$

In the equation above, $\boldsymbol{M} \in \mathbb{R}^{w \times h}$ is a 2-D matrix (for example, an image with width w and height h), $\boldsymbol{K} \in \mathbb{R}^{s \times t}$ is a learnable filter, or *kernel*, with width $s \ll w$ and

height $t \ll h$, and $*$ is a convolutional operator. In practice, the filter is "slid" on the input row-wise. The values in the feature map have larger magnitude 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 map is computed using the same weights (those of the kernel). This approach is called *weight sharing*, and it implements *translational invariance*, which intuitively means that the features are detected regardless of their position in the input. Another essential layer of CNNs is the *pooling layer*. A pooling layer works by dividing the input into 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 feature. Convolutional and pooling layers are applied sequentially to the input data, followed by a standard hidden layer activation function, usually a ReLU. The three layers (Convolution-Pooling-ReLU) in cascade are the standard building block of convolutional architectures. The last layers of a CNN usually consist of a MLP (or a single output layer), which uses the final representation given by the convolutional blocks as input and computes 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 for *speech recognition* tasks) and 3-D inputs (such as video frames for *object tracking* tasks).

2.7 Deep Generative Models

Typical unsupervised tasks require to learn the probability distribution of the data, or some aspect of it, such as, for example, how the data is clustered together. Broadly speaking, once the input probability distribution is known, one could use it to perform:

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

Deep Generative Models (DGMs) [GBC16] are essentially deep architectures that learn how to do inference and sampling, or just sampling, from the data distribution. Based on this definition, we can further distinguish between:

- *prescribed* (or *explicit*) DGMs, which can jointly learn inference as well as sampling;
- *implicit* DGMs, which only learn sampling.

Below, we provide a brief overview on the topic, assuming learning happens through an unlabeled dataset $\mathbb{D} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{n}$, where $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$.

2.7.1 Prescribed Deep Generative Models

Prescribed models are trained with MLE. Specifically, the goal is to approximate the true distribution of the data $p(\boldsymbol{x})$ using a model $p_{\boldsymbol{\theta}}$, trained to minimize the KLD between the empirical distribution $\hat{p}$ and the output distribution:

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

Similarly to Section 2.4.2, the term $\log \hat{p}(\boldsymbol{x})$ does not depend on the parameters of the model; thus, the following quantity can be minimized equivalently:

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

Once again, maximizing the likelihood of the observed data in $\mathbb{D}$ is equivalent to minimizing the negative log-likelihood with ERM:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} -\log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}).$$

Two prescribed DGMs of relevance to this thesis is are described in the following. Besides those discussed below, other prescribed DGMs include for example Sigmoid Belief Networks [Nea92], and Flow-based models [RM15].

Autoregressive Models

An Autoregressive (AR) model is a prescribed generative model which decomposes the target distribution as a product of conditionals according the chain rule of probability:

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

where $\boldsymbol{x} = (x_1, x_2, \ldots)$ is a sequence of random variables, $x_{<i} = (x_1, x_2, \ldots, x_{i-1})$, and $p_{\boldsymbol{\theta}}$ is a NN usually shared across each conditional. Clearly, the main requirement to apply an AR decomposition is that $\boldsymbol{x}$ can be decomposed as a sequence of simpler conditionals. Inference in AR models is achieved by forward propagating the input through each network in the order given by the decomposition, and summing up the intermediate log-probabilities. To generate a data point with a trained AR model, it is sufficient to sample the output distribution inferred by the network in the order given by the chain rule decomposition. Given their sequential nature, autoregressive models are often implemented with Recurrent Neural Networks (see Section 3.3); but in principle, any neural network that takes two inputs (the current input of the conditional x_i and a summarization of $x_{<i}$) can be used. Another family of autoregressive models use neural networks with *masking* techniques to constrain their output to follow a given chain rule order. For example, the model by Germain et al. [Ger+15] uses autoencoders, while van den Oord et al. [van+16] use convolutional layers.

Variational Autoencoders

A Variational Autoencoder (VAE) [KW14] is a prescribed DGMs which originates from the family of *latent variable models*. In latent variable models, a set of latent variables $\boldsymbol{z} \in \mathbb{R}^z$, also called explaining factors, is incorporated to the data distribution by marginalization as follows:

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x} \mid \boldsymbol{z})\, q(\boldsymbol{z})\, d\boldsymbol{z}.$$

In the above formula, $p(\boldsymbol{x} \mid \boldsymbol{z})$ is a *decoding distribution* and $q(\boldsymbol{z})$ 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 cannot be observed in general, VAEs introduce an *encoding distribution* $q(\boldsymbol{z} \mid \boldsymbol{x})$ to produce latent variables given an observed 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:

$$\mathrm{ELBO}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z} \sim q(\boldsymbol{z} \mid \boldsymbol{x})} \left[\log p(\boldsymbol{x} \mid \boldsymbol{z})\right] - \mathrm{KLD}(q(\boldsymbol{z} \mid \boldsymbol{x}) \parallel q(\boldsymbol{z})),$$

and such that $\log p(\boldsymbol{x}) \geq \mathrm{ELBO}(\boldsymbol{x})$. 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 $p(\boldsymbol{x} \mid \boldsymbol{z})$ under the encoding distribution (the first term), while making the encoding distribution $q(\boldsymbol{z} \mid \boldsymbol{x})$ close to the prior distribution $q(\boldsymbol{z})$ at the same time (as imposed by the KLD term). In practice, $q(\boldsymbol{z})$ is chosen to be a standard Gaussian $\mathbb{N}(\boldsymbol{0}, \boldsymbol{I})$ with unit covariance matrix $\boldsymbol{I}$. The encoding distribution $q(\boldsymbol{z} \mid \boldsymbol{x})$ is also a Gaussian distribution, whose mean and standard deviation are given by two different neural networks (which share some parameters but not the output layers). Specifically, $q_{\boldsymbol{\psi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathbb{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ with $\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \in \mathbb{R}^z$, such that $\boldsymbol{\mu} = f(\boldsymbol{x})$, $\boldsymbol{\sigma}^2 = g(\boldsymbol{x})$. The fact that both the prior and the encoding distribution are Gaussian gives the KLD term of the ELBO in closed form. The decoding distribution $p(\boldsymbol{x} \mid \boldsymbol{z})$ is implemented as another deep NN $p_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$. The loss function minimized by a VAE is thus the following:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{x}) = -\log p_{\boldsymbol{\phi}}(\boldsymbol{x} \mid \boldsymbol{z}) + \mathrm{KLD}(q_{\boldsymbol{\psi}}(\boldsymbol{z} \mid \boldsymbol{x}) \parallel \mathbb{N}(\boldsymbol{0}, \boldsymbol{I})),$$

with $\boldsymbol{z} \sim \mathbb{N}(f(\boldsymbol{x}), g(\boldsymbol{x}))$ and $\boldsymbol{\theta} = (\boldsymbol{\psi}, \boldsymbol{\phi})$. The leftmost term is a likelihood-based reconstruction loss, and the rightmost 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 $\boldsymbol{x}$ 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 $\boldsymbol{z}$. This in turn is given to the decoder network, which outputs a reconstruction $\boldsymbol{r}$. One major issue with this formulation is that the model cannot be trained with SGD,

since the gradient of a stochastic operation (sampling from the encoding distribution) is not defined. Thus, the sampling process is reparameterized as $\mathbb{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2) = \boldsymbol{\mu} + \boldsymbol{\varepsilon}\boldsymbol{\sigma}^2$, with $\boldsymbol{\varepsilon} \in \mathbb{R}^z \sim \mathbb{N}(\mathbf{0}, \boldsymbol{I})$. This way, the stochastic operation is independent of the input, and the gradient computation becomes deterministic. The reparameterized model is shown in Figure 2.5. The VAE has several interesting properties: firstly, the latent

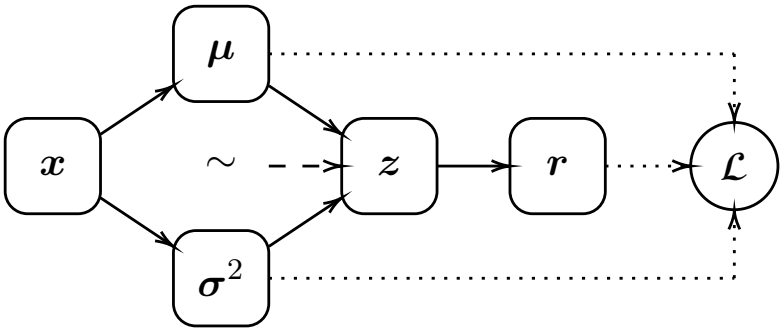


FIGURE 2.5: A Variational Autoencoder.

space learned by a 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 generative modalities: an unconstrained one, which can be achieved by discarding the encoder network and starting the generative process by sampling from the prior $\mathbb{N}(\mathbf{0}, \boldsymbol{I})$; and a conditional one, which is obtained by running an input $\boldsymbol{x}$ through the entire network. This last modality is generative in the sense that the network outputs a variation (not an identical copy) of the input, due to the stochasticity induced by sampling the learned encoding distribution.

2.7.2 Implicit Deep Generative Models

Implicit models [ML16] learn a stochastic procedure that generates data similar to that of the training distribution. The most salient characteristic of implicit DGMs is that they are not trained to minimize the negative log-likelihood of the training set. Instead, an implicit model consists of a deterministic **generator** function $G_{\boldsymbol{\psi}} : \mathbb{R}^z \to \mathbb{R}^d$ parameterized by weights $\boldsymbol{\psi}$, which maps latent variables (obtained by some prior $q(\boldsymbol{z})$) into data samples:

$$\tilde{\boldsymbol{x}} = G_{\boldsymbol{\psi}}(\boldsymbol{z}),$$

where $\boldsymbol{z} \sim q(\boldsymbol{z})$. The mapping function $G_{\boldsymbol{\psi}}$ (which is usually a deep NN) induces a density over the data, but does not give its explicit form: one can observe samples from the density, but cannot compute their probability. This implies that MLE and other MLE-based techniques such as variational learning are not applicable anymore.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) [Goo+14] solve the problem above with a technique known as *adversarial training*. Specifically, they introduce a function

$D_{\boldsymbol{\phi}} : \mathbb{R}^d \to (0,1) \subset \mathbb{R}$, parameterized by weights $\boldsymbol{\phi}$, called **discriminator**. The discriminator is basically a binary classifier that is given either samples from the dataset or from the generator, and its purpose is to classify their origin correctly. The whole process translates into the following min-max objective function:

$$\min_{\boldsymbol{\psi}} \max_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{x}\sim\hat{p}(\boldsymbol{x})} \left[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})} \left[\log\left(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\psi}}(\boldsymbol{z}))\right)\right],$$

where in practice $G_{\boldsymbol{\psi}}$ is trained to improve the samples it generates, so that $D_{\boldsymbol{\phi}}$ classifies them wrongly as if they were real; conversely, $D_{\boldsymbol{\phi}}$ is trained to improve at better distinguishing real samples from samples generated by $G_{\boldsymbol{\psi}}$. The whole architecture is trained to reach a Nash equilibrium, so that both $G_{\boldsymbol{\psi}}$ and $D_{\boldsymbol{\phi}}$ cannot improve further. During adversarial training, the parameters of the discriminator and the generator are updated in turns. First, the discriminator is trained to minimize the following loss function:

$$\mathcal{L}_D(\boldsymbol{\theta}, \boldsymbol{x}) = -\log D_{\boldsymbol{\phi}}(\boldsymbol{x}) - \log\left(1 - D_{\boldsymbol{\phi}}(\tilde{\boldsymbol{x}})\right),$$

where $\tilde{\boldsymbol{x}} = G_{\boldsymbol{\psi}}(\boldsymbol{z})$ with $\boldsymbol{z} \sim q(\boldsymbol{z})$ and $\boldsymbol{\theta} = (\boldsymbol{\psi}, \boldsymbol{\phi})$. Thus, the discriminator solves a binary classification task, where examples are labeled as positive if real, and negative if generated. After the discriminator has been optimized, the generator is trained to minimize the following loss:

$$\mathcal{L}_G(\boldsymbol{\theta}, \boldsymbol{x}) = \log\left(1 - D_{\boldsymbol{\phi}}(\tilde{\boldsymbol{x}})\right),$$

which corresponds to making the discriminator classify the generated sampled as real. Figure 2.6 shows a schematic representation of a GAN. Notice that, in order to train

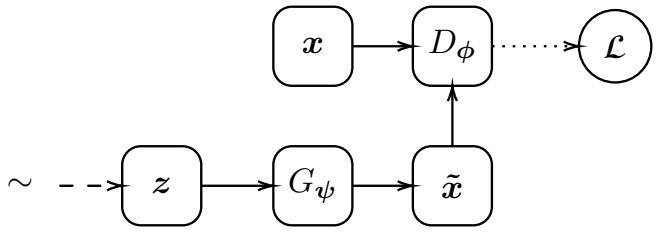


FIGURE 2.6: A Generative Adversarial Network. Here, the tilde symbol over the vector yielded by the generator indicates that it does not come from the training set, but it is generated. The discriminator $D_{\boldsymbol{\phi}}$ must distinguish between generated and real samples (indicated without the tilde).

the architecture end-to-end, the samples taken from the generator of a GAN must be differentiable. GANs, in their original formulation, suffer from two main problems: one is *mode collapse*, which means that in case of multi-modal distributions, the generator tends to focus only on one mode, ignoring the others; another is *discriminator saturation*, which means that when the discriminator becomes very confident at discriminating the samples, the gradient of the generator vanishes [Goo+14]. These two issues have been addressed by several works in the literature [Sal+16].

Chapter 3

Deep Learning in Structured Domains

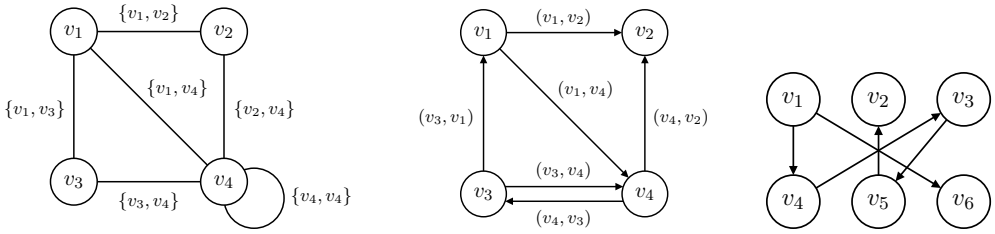
A **structured domain** is a data domain whose elements are formed by a set of atomic *entities*, and the *relations* between them. Structured data is common in several fields, such as biology, chemistry, finance, social networks, and many more. Typical examples are sequences such as time-series data, hierarchical structures such as parse trees, and 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 can be arbitrary. 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 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** [BM+76]. Informally, a graph is a collection of *vertices* (the entities) connected through a collection of *edges* (the relations). In the literature, vertices are sometimes called *nodes*, while edges are also referred to as *arcs* or *links*. Formally, a graph with n vertices is a pair

$$\mathrm{g} = \langle \mathcal{V}_\mathrm{g}, \mathcal{E}_\mathrm{g} \rangle,$$

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



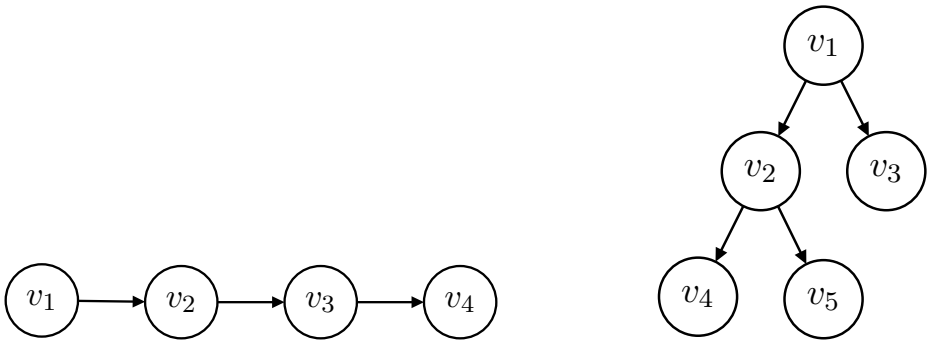
(A) An undirected graph. (B) A directed graph. (C) A bipartite graph.

FIGURE 3.1: Three examples of graphs.

Directed Graphs A **directed graph** is one where the edges are ordered pairs of vertices, or equivalently one where $\mathcal{E}_\mathsf{g} \subseteq \mathcal{V}_\mathsf{g} \times \mathcal{V}_\mathsf{g}$. 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 $v \in \mathcal{V}_\mathsf{g}$, the set of all vertices from which an edge reaches v is called *predecessors* set, and is defined as $\mathcal{N}_\rightarrow(v) = \{u \in \mathcal{V}_\mathsf{g} \mid (u, v) \in \mathcal{E}_\mathsf{g}\}$. 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 $\mathcal{N}_\leftarrow(v) = \{u \in \mathcal{V}_\mathsf{g} \mid (v, u) \in \mathcal{E}_\mathsf{g}\}$. 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: $\mathcal{N}(v) = \mathcal{N}_\rightarrow(v) \bigcup \mathcal{N}_\leftarrow(v)$. Alternatively, one can view the neighborhood as a function $\mathcal{N} : \mathcal{V}_\mathsf{g} \rightarrow 2^{\mathcal{V}_\mathsf{g}}$ 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 $\mathcal{V}_\mathsf{g}$ in two disjoint subsets $\mathcal{V}_\mathsf{g}^+$ and $\mathcal{V}_\mathsf{g}^-$, such that $(u, v) \in \mathcal{E}_\mathsf{g}$ if and only if either $u \in \mathcal{V}_\mathsf{g}^+$ and $v \in \mathcal{V}_\mathsf{g}^-$, or alternatively $v \in \mathcal{V}_\mathsf{g}^+$ and $u \in \mathcal{V}_\mathsf{g}^-$. Figure 3.1c shows an example of bipartite graph, where $\mathcal{V}_\mathsf{g}^+ = \{v_1, v_2, v_3\}$ and $\mathcal{V}_\mathsf{g}^- = \{v_4, v_5, v_6\}$.

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, *i.e.* $(v_i, v_{i+1}) \in \mathcal{E}_\mathsf{g}$, $\forall i = 1, \ldots, i-1$. 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. If, given two vertices $u, v \in \mathcal{V}_\mathsf{g}$ 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. The *shortest path* from a node u to a node v is the path, among all paths from u to v, with the smallest length. We indicate it with the notation $u \xrightarrow{\mathsf{g}} v$. 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) A sequence. (B) A tree.

FIGURE 3.2: Special classes of graphs.

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 $\mathtt{t}$ 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 $(u, v) \in \mathcal{E}_\mathtt{t}$, 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, except for 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. For example, in *binary* trees (such as the one in Figure 3.2b), each node can have a left and right child. Trees have a recursive structure: every node $v \in \mathcal{V}_\mathtt{t}$ is itself the root of a tree, called *sub-tree of* $\mathtt{t}$ *rooted at* v, and indicated as $\mathtt{t}_v$. If $\mathcal{V}_{t_v}$ contains only v, v is called a *leaf*. Trees are useful to encode *hierarchical* relationships among nodes.

A graph $\mathtt{s}$ 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 $\mathcal{V}_\mathtt{s} = (v_1, v_2, \ldots, v_n)$. 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** $\mathtt{h} = \langle \mathcal{V}_\mathtt{h}, \mathcal{E}_\mathtt{h} \rangle$ of a graph $\mathtt{g}$ is any graph for which $\mathcal{V}_\mathtt{h} \subseteq \mathcal{V}_\mathtt{g}$ and $\mathcal{E}_\mathtt{h} \subseteq \mathcal{E}_\mathtt{g}$. If $\mathcal{E}_\mathtt{h} = (\mathcal{V}_\mathtt{h} \times \mathcal{V}_\mathtt{h}) \bigcap \mathcal{E}_\mathtt{g}$, or equivalently, if $\mathcal{V}_\mathtt{h}$ contains only vertices that are endpoints in $\mathcal{E}_\mathtt{g}$, the resulting subgraph is called **induced subgraph**. We indicate the subgraph of $\mathtt{g}$ induced by $\mathtt{h}$ with the notation $\mathtt{g}_{[\mathtt{h}]}$.

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 $\mathtt{g}$ 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 $\langle \mathtt{g}, \chi, \xi \rangle$, where $\chi : \mathcal{V}_{\mathtt{g}} \to \mathbb{R}^d$ is a mapping from the space of vertices to a space of d-dimensional *vertex features*, and $\xi : \mathcal{E}_{\mathtt{g}} \to \mathbb{R}^e$, 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. In most cases, we omit to define χ and ξ explicitly, and provide the vertex and edge features directly as sets, *e.g.* $\boldsymbol{x}_{\mathtt{g}} = \left\{ \boldsymbol{x}_{[v]} \in \mathbb{R}^d \mid v \in \mathcal{V}_{\mathtt{g}} \right\}$ for the vertex features, and $\boldsymbol{e}_{\mathtt{g}} = \left\{ \boldsymbol{e}_{[u,v]} \in \mathbb{R}^e \mid (u,v) \in \mathcal{E}_{\mathtt{g}} \right\}$ for the edge features. In this case, an attributed graph is a triplet:

$$\mathtt{G} = \langle \mathtt{g}, \boldsymbol{x}_{\mathtt{g}}, \boldsymbol{e}_{\mathtt{g}} \rangle.$$

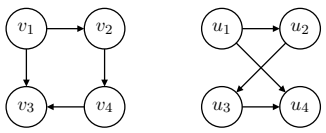
If some ordering of the vertices and edges is assumed, we can represent equivalently χ as a matrix $\boldsymbol{X}_{\mathtt{g}} \in \mathbb{R}^{n \times d}$ where the i-th row contains the vertex features of the i-th vertex; analogously, we can define ξ as a matrix of edge features $\boldsymbol{E}_{\mathtt{g}} \in \mathbb{R}^{m \times e}$. In this case, an attributed graph is a triplet:

$$\mathtt{G} = \langle \mathtt{g}, \boldsymbol{X}_{\mathtt{g}}, \boldsymbol{E}_{\mathtt{g}} \rangle$$

. In this thesis, we use both notations interchangeably.

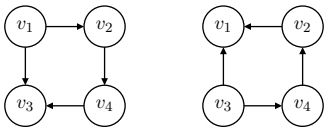
3.1.2 Isomorphism, Automorphism, and Canonization

An **isomorphism** between two graphs $\mathtt{g}$ and $\mathtt{h}$ is a bijection $\vartheta : \mathcal{V}_{\mathtt{g}} \to \mathcal{V}_{\mathtt{h}}$ such that $(u,v) \in \mathcal{E}_{\mathtt{g}}$ if and only if $(\vartheta(u), \vartheta(v)) \in \mathcal{E}_{\mathtt{h}}$. 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 ϑ bijection. An **automorphism** $\pi : \mathcal{V}_{\mathtt{g}} \to \mathcal{V}_{\mathtt{g}}$ is an isomorphism between $\mathtt{g}$ and itself. Since π is essentially a permutation of the vertex set, it follows that a graph always has up to $n!$ possible automorphism. Intuitively, and similarly to graph isomorphism, graph automorphism conveys 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 isomorphism and automorphism is the problem of **graph canonization**, where a canonical ordering (or form) of the graph vertices is sought, such that every graph $\mathtt{h}$ isomorphic to a given graph $\mathtt{g}$ 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 graph isomorphism and automorphism.

3.1.3 Graphs 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 a_{ij} of its corresponding adjacency matrix $\boldsymbol{A}_{\mathsf{g}} \in \mathbb{R}^{n \times n}$ are defined as follows:

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

Note that the diagonal entries 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 $a_{ij} = a_{ji},\ \forall\, i, j = 1, \ldots, n$. Adjacency matrices make some calculations of graph properties particularly convenient: for example, the in-degree and out-degree of a vertex $v_j \in \mathcal{V}_{\mathsf{g}}$ can be obtained by performing row-wise and column-wise sums on $\boldsymbol{A}_{\mathsf{g}}$:

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

Adjacency matrices are also useful to understand concepts such as graph automorphism: 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 of graphs are the *Laplacian matrix* $\boldsymbol{L}_{\mathsf{g}} \in \mathbb{R}^{n \times n} = \boldsymbol{D}_{\mathsf{g}} - \boldsymbol{A}_{\mathsf{g}}$, and the *symmetric normalized Laplacian matrix* $\tilde{\boldsymbol{L}}_{\mathsf{g}} \in \mathbb{R}^{n \times n} = \boldsymbol{I} - \boldsymbol{D}_{\mathsf{g}}^{-\frac{1}{2}} \boldsymbol{A}_{\mathsf{g}} \boldsymbol{D}_{\mathsf{g}}^{-\frac{1}{2}}$. In both definitions, the matrix $\boldsymbol{D}_{\mathsf{g}} \in \mathbb{R}^{n \times n}$ is the *degree matrix*, where all entries are zero except the diagonal entries, for which $d_{ii} = \text{degree}(v_i)$. These matrices provide information about the graph connectivity through their eigenvalues and eigenvectors. Following, we shall use $\boldsymbol{A}_{\mathsf{g}}$ instead of g to denote the structure of the graph whenever needed from the context.

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 $\mathfrak{T} : \mathcal{X} \to \mathcal{Y}$ 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 $\mathfrak{T} = \mathfrak{T}_{\text{enc}} \circ \mathfrak{T}_{\text{out}}$, where:

- $\mathfrak{T}_{\text{enc}}$ is called *encoding function* or *state transition function*, and it 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 its corresponding element and of the elements it depends on;
- $\mathfrak{T}_{\text{out}}$ 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 $\mathbb{D}$ consisting of n training pairs, we distinguish two learning problems:

- in *structure-to-structure* tasks, the dataset is composed of pairs of attributed graphs $\mathbb{D} = \{(\mathtt{G}_{(i)}, \mathtt{H}_{(i)})\}_{i=1}^{n}$, where $\mathtt{G} = \langle \mathtt{g}, \boldsymbol{x}_{\mathtt{g}}, \boldsymbol{e}_{\mathtt{g}} \rangle$ is an input graph, $\mathtt{H} = \langle \mathtt{h}, \boldsymbol{y}_{\mathtt{h}}, \boldsymbol{e}_{\mathtt{h}} \rangle$ is an output graph, and the two underlying unattributed graphs $\mathtt{g}$ and $\mathtt{h}$ are isomorphic under a bijection ϑ. The task is to predict the target associated to an output graph vertex, given the corresponding features of its isomorphic vertex. The objective function minimized in these tasks is the following:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} -\log p(\mathtt{H}_{(i)} \mid \mathtt{G}_{(i)}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{v \in \mathcal{V}_{\mathtt{g}_{(i)}}} -\log p(\boldsymbol{y}_{[\vartheta(v)]} \mid \boldsymbol{x}_{[v]});$$

- in *structure-to-element* tasks, the dataset has the form $\mathbb{D} = \{(\mathtt{G}_{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{n}$, where $\mathtt{G} = \langle \mathtt{g}, \boldsymbol{x}_{\mathtt{g}}, \boldsymbol{e}_{\mathtt{g}} \rangle$ is an input graph and $\boldsymbol{y} \in \mathbb{R}^{y}$ is an output vector. The task is to predict a single output vector (or scalar) from the structure and the features of $\mathtt{G}$. The objective function minimized in these tasks is the following:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} -\log p(\boldsymbol{y}^{(i)} \mid \mathtt{G}_{(i)}).$$

  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 $\boldsymbol{y}$. 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 the overall structure. This process is similar to

the localized processing of images performed by CNNs, which work 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 $\mathtt{S}$ be an attributed sequence of length m with elements $\mathcal{V}_{\mathtt{S}} = (v_1, v_2, \dots, v_m)$, and let $\boldsymbol{x}_{\mathtt{S}} = \big(\boldsymbol{x}_{[1]}, \boldsymbol{x}_{[2]}, \dots, \boldsymbol{x}_{[m]}\big)$ be its element features. Here, we slightly abuse the notation $\boldsymbol{x}_{[v_t]}$ in favor of $\boldsymbol{x}_{[t]}$, since sequence elements are ordered. Let us assume a supervised setting, and the availability of an isomorphic attributed sequence $\mathtt{R}$ with targets $\boldsymbol{y}_{\mathtt{R}} = \Big(\boldsymbol{y}_{[1]}, \boldsymbol{y}_{[2]}, \dots, \boldsymbol{y}_{[m]}\Big)$. The state transition function of a RNN, applied locally to each sequence element, has the following general form:

$$\boldsymbol{h}_{[t]} = \begin{cases} \mathbf{0} & \text{if } t = 0 \\ \mathfrak{T}_{\text{enc}}(\boldsymbol{x}_{[t]}, \boldsymbol{h}_{[t-1]}) & \text{otherwise,} \end{cases}$$

where $\boldsymbol{h}_{[t]} \in \mathbb{R}^h$ is a state vector, also known as **hidden state**, and $\mathbf{0}$ is a zero vector. The calculation of the hidden state performed by the state transition function $\mathfrak{T}_{\text{enc}}$ 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 the zero vector $\boldsymbol{h}_{[0]} = \mathbf{0}$. 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 [SS95]. 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 an attributed sequence $\mathtt{S}$ with features $\boldsymbol{x}_{\mathtt{S}}$, the original implementation of the state transition function of a RNN is defined as follows[1]:

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

The above is also called **recurrent layer**. The weight matrices $\boldsymbol{W} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{U} \in \mathbb{R}^{h \times h}$, 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:

$$\boldsymbol{o}_{[t]} = g(\boldsymbol{h}_{[t]}), \ \forall t = 1, \dots, m,$$

where g 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:

$$\boldsymbol{o} = g(\boldsymbol{h}_{[m]}).$$

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 $\boldsymbol{o}_{[t]}$ to the isomorphic sequence element $\boldsymbol{y}_{[t]}$ in the target sequence with the loss function $\mathcal{L}$. The loss is summed up over all the elements in the sequence, and averaged over all sequences in the dataset to minimize the MLE objective. Notice that it is possible to stack multiple recurrent layers and create deep RNNs by feeding the hidden state produced by a recurrent layer as input to a subsequent recurrent layer, other than to the next step of the recurrence. In these cases, the output is computed after by the last recurrent layer in the stack. RNNs can be also adapted to learn structure-to-structure distributions of the kind $p(\mathtt{R} \,|\, \mathtt{S})$, where the underlying unattributed graphs $\mathtt{r}$ and $\mathtt{s}$ 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 $\mathtt{S}$ (for example, its last hidden state as seen above); the other acts as a decoder of the target sequence $\mathtt{R}$, 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) [Wer88], 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

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

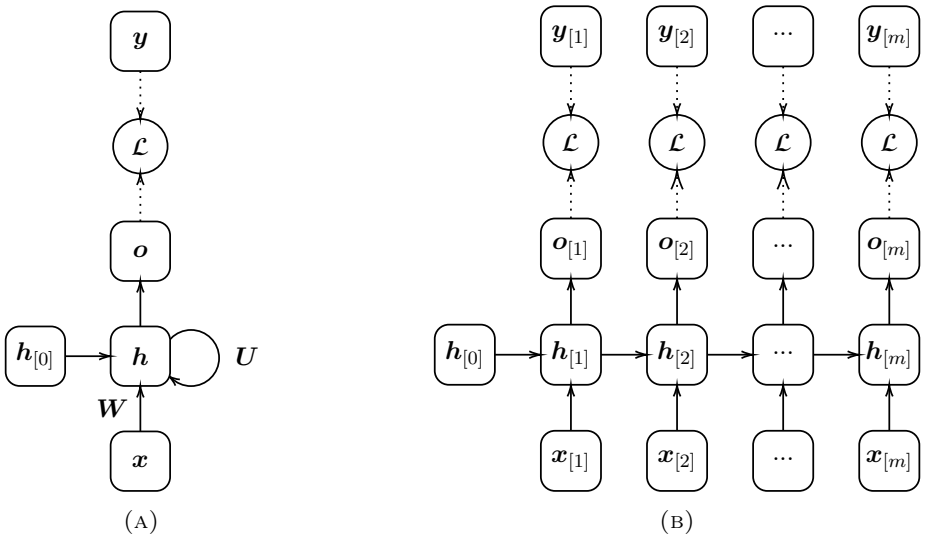


FIGURE 3.4: (A): An example of recurrent neural network that can learn a structure-to-structure task. (B): the same network unfolded over a training pair of sequences of length m.

update requires $\mathcal{O}(mb)$ 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 long sequences, and can lead to instabilities like gradient vanishing. Thus, in practical settings, faster BPTT variants are often used, such as truncated BPTT [Jae02].

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.* [BSF94]), and is mostly due to the gradient vanishing or exploding problems. While gradient exploding can be dealt by clipping the norm or the values of the gradients, 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*, that is, controlling 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 these purposes adaptively, driven by data. The most used RNN variant that implements gating mechanisms is the Long Short-Term Memory (LSTM) [HS97]. A LSTM is composed of a *cell* $\boldsymbol{c} \in \mathbb{R}^h$, an *input gate* $\boldsymbol{i} \in \mathbb{R}^h$, a *forget gate* $\boldsymbol{f} \in \mathbb{R}^h$, and an *output gate* $\boldsymbol{g} \in \mathbb{R}^h$. Assuming an input sequence element $\boldsymbol{x}_{[t]} \in \mathbb{R}^d$, the hidden state $\boldsymbol{h}_{[t]} \in \mathbb{R}^h$ of a LSTM is computed

as follows:

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

where $\odot$ is the Hadamard (element-wise) product between matrices. Notice that the weight matrices $\boldsymbol{W}_i \in \mathbb{R}^{d\times h}$ and $\boldsymbol{U}_i \in \mathbb{R}^{h\times h}$ with $i = 1, \dots, 4$ 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) [Cho+14] gating mechanism is a lightweight alternative to LSTM which uses fewer parameters, though it is slightly less powerful [GJ20]. A GRU uses two gates, an *update* gate $\boldsymbol{u} \in \mathbb{R}^h$ and a *reset* gate $\boldsymbol{r} \in \mathbb{R}^h$, and computes the hidden state as follows:

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

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

Besides being used as supervised models, RNNs can be also used as generative models of sequences [Gra13]. Specifically, given a dataset $\mathbb{D} = \{\mathtt{S}_{(i)}\}_{i=1}^{n}$ of sequences, they can be trained to learn a model $p_{\boldsymbol{\theta}}$ of the underlying distribution $p(\mathtt{S})$ as follows:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{\mathtt{S}\in\mathbb{D}} -\log p_{\boldsymbol{\theta}}(\mathtt{S}) = \frac{1}{n} \sum_{\mathtt{s}\in\mathbb{D}} \sum_{t=1}^{|\mathtt{s}|} -\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_{[t]} \mid \boldsymbol{x}_{[t-1]}, \boldsymbol{h}_{[t-1]}),$$

where $|\mathtt{s}|$ indicates the sequence length, $\boldsymbol{x}_{[0]}$ is a special *start of sequence symbol* $\langle \mathsf{S} \rangle$, and $\boldsymbol{h}_{[0]} = \mathbf{0}$ as usual. Once the network is trained, a sequence can be generated one element at a time. The process is initialized by feeding the start of sequence token

$\boldsymbol{x}_{[0]}$ and the initial hidden state $\boldsymbol{h}_{[0]}$, and proceeds repeating the following instructions until an *end of sequence token* $\langle \mathsf{E} \rangle$ is predicted by the network:

- update the current state $\boldsymbol{h}_{[t]}$ with the RNN and predict an output $\boldsymbol{o}_{[t]}$, which corresponds to a conditional distribution over the possible sequence elements. Sample a sequence element $\tilde{\boldsymbol{x}}_{[t]}$ according to this distribution;
- feed the sampled element $\tilde{\boldsymbol{x}}_{[t]}$ and the updated state $\boldsymbol{h}_{[t]}$ to the network and repeat.

We call this process *autoregressive sampling mode*. The process is depicted in Figure 3.5. During training, the sampling process breaks the differentiability of the model.

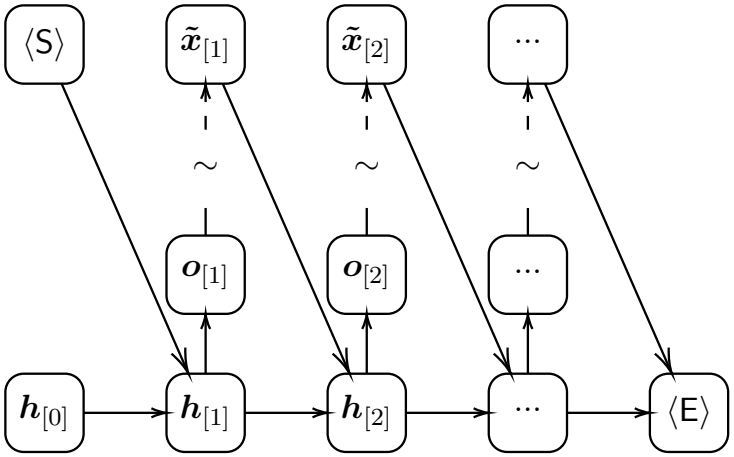


FIGURE 3.5: An example of training a recurrent neural network for learning an autoregressive distribution. The dashed arrows indicate non-differentiable operations.

Hence, one resorts to reparameterization [JGP17] or differentiates the output $\boldsymbol{o}_{[t]}$ instead of the hard sample [BLC13]. Another option during training is *teacher forcing* [WZ89]: in this case, the knowledge of the elements of the sequence during training is exploited, by feeding the ground truth sequence element (instead of the sampled value) as the input for the next sequence. 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.

3.4 Recursive Neural Networks

A Recursive Neural Network (RecNN) [SS97; FGS98] is a NN architecture that can adaptively process hierarchical data. Using trees as an example, let $\mathtt{T}$ be an attributed tree with m nodes $\mathcal{V}_{\mathtt{t}} = \{v_1, v_2, \ldots, v_m\}$, and $\boldsymbol{x}_{\mathtt{t}} = \left\{\boldsymbol{x}_{[v]} \in \mathbb{R}^d \mid v \in \mathcal{V}_{\mathtt{t}}\right\}$ be its set node of features. The state transition function of a RecNN, applied locally to each node

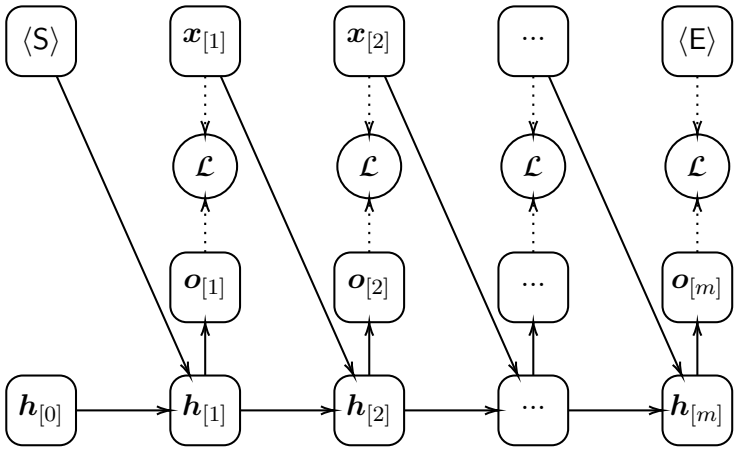


FIGURE 3.6: The teacher forcing strategy for training a sequence generator with RNNs.

$v \in \mathcal{V}_\mathfrak{t}$, is the following:

$$\boldsymbol{h}_{[v]} = \begin{cases} \mathbf{0} & \text{if } \mathrm{ch}(v) = \emptyset \\ \mathfrak{T}_{\text{enc}}(\boldsymbol{x}_{[v]}, \boldsymbol{h}_{[\mathfrak{t}_v]}) & \text{otherwise,} \end{cases}$$

where $\boldsymbol{h}_{[\mathfrak{t}_v]} \in \mathbb{R}^h$ is the hidden state of the sub-tree rooted at v. As with RNNs, the state transition function $\mathfrak{T}_{\text{enc}}$ 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 [SS97] and the development of a general framework for learning with tree-structured data [FGS98], 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 been applied fruitfully in several fields, including among others Cheminformatics [MSS07; LPB13], NLP [Cos+; Cos+03; Stu+03; Soc+13] and scene parsing [Soc+11]. 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 [HF99], and from labeled DPAGs to real values [HMS05] have been proved. RecNNs can be trained with Backpropagation Through Structure (BPTS) [GK96], a variant of BPTT where the error of the network is propagated to the tree structure.

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:

$$\boldsymbol{h}_{[v]} = \sigma\left(\boldsymbol{W}\boldsymbol{x}_{[v]} + \boldsymbol{U}_{\mathrm{l}}\boldsymbol{h}_{[\mathrm{l}(v)]} + \boldsymbol{U}_{\mathrm{r}}\boldsymbol{h}_{[\mathrm{r}(v)]}\right),\ \forall v \in \mathcal{V}_{\mathtt{t}}.$$

In the above formula, g can be any hidden activation function, and $\boldsymbol{W} \in \mathbb{R}^{d\times h}$, $\boldsymbol{U}_{\mathrm{l}}$, and $\boldsymbol{U}_{\mathrm{r}} \in \mathbb{R}^{h\times h}$, 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 3.7 shows the unfolded RecNN over the tree. The final output of the entire structure is obtained using the hidden state of the root node v_1 as:

$$\boldsymbol{o} = g\left(\boldsymbol{h}_{[v_1]}\right),$$

where $\boldsymbol{o} \in \mathbb{R}^y$ is the output of the network and g 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:

$$\boldsymbol{o}_{[v]} = g\left(\boldsymbol{h}_{[v]}\right),\ \forall v \in \mathcal{V}_{\mathtt{t}}.$$

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

3.5 Deep Graph Networks

The RNN and RecNN models presented in Sections 3.3 and Section 3.4 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

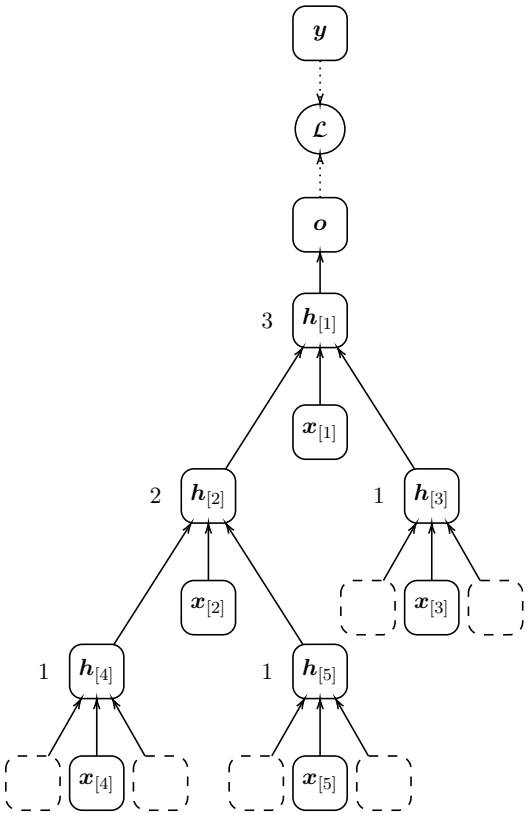


FIGURE 3.7: A recursive neural network unfolded over the tree of Figure 3.2b for a structure-to-element task. The number at the left of the hidden states indicates the order in which they are calculated. Notice the initialization of the hidden states at the leaves (indicated by dashed boxes).

the 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 an attributed graph $\mathtt{G}$ with vertex features $\boldsymbol{x}_{\mathtt{g}} = \{\boldsymbol{x}_{[v]} \mid v \in \mathcal{V}_{\mathtt{g}}\}$, the state transition function computed by a DGN, applied locally to each vertex of $v \in \mathcal{V}_{\mathtt{g}}$, has the following form:

$$\boldsymbol{h}_{[v]}^{(\ell)} = \begin{cases} \boldsymbol{x}_{[v]} & \text{if } \ell = 0 \\ \mathfrak{T}_{\text{enc}}^{(\ell)}(\boldsymbol{x}_{[v]}, \boldsymbol{h}_{[v]}^{(\ell-1)}) & \text{otherwise,} \end{cases}$$

where $\boldsymbol{h}_{[v]}^{(\ell)} \in \mathbb{R}^h$ is now the state of vertex v at iteration ℓ, and $\mathfrak{T}_{\text{enc}} = \mathfrak{T}_{\text{enc}}^1 \circ \mathfrak{T}_{\text{enc}}^2 \circ \ldots \circ \mathfrak{T}_{\text{enc}}^{(\ell)}$. 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 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).

3.5.1 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 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 [Mic09]. Context diffusion in a DGN is obtained through **message passing**, *i.e.* by repeatedly applying the following procedures:

- each node constructs a *message* vector from 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 3.8 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.

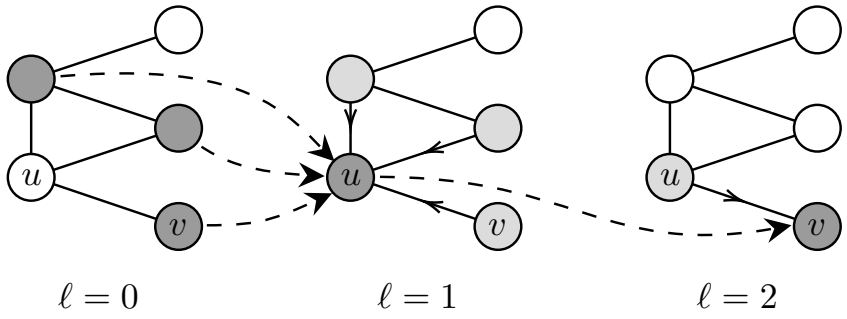


FIGURE 3.8: Context diffusion through message passing. Directed edges represent messages (*e.g.* from node u to v at iteration $\ell = 2$). Dashed arrows represent the implicit contextual information received by a node (in dark gray) through the messages from its neighbors (in light gray). Focusing on node v, its context at iteration $\ell = 2$ is composed all the dark gray nodes (including v itself).

Recursive Approaches

In the recursive approach to context diffusion, message passing is formulated as a dynamical system. In this case, the state transition function is recursive, meaning that $\mathfrak{T}_{\text{enc}} = \mathfrak{T}^{(1)}_{\text{enc}} = \mathfrak{T}^{(2)}_{\text{enc}} = \ldots = \mathfrak{T}^{(\ell)}_{\text{enc}}$. Practically speaking, the mutual dependencies between hidden states are modeled with a single recurrent layer, which is run indefinitely until convergence. Some well-known representatives of this paradigm are the Graph Neural Network [Sca+09], the Graph Echo State Network [GM10], and the more recent Fast and Deep Graph Neural Network [GM20]. To ensure convergence, these approaches impose contractive dynamics 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 [LZB+16], where, differently from Scarselli et al. [Sca+09], 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 [MP07].

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. In practice, the state transition function is no more recursive, but changes at every layer. 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 [Mic09], where it is shown formally that the context of a node increases as a function of the network depth, up to including all the nodes in the graph. Feed-forward approaches

are nowadays the main paradigm to design DGNs, due to their simplicity, efficiency, and performance on 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 [BSF94; LHW18]. For the rest of this thesis, all the DGNs used shall be feed-forward.

Constructive Approaches

Constructive approaches are a special case of feed-forward models, in which training is performed layer-wise. The neural network architecture exploited by these models is a modification of the Cascade Correlation algorithm [FL90] for graphs. 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 [MHN18; Bia+00a]. 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 [Mic09], 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 [BEM18].

3.5.2 Building Blocks of Deep Graph Networks

DGNs are built from several architectural components, which we cover in detail in this section. In short, a supervised 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 using the representation obtained by the graph processing layers. 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.

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 an attributed graph $\mathtt{G}$ with n nodes, and its node attributes $\boldsymbol{x}_{\mathtt{g}} = \{\boldsymbol{x}_{[v]} \in \mathbb{R}^d \mid v \in \mathcal{V}_{\mathtt{g}}\}$,

one general formulation of a GCL is the following:

$$\boldsymbol{h}_{[v]}^{(\ell)} = U\left(\boldsymbol{h}_{[v]}^{(\ell-1)}, A\left(\left\{ T(\boldsymbol{h}_{[u]}^{(\ell-1)}) \mid u \in \mathcal{N}(v) \right\}\right)\right), \ \forall v \in \mathcal{V}_{\mathtt{g}}, \tag{3.1}$$

where $\boldsymbol{h}_{[v]}^{(\ell)} \in \mathbb{R}^{h_\ell}$ is the hidden state of the node at layer ℓ, $\boldsymbol{h}_{[v]}^{(\ell-1)} \in \mathbb{R}^{h_{\ell-1}}$ is the hidden state of the node at the previous layer $\ell - 1$, and by convention $\boldsymbol{h}_{[v]}^{(0)} = \boldsymbol{x}_{[v]}$. 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 inside a GCL:

- $T : \mathbb{R}^{h_{\ell-1}} \to \mathbb{R}^{h_{\ell-1}}$ is a *transform* function that applies some transformation to the hidden states of neighbors of node v at layer $\ell-1$. This can be any function, either fixed or adaptive (implemented by a neural network);
- $A : (\mathbb{R}^{h_{\ell-1}} \times \mathbb{R}^{h_{\ell-1}} \times \ldots) \to \mathbb{R}^{h_{\ell-1}}$ is an *aggregation* function that maps a *multiset*[2] 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*;
- $U : (\mathbb{R}^{h_{\ell-1}} \times \mathbb{R}^{h_{\ell-1}}) \to \mathbb{R}^{h_\ell}$ is an *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. Similarly to T, U can also be fixed or adaptive.

The usage of a permutation invariant function to compute the state of the neighbors is crucial, as it allows acquiring 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 of Kipf et al. [KW17], corresponding to the Graph Convolutional Network (GCN) model:

$$\boldsymbol{h}_{[v]}^{(\ell)} = \text{sigmoid}\left(\boldsymbol{W}^{(\ell)} \sum_{u \in \mathcal{N}(v)} \tilde{l}_{uv} \boldsymbol{h}_{[u]}^{(\ell-1)}\right), \ \forall v \in \mathcal{V}_{\mathtt{g}}, \tag{3.2}$$

where $\tilde{l}_{uv}$ is the entry of the symmetric normalized graph Laplacian $\tilde{\boldsymbol{L}}_{\mathtt{g}}$ related to nodes u and v, and:

$$T(\boldsymbol{h}_{[u]}^{(\ell-1)}) = \boldsymbol{t}_{[v]}^{(\ell-1)} = \tilde{l}_{uv}\, \boldsymbol{h}_{[u]}^{(\ell-1)} \tag{3.3}$$

$$A\left(\left\{ \boldsymbol{t}_{[v]}^{(\ell-1)} \mid u \in \mathcal{N}(v) \right\}\right) = \boldsymbol{n}_{[v]}^{(\ell-1)} = \sum_{u \in \mathcal{N}(v)} \boldsymbol{t}_{[v]}^{(\ell-1)} \tag{3.4}$$

$$U(\boldsymbol{h}_{[v]}^{(\ell-1)}, \boldsymbol{n}_{[v]}^{(\ell-1)}) = \boldsymbol{h}_{[v]}^{(\ell)} = \text{sigmoid}\left(\boldsymbol{W}^{(\ell)}\, \boldsymbol{n}_{[v]}^{(\ell-1)}\right). \tag{3.5}$$

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

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 [Zah+17]. 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 RecNNs, 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:

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

where $\boldsymbol{A}_{\mathtt{g}} \in \mathbb{R}^{n\times n}$ is the adjacency matrix of the graph, $\boldsymbol{H}^{(\ell-1)} \in \mathbb{R}^{n\times h}$ are the hidden states computed at layer $\ell-1$ where by convention $\boldsymbol{H}^{(0)} = \boldsymbol{X}_{\mathtt{g}} \in \mathbb{R}^{n\times d}$ is the matrix of node features, and $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{h_{(\ell-1)}\times h_{(\ell)}}$ is the matrix of trainable layer-wise weights. Here, the adjacency matrix substitutes the neighborhood function $\mathcal{N}$, and the adjacency sets are 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 an attributed graph $\mathtt{G}$, we assume a set of edge features of the form $\boldsymbol{e}_{\mathtt{g}} = \left\{\boldsymbol{e}_{[u,v]} \in \mathcal{C} \mid (u,v) \in \mathcal{E}_{\mathtt{g}}\right\}$, 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:

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

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. 3.5 is modified as follows:

$$\boldsymbol{h}_{[v]}^{(\ell)} = \mathrm{sigmoid}\left(\sum_{c\in\mathcal{C}} \boldsymbol{W}_c^{(\ell)} \sum_{u\in\mathcal{N}_c(v)} \tilde{l}_{uv}\, \boldsymbol{h}_{[v]}^{(\ell-1)}\right),\ \forall v \in \mathcal{V}_{\mathtt{g}},$$

where the weight matrices $\boldsymbol{W}_c^{(\ell)}$ are now edge-specific, so that the contributions of the different edge types are weighted adaptively [Mic09; Sch+18]. 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) [Gil+17].

Node Attention Attention mechanisms [BCB15] 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 the neighborhood aggregation, the contribution of the transformed nodes in the neighborhood is weighted by a scalar $a_{uv}^{(\ell)} \in \mathbb{R}$, called *attention score* as follows:

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

The attention scores are derived from *attention coefficients* $w_{vu}^{(\ell)}$, which are essentially similarity scores between the neighbor and the current node, calculated as follows:

$$w_{vu}^{(\ell)} = f(\boldsymbol{h}_{[v]}^{(\ell)}, \boldsymbol{h}_{[u]}^{(\ell)}),$$

where f is an arbitrary neural network. Different attention mechanisms are defined based on how f is implemented (*e.g.* in the Graph Attention Network [Vel+18], the two states are concatenated). 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 3.9 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.

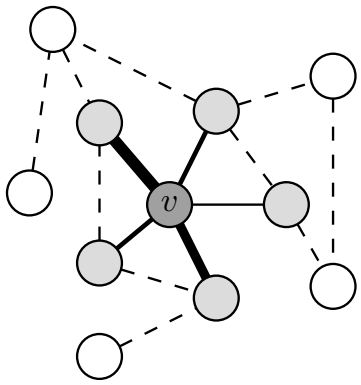


FIGURE 3.9: An example of node attention. Edge thicknesses are not related to the strength of the connection between node v (in dark gray) and its neighbors (in light gray), but it represents the degree of similarity between the node states, quantified in a probabilistic sense by the attention score. 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 importance sampling [GM20], or even extended to sampling a bounded number of nodes which are not necessarily in the immediate neighborhood of the current node [HYL17]. The latter requires adding 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.

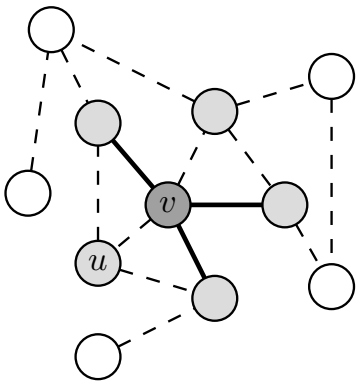


FIGURE 3.10: An example of node sampling. We focus on a node v (in dark gray), and its neighbors (in light gray). Dashed edges connect nodes that are not involved in the neighborhood aggregation; notice how node u is excluded, even though it is effectively a neighbor of v.

Readout Layers

As we have seen, the application of L DGN layers to an attributed graph $\mathtt{G}$ 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 vector 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 embedding**) $\boldsymbol{h}_{[v]}^{(*)}$ for each node as follows:

$$\boldsymbol{h}_{[v]}^{(*)} = \mathrm{R'}_v\left(\left\{\boldsymbol{h}_{[v]}^{(\ell)} \mid \ell = 1, \ldots, L\right\}\right), \ \forall v \in \mathcal{V}_{\mathtt{g}}.$$

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 layer-wise aggregations need not be permutation-invariant. Usual choices of $\mathrm{R'}_v$ 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:

$$\boldsymbol{o}_{[v]} = g(\boldsymbol{h}_{[v]}^{(*)}),\ \forall v \in \mathcal{V}_{\mathtt{g}},$$

where $\boldsymbol{o}_{[v]} \in \mathbb{R}^y$ and g can be any arbitrarily complex neural network as usual. A visual example of the process for a single node is shown in Figure 3.11. In graph

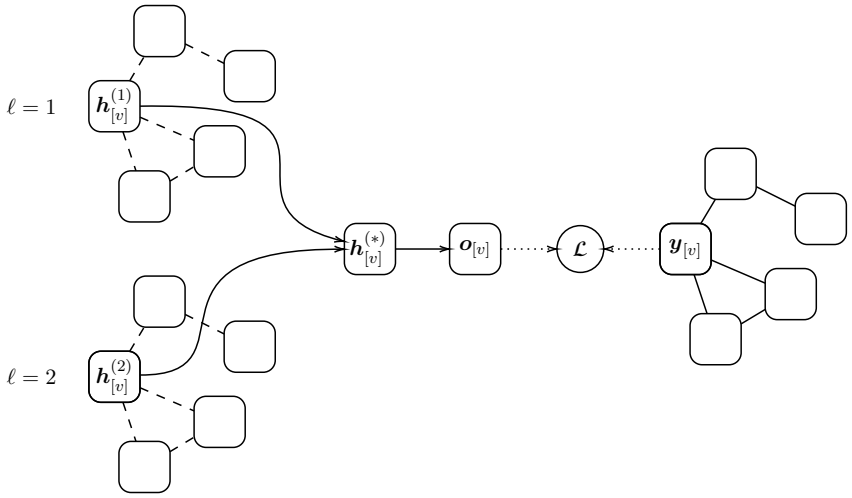


FIGURE 3.11: The role of hidden state readout function in a node classification/regression task. Here, we focus on a single node $\boldsymbol{h}_{[v]}^{(\ell)}$, where we drop the subscripts to avoid visual cluttering. Grey nodes replace the $[v]$ subscript. The hidden state readout function creates a node representation $\boldsymbol{h}_{[v]}^{(*)}$ 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 $\boldsymbol{y}_{[v]}$ in the isomorphic target graph. This operation is repeated for every node in the 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**) $\boldsymbol{h}_{\mathtt{g}}$, *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 an attributed graph $\mathtt{G}$ as follows:

$$\boldsymbol{h}_{\mathtt{g}} = \mathrm{R}_{\mathtt{g}}\left(\left\{\boldsymbol{h}_{[v]}^{(*)} \mid v \in \mathcal{V}_{\mathtt{g}}\right\}\right).$$

Typical readouts for DGNs include simple functions such sum, mean, max, or more complex aggregators such as deep sets models [Zah+17]. Finally, the graph embedding is fed to an output layer or a downstream network to compute the associated output:

$$\boldsymbol{o} = g(\boldsymbol{h}_{\mathtt{g}}).$$

A graph readout applied to an example graph is shown in Figure 3.12.

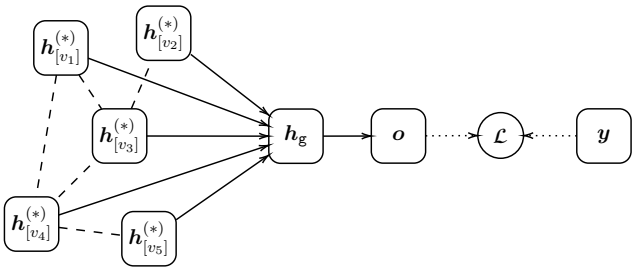


FIGURE 3.12: A graph readout on an example graph for a graph classification/regression task. Here, we assume the node representations $\boldsymbol{h}^{(*)}_{[v]}$ have already been obtained by a node readout (not shown).

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 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 3.13, 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 by Ying et al. [Yin+18]. Given an attributed graph $\mathtt{G}$ with n nodes, and assuming the $\ell - 1$ GCLs have been applied, DiffPool computes two matrices:

$$\begin{aligned}\boldsymbol{Z}^{(\ell-1)} &= \mathrm{S}_e(\boldsymbol{A}^{(\ell-1)}, \boldsymbol{H}^{(\ell-1)}) \in \mathbb{R}^{n\times h}\\ \boldsymbol{S}^{(\ell-1)} &= \mathrm{softmax}\left(\mathrm{S}_p(\boldsymbol{A}^{(\ell-1)}, \boldsymbol{H}^{(\ell-1)})\right) \in \mathbb{R}^{n\times k},\end{aligned}$$

where S_e and S_p are stacks of graph convolutional layers. The matrix $\boldsymbol{S}$ learns 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:

$$\begin{aligned}\boldsymbol{H}^{(\ell)} &= \boldsymbol{S}^{(\ell-1)}\boldsymbol{Z}^{(\ell-1)} \in \mathbb{R}^{k\times h}\\ \boldsymbol{A}^{(\ell)} &= \boldsymbol{S}^{(\ell-1)}\boldsymbol{A}^{(\ell-1)}\boldsymbol{S}^{(\ell-1)} \in \mathbb{R}^{k\times k}\end{aligned}$$

where $\boldsymbol{A}^{(0)} = \boldsymbol{A}_{\mathtt{g}}$ and $\boldsymbol{H}^{(0)} = \boldsymbol{X}_{\mathtt{g}}$. Thus, after applying the DiffPool layer, the size of the graph is reduced progressively from n to k nodes.

Topological pooling, on the other hand, uses not-differentiable methods which leverage the global structure of the graph, and the communities beneath it. These

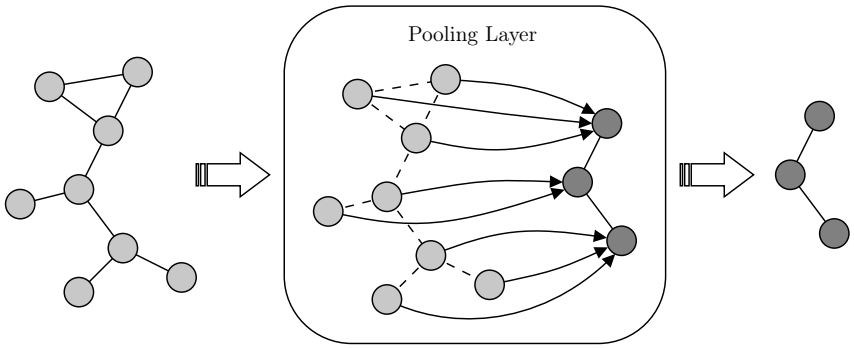


FIGURE 3.13: A visual example of a graph pooling layer.

methods work by grouping nodes according to well known graph theory tools, such as spectral clustering (see *e.g.* Von Luxburg [Von07] and Dhillon et al. [DGK07]).

3.5.3 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 kind of structures the network should preferably learn. The regularized objective function for supervised tasks[3] has the form:

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

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 $\boldsymbol{H}_{\mathsf{g}}^{(\ell)} \in \mathbb{R}^{n \times h}$. An example of regularization widely employed in practical settings is the *link prediction* unsupervised loss, defined as:

$$\Psi(\boldsymbol{H}^{(\ell)}) = \sum_{u,v} \left\| \boldsymbol{h}_{[v]}^{(\ell)} - \boldsymbol{h}_{[u]}^{(\ell)} \right\|_2, \tag{3.6}$$

where the summation ranges over 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.

3.6 Deep Generative Learning of Graphs

In this section, we discuss the graph generation problem. The term "graph generation" is purposely kept broad, to include a variety of methodologies which learn processes that generate graphs from a set of training examples. This research field originates

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

from generative models of graphs for the theoretical understanding of graph properties, which have been studied thoroughly since the '50s in the field of graph theory. The first known generative model is the Erdös-Rényi (ER) model [ER59]. The ER model studies random graphs, where the connectivity among graph nodes is modeled independently. This model is useful to study theoretical properties of graphs, such as how the global connectivity of the graph evolves as its size grows. Another historical model of graphs is the Watts-Strogatz (WS) model [WS98]. The WS model concerns "small world" graphs, *i.e.* graphs where it is possible to reach any other node in the graph with very short paths, regardless of their size. This property arises in several real-world graphs such as social and electrical networks. Lastly, the Barabási-Albert (BA) model [BA99], which models the so-called "preferential attachment" property, where the connectivity potential of a node is directly proportional to its number of neighbors.

While being useful to the study and understanding of graph properties, these models usually fail to generalize to real-world graph distributions, because they can only model one or a limited number of graph properties; moreover, their parameters cannot be learned from data in general. Other methods build upon the ideas of these seminal ideas, proposing frameworks to approximate graph distributions and their parameters with statistical and probabilistic tools. Relevant works in these field are stochastic block models [Air+08], exponential random graphs [Rob+07], and Kronecker graphs [Les+10]. All these methods can approximate more complex graph distributions such as graphs with communities, but are still limited to specific kinds of graphs and scale poorly to large datasets.

Another fruitful line of research as regards graph generation is related of graph grammars. A graph grammar is a set of production rules that allow an input graph to be rewritten into a new one. The transformation is performed by searching the input graph for a certain pattern, which is substituted with a pattern specified by the production rule. Virtually, any graph can be transformed in any other graph by an expressive grammar. Learning in the context of graph grammars can be cast as the problem of learning to sample production rules from an unknown probability distribution. However, sampling a graph using a graph grammar is computationally expensive, as every application of a production rule requires to solve a graph isomorphism problem. Nonetheless, many relevant works concerning graph generation with grammars have been proposed; we present two of the most relevant for completeness. In [Agu+16], the authors present an approach based on so-called hyper-graph grammars, which work by transforming a graph into a clique tree. The approach is capable of automatically extracting the building blocks of a larger graph, and to use them both for reconstruction and to generalize to similar graphs. The work in [Cos17] proposes a different graph grammar, called locally substitutable, which can be learned adaptively from data. The grammar is based on substituting "core" subgraphs (rooted neighborhood subgraphs) with other similar core subgraphs, while maintaining the same connectivity with the 2-hop edge frontier. Novel graphs can be generated using

a Metropolis-Hastings sampler, whose proposal distribution is learned with by a One-Class SVM [Sch+01], adapted to work with graphs with a Neighborhood Subgraph Pairwise Distance Kernel [CG10].

While effective in a relevant number of tasks, all the above approaches suffer when applied to large-scale learning from real-world dataset. To generate graphs for practical applications, a more flexible and efficient class of learning models is needed. Following, we review a variety of generative models of graphs based on Deep Learning approaches. The advantages of such models are the possibility to approximate complex distributions efficiently and effectively, and the flexibility to use different generative paradigms to adapt to the task at hand.

3.6.1 The Challenges of Graph Generation

The problem of generating graph structures from arbitrary distributions is arguably harder than predictive modeling. Some of the challenges that need to be addressed when designing deep generative models of graphs are related to the complexity of graph spaces. In particular, we report the following:

- *size of graphs spaces.* Graphs spaces are combinatorial, and thus very large. For example, the size of the space of undirected graphs with m nodes is $\binom{m}{2} = \frac{m(m-1)}{2}$, which becomes very large even for graphs with a moderate number of nodes. Thus, trivial approaches such as exhaustive enumeration are generally intractable. Moreover, real-world graph distributions are usually defined over attributed graphs with variable size, which makes the search space even larger;

- *discreteness of graphs spaces.* Graphs are discrete objects. This contrasts with the nature of neural networks, which can back-propagate only through continuous and differentiable objects. Hence, the learning process must be adjusted to work with discrete structures;

- *sparsity of graphs spaces.* For most generative tasks, only very small subsets of graph space contain graphs with non-zero probability. Thus, any generative method should be designed to focus on regions where interesting graphs are contained, to avoid efficiency issues;

- *complex dependencies.* Most real-world graph have hard structural constraints that are very difficult to enforce on a generative model (which is inherently stochastic). For example, cycle graphs are such that if only one edge is misplaced, the graph is not a cycle anymore;

- *non-unique representations.* In general, graphs are invariant under node permutation. Thus, the same graph can be potentially represented by up to $n!$ possible adjacency matrices, depending on the node permutation. This poses constraints on the kind of graph representations a generative model can learn. For example, generative methods that map graphs into a latent space must take

into account that different node permutations of the same graph must map to the same latent vector. While invariance to node permutation can be addressed by assuming an order of the graph nodes, this introduces the problem of maintaining order consistency among different graphs.

3.6.2 Generative Tasks

The family of DGMs of graphs is flexible enough to model different kinds of generative tasks. Assuming unattributed graphs for simplicity, we loosely follow the taxonomy proposed by Guo et al. [GZ20] and distinguish two main tasks related to graph generation:

- *unconditional generation*, where the task is to explicitly learn a distribution $p(\mathbf{g})$ over graphs, or some parameterized function that produces samples from it. Here, the term "unconditional" refers to the fact that the generation starts with drawing a vector $\boldsymbol{z} \in \mathbb{R}^z$ from some easy to sample prior distribution $p(\boldsymbol{z})$, which is usually assumed to be an isotropic Gaussian or a uniform distribution;

- *conditional generation*, where the aim is to learn a conditional distribution $p(\mathbf{g} \mid \boldsymbol{y})$ with $\boldsymbol{y} \in \mathbb{R}^y$, or the corresponding parameterized sampling mechanism. The purpose of the conditioning vector is to drive the generative process towards producing a graph with desired characteristics. For example, one might want to generate a graph whose structure resembles that of a graph given as input to the DGM. In this case, $\boldsymbol{y}$ is the representation of the conditioning graph.

Hereafter, we consider the task of unconditional generation, where we assume access to a dataset of graphs $\mathbb{D} = \{\mathbf{g}_i\}_{i=1}^n$. Broadly speaking, defining a DGM of graphs requires specifying two components: a *graph decoder*, which takes care of generating a graph, and an end-to-end generative framework used to optimize the model parameters. As regards the latter, common generative frameworks include VAEs, GANs, and flow-based models [RM15]. Here, we only describe approaches based on the first two.

3.6.3 Graph Decoders

The graph decoder is the architectural component that outputs some conditional distribution, which can be sampled to generate a graph. If the framework in which the decoder is placed allows for inference (such as the VAE), the conditional can be learned with MLE; otherwise, (such as with GANs) it is used only for sampling, and its parameters are optimized with adversarial training. Ideally, graph decoders should generate permutation invariant graphs; however, this is rarely the case. The major hurdle to devise permutation invariant graph decoders is their computational cost; even though some methods do exist [Niu+20], they are still too inefficient to be deployed in real world scenarios. Thus, in the following, we assume non-invariance. One particular caveat that needs to be addressed during the training phase of a graph decoder is maintaining the differentiability of the architecture while still generating

hard graph samples. This is critical especially in GAN-like architectures, where the discriminator must be trained with actual graphs. As with sequence generation with RNNs, the same techniques (straight-through gradient estimation [BLC13], reparameterization [KW14], or even reinforcement learning-based techniques [Wil92]) can be used for this purpose. There are two main paradigms to implement adaptive graph decoders, which we detail in the following.

One-shot Decoders

This class of graph decoders outputs a dense probabilistic adjacency matrix $\tilde{\boldsymbol{A}}_{\mathrm{g}} \in \mathbb{R}^{n \times n}$, where n is the maximum number of nodes allowed. The probabilistic matrix is sampled entry by entry to produce an actual adjacency matrix. The entries of the matrix are modeled as independent Bernoulli variables, which indicate the presence or absence of an edge. Generally, the elements in the adjacency matrix a_{ii} are modeled as independent Bernoulli variables that specify if a node belongs to the graph or not. In practice, the entries of the matrix are produced by a neural network with sigmoid outputs that predicts an $n \times n$ vector. Thus, the adjacency matrix can be sampled in parallel (hence the term "one-shot"). Two possible approaches to specify a one-shot decoder are:

- *graph-based* decoders, which require a graph representation $\boldsymbol{z}$. In this case, the decoder models the conditional as follows:

$$p(\tilde{\boldsymbol{A}}_{\mathrm{g}} \,|\, \boldsymbol{z}) \approx p(\tilde{\boldsymbol{A}}_{\mathrm{g}} \,|\, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}),$$

  where $p_{\boldsymbol{\theta}}$ is a neural network with sigmoid outputs that predicts the matrix from the graph representation;

- *node-based* decoders, which require a matrix of node representations $\boldsymbol{Z} \in \mathbb{R}^{n \times z}$. In this case, the decoder models the conditional as follows:

$$p(\tilde{\boldsymbol{A}}_{\mathrm{g}} \,|\, \boldsymbol{Z}) \approx \prod_{i=1}^{n} \prod_{j=1}^{n} p(a_{ij} \,|\, \boldsymbol{z}_i, \boldsymbol{z}_j) = \prod_{i=1}^{n} \prod_{j=1}^{n} p_{\boldsymbol{\theta}}(\boldsymbol{z}_i \boldsymbol{z}_j),$$

  where i and j range over the matrix rows, and $p_{\boldsymbol{\theta}}$ is a neural network that takes as input a pair of node representations, and applies a sigmoid function to their dot product. The idea is that nodes that are close in representation space should be more likely to be connected.

One-shot approaches are usually fast to train and to take samples from. However, they are too simplistic, in that they assume the edges are generated independently (which is usually not the case for real-world graphs). Furthermore, the maximum number of nodes must be pre-specified, which makes them unable to generalize to larger graphs.

Autoregressive Decoders

Autoregressive decoders assume that graphs are generated by some sequential process that involves its set of nodes. Specifically, the generative process is the following:

$$p(\mathbf{s}) = \int p(\mathbf{s}, \pi)\, d\pi = \int p(\mathbf{s} \mid \pi)\, q(\pi)\, d\pi,$$

where $\mathbf{s}$ are sequences that generate graphs one component at a time, and the order of generation is given directly or indirectly by a node permutation drawn from a prior $q(\pi)$. The idea is to decompose the generating sequence autoregressively as follows:

$$p(\mathbf{s}) = \int p(\mathbf{s}_i \mid \mathbf{s}_{<i}, \pi)\, q(\pi)\, d\pi.$$

However, this requires to integrate over all $n!$ possible node permutations, which becomes intractable for moderately large graphs. An approximate solution to this issue is to assume some ordering and create the sequences before training, as a preprocessing step. Once the sequence are fixed, the chain-rule decomposition becomes tractable:

$$p(\mathbf{s}) \approx p^{\pi}_{\boldsymbol{\theta}}(\mathbf{s}) = p^{\pi}_{\boldsymbol{\theta}}(\mathbf{s}_i \mid \mathbf{s}_{<i}).$$

Depending on the nature of the graph generating sequence, we distinguish four possible approaches to develop autoregressive graph decoders:

- *node-based* approaches decompose the graph as a sequence of actions performed on an initially empty graph. These action correspond to decisions such as whether to add a node to the existing graph, and which nodes it must be connected to. In this case, $\mathbf{s}_{<i}$ is a vector that represents the current state of the graph. For all these models, one has two options as to how to implement the autoregressive network. One approach is to use a hierarchy of RNNs: one keeps track of the state of the nodes added to the graph, and the other is responsible to connect newly added nodes to the current graph, given the state of the first [You+18b]. The other choice is to update the state of the current graph with a DGN. This state is passed to the networks responsible for deciding which action to perform [Li+18];

- *edge-based* approaches decompose the graph as a sequence of edges. To produce an ordered sequence of edges, one must first order its nodes, then label the nodes with progressive integers, and then sort its set of edges in lexicographic order. In this case, $\mathbf{s}_{<i}$ represents the state of the graph indirectly, by keeping memory of the edges of the sequence already generated. These approaches are mostly implemented with RNNs [GJR20; BMP19a];

- *rule-based* approaches can be applied in cases where the graph generation can be decomposed in a sequence of production rules over some known grammar

(*e.g.* for molecules or computer programs) [KPH17; DTD+18]. In this case, the model generates a sequence of production rules to construct a desired graph;

- *motif-based* approaches decompose the graph as a sequence (or even a tree) of *motifs*, *i.e.* very small and manageable subgraphs, which are combined adaptively. Here, the challenge is mainly how to decompose the graphs into sequences of motifs.

A special kind of sequential decomposition for graphs is the Simplified Molecular Input Line Entry System (SMILES) linearization applied to molecules. We shall define the SMILES more precisely in Section 7.1.1; for the moment, it is sufficient to say that the SMILES encoding of a molecule is a string of ASCII characters that represents its structure. When a domain-specific linearization techniques such as SMILES is not available, the sequences representing the graph generative process are constructed based on some node ordering strategy. One general strategy to do so is to choose one node at random, then visit the graph nodes with a depth-first or breadth-first traversal. The order by which the nodes are visited is used to determine the order of the elements in the sequences. Clearly, this approach is not optimal since it heavily depends on the starting node, and may produce very different sequences for different starting node choices. However, it has been shown to work empirically [You+18b; Li+18; BMP19a; GJR20]. The pros and cons of autoregressive decoders are orthogonal to those of one-shot decoders: briefly, they allow generating variable-sized graphs seamlessly, and they can model dependencies between nodes and edges by means of the autoregressive property. However, both training and sampling processes are slower in terms of computational time, because the graphs are reconstructed one sequence element at a time and not in parallel.

3.6.4 Performance Evaluation

The desired end result of training a generative model is that the structure of samples generated by the network should resemble that of graphs in the training set, without being identical. This is more challenging with respect to predictive tasks, since the samples of a generative model do not exist until they are generated. Hence, there are no strong guarantees that the learned distribution is the correct one. Furthermore, the model can very easily obtain misleading performances by learning to replicate training graphs exactly, or repeating the same graph typology over and over. Thus, one critical aspect of using generative models of graphs is how to assess performances. Below, we define two broad classes of metrics that allow the evaluation of graph models, assuming the availability of a training sample $\mathbb{T} \subseteq \mathbb{D} = \{\mathtt{g}\}_{i=1}^{n}$, and a collection[4] $\mathbb{G} = \{\mathtt{g}'\}_{i=1}^{m}$ of samples generated by the model.

[4]Here, we use the term "collection" to indicate a multiset, meaning that it can possibly contain duplicate elements.

Quantitative Metrics

Quantitative metrics measure the rate at which the generative model produces diverse and heterogeneous graphs, without taking into account structural similarity. The three main quantitative metrics considered in the literature are:

- *novelty* measures the ratio of generated samples that are not training samples. A high novelty indicates that the model has not learned to replicate training graphs. Conversely, low novelty indicates that the model has overfit the training data. Formally, it is measured as $1 - \frac{|\mathbb{G} \bigcap \mathbb{T}|}{|\mathbb{T}|}$;
- *uniqueness* measures the ratio of unique graphs with respect to the total number of graphs generated. A low uniqueness rate might indicate that the model has overfit one specific graph structure. To calculate uniqueness, one first checks every graph for isomorphism with every other graph in the generated sample, removing them. If the resulting set of unique graphs is indicated by $\mathbb{U}$, uniqueness is calculated as $\frac{|\mathbb{U}|}{|\mathbb{G}|}$.
- *validity* measures the ratio of generated graphs that respect some validity constraint, out of the total number of graphs generated. To calculate validity, one needs to check whether every generated graph satisfies some structural constraint or not (*e.g.* the presence of a cycle). If $\mathbb{V}$ is the collection of graphs that satisfy the structural constraints, validity is calculated as $\frac{|\mathbb{V}|}{|\mathbb{G}|}$. This metric is particularly useful in molecular generation tasks, since chemically invalid molecules are useless. When assessing validity is required, novelty and uniqueness are usually conditioned on validity first, meaning that the all the invalid graphs are removed from $\mathbb{G}$ before calculating these two metrics.

Qualitative Metrics

Quantitative metrics give only one side of the spectrum relatively to how a generative model is performing. For example, assessing novelty alone might be misleading, since a high novelty rate can also be associated to underfitting (meaning that the model generates graphs very different from the training sample, which are trivially novel). Thus, a proper evaluation of generative models must also include a series of metrics that consider the structural properties of the generated graphs. We call such metrics *qualitative*. The framework under which qualitative metrics are assessed consists of comparing the empirical distribution of a certain graph property in the training sample, to the empirical distribution on the generated sample. Given a generic graph with d nodes, coming from one of the two samples indifferently, a relevant subset of such properties includes:

- node degree distribution, that is, a d-dimensional vector where each position contains the degree of the corresponding node. Notice the length of the vector may differ across different graphs, since their number of nodes may change;

- clustering coefficient distribution. The clustering coefficient of a node v is defined as the ratio between the number of actual connections between neighbors of v out of the total number of possible connections. In other words, it is a relative measure of how many "closed triangles" (fully connected graphs with three nodes) the node is part of. Similarly to the node degree distribution, it consists of a d-dimensional vector where each position contains the clustering coefficient of the corresponding node;
- number of nodes of the graph, which is a single integer;
- number of edges of the graph, which is again a single integer;
- average orbit counts. Orbits are subgraphs with fixed size and shape, usually small. Counting orbits in a graph can be viewed as a generalization of the clustering coefficient to more complex substructures than triangles. In practice, it consists in a dk-dimensional vector, where k is the number of orbits considered;
- Neighbourhood Subgraph Pairwise Distance Kernel (NSPDK) [CG10], which measures the similarity between two graphs by counting the number of matching induced subgraphs between them. The subgraphs are derived node-wise, by considering neighborhoods of a node comprising nodes at increasing path lengths. Differently from the other qualitative metrics, the NSPDK provides a global measure of similarity between graphs, since it is based on multiple subgraph matching. In practice, it is a vector of length $|\mathbb{G}|-1$ ($|\mathbb{T}|-1$, respectively), where each position measures the similarity of the graph with another graph in the sample.

Once the graph properties are calculated for each graph in the sample, there are two options to measure the distance between the empirical distribution of the training sample versus the generated sample, based on their respective number of elements:

- if $n = m$, one can compute their empirical KLD as follows:

$$\overline{\mathrm{KLD}}(\mathbb{G} \parallel \mathbb{T}) = \frac{1}{n}\sum_{i=1}^{n} \mathrm{prop}(\mathbf{g}_{(i)}) \log\left(\frac{\mathrm{prop}(\mathbf{g'}_{(i)})}{\mathrm{prop}(\mathbf{g}_{(i)})}\right),$$

  where prop is one of the properties mentioned above;

- if $n \neq m$, one can either concatenate all the values of the property for each graph in the sample, and then fit a histogram with an equal number of bins to make their length match in order to apply the empirical KLD. Another, more general, approach to compare distribution when the two samples have different lengths is to compute their Maximum Mean Discrepancy (MMD) [Gre+12]. Intuitively, the MMD measures the distance between two distributions as the sum of the distances between their matching moments. The computation of these distances can be generalized to an infinite space of moments by applying a kernel trick [HSS08].

Part II

The Evaluation of Deep Graph Networks and Applications to Computational Biology

Chapter 4

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 need to undergo 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 out by researchers [Shc+18], the papers that introduce novel architectures often adopt not reproducible or unfair experimental setups, which make the comparisons among models unreliable. Moreover, there is a tendency of attributing improved performances to subtle architectural changes, when in reality the improvement is a consequence of a more extensive model selection [LS18]. In this section, we present three contributions related to address these important issues in the context of DGNs. Specifically, we:

- provide a rigorous evaluation of existing DGNs models in the context of graph classification, using a standardized and reproducible experimental environment. Specifically, we perform numerous 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.

4.1 Datasets

All the chosen graph datasets are publicly available [Ker+16] and represent a relevant subset of those most frequently used benchmarks in the literature to compare DGNs. Some collect molecular graphs, while others contain social graphs. Specifically, we use the following chemical datasets:

- D&D [DD03] is a graph dataset in which nodes are amino acids, and there is an edge between two nodes if 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 [Bor+05] is a subset of D&D where the largest graphs have been removed;
- NCI1 [WWK08] 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 taken from the BRENDA enzyme database [Sch+04]. In this case, the task is to correctly assign each enzyme to one out of 6 Enzyme Commission (EC) numbers.

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

- IMDB-BINARY and IMDB-MULTI [YV15] 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 [YV15] 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 [YV15] is a dataset where each graph is an ego-network of different researchers from some research field. There is an edge between two authors if they coauthored a scientific article. 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 assign the same feature or the degree to all the 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	-

4.2 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 graph readout layer and a final MLP classifier, which maps the graph representation to the dataset-dependent output. Following, we describe in detail the kinds of GCL used by the DGNs.

Graph Isomorphism Network The Graph Isomorphism Network (GIN) convolutional layer by Xu et al. [Xu+19] is implemented as follows:

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

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.

GraphSAGE The Graph SAmple and aggreGatE (GraphSAGE) layer by [HYL17] is implemented as follows:

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

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 3.5.2, where S_e and S_p are a stack of 3 GraphSAGE layers respectively. Thus, one DiffPool layer requires computing 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 by [SK17] is implemented as follows:

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

where MLP is a neural network that computes a weight between the hidden state of the current node $\boldsymbol{h}_{[v]}^{(\ell-1)}$ and its generic neighbor $\boldsymbol{h}_{[u]}^{(\ell-1)}$, 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 comprising a stack of 3 ECC layers.

DGCNN The graph convolutional layer of the Deep Graph Convolutional Neural Network (DGCNN) [Zha+18], is the following:

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

where $\boldsymbol{D}^{-1}$ 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 Baselines

We compare the proposed architectures with two structure-agnostic baselines, one for molecular graphs, and one for social graphs. For molecular graphs, except for graphs in the ENZYMES dataset, we use a MLP-based model [Ral+05], applied as follows to a generic graph `G`:

$$\boldsymbol{h}_{\mathtt{g}} = \mathrm{MLP}\left(\sum_{v\in\mathcal{V}_{\mathtt{g}}} \boldsymbol{x}_{[v]}\right).$$

This architecture corresponds to summing the node features together (which are one-hot encoded vectors representing the atom types), and applying a 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 attributed graph $\mathtt{G}$ as follows:

$$\boldsymbol{h}_{\mathtt{g}} = \mathrm{MLP}_{\mathtt{g}} \left(\sum_{v \in \mathcal{V}_{\mathtt{g}}} \mathrm{MLP}_{v}(\boldsymbol{x}_{[v]}) \right),$$

where MLP_v is a linear layer plus a ReLU non-linearity, and $\mathrm{MLP}_{\mathtt{g}}$ is a two hidden layer MLP with ReLU non-linearities [Zah+17]. 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.

4.4 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. All models are trained with the Adam [KB15] 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. For the baselines, 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 coefficient. To be consistent with the literature, we implement early stopping with patience, where training stops if a number of epochs have passed without improvement on the validation set. A high patience 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.

4.5 Results

The experimental results are reported in Table 4.2 (for the chemical datasets) and Table 4.4 (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 ($\approx 100\%$ 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.

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)

To rigorously assess whether the best model effectively outperforms the competitors, we use the method proposed in [Ben+17]. Specifically, the method computes the posterior probability of the differences in performance between the two models on a given dataset. The method then draws samples from this posterior, and estimates how many times one model outperforms the other, or how many times they perform equivalently (e.g. if their difference in performance falls in a predefined region around the posterior mean, called rope). For our use case, we set the rope to the standard deviation of the best model in a particular dataset. Table 4.3 presents the results.

Table 4.3: We compare the best scoring models to all the others on each dataset. Tale entries represent the probability that the best model performs better than the competitor, while the width of the region of equivalence is reported in brackets. Bold entries highlight cases where the two models are considered equally performant by the Bayesian test.

Dataset	Best Model	Baseline	DGCNN	DiffPool	ECC	GIN	GraphSAGE
DD	**Baseline**	-	**0.05 (0.95)**	**0.27 (0.73)**	0.72 (0.28)	**0.17 (0.83)**	0.74 (0.26)
NCI1	**GIN**	1.00 (0.00)	0.96 (0.04)	0.91 (0.09)	0.98 (0.02)	-	0.99 (0.01)
PROTEINS	**Baseline**	-	**0.26 (0.74)**	**0.17 (0.83)**	**0.4 (0.56)**	**0.25 (0.75)**	**0.35 (0.64)**
ENZYMES	**Baseline**	-	1.00 (0.00)	**0.38 (0.62)**	1.00 (0.00)	**0.32 (0.68)**	**0.58 (0.42)**
IMDB-B	**GraphSAGE**	1.00 (0.00)	1.00 (0.00)	**0.07 (0.93)**	**0.16 (0.683)**	**0.02 (0.95)**	-
IMDB-M	**GraphSAGE**	1.00 (0.00)	0.99 (0.01)	**0.16 (0.84)**	**0.25 (0.75)**	**0.00 (0.90)**	-
REDDIT-B	**GIN**	0.97 (0.03)	0.98 (0.02)	1.00 (0.00)	1.00 (0.00)	-	**0.00 (0.99)**
REDDIT-5K	**GIN**	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	-	**0.00 (0.97)**
COLLAB	**GIN**	1.00 (0.00)	1.00 (0.00)	0.81 (0.00)	1.00 (0.00)	-	**0.00 (0.99)**
IMBD-B	**GIN + Degree**	**0.17 (0.73)**	**0.20 (0.79)**	**0.29 (0.71)**	**0.46 (0.54)**	-	**0.01 (0.70)**
IMDB-M	**Baseline + Degree**	-	0.50 (0.49)	**0.49 (0.51)**	0.78 (0.21)	**0.22 (0.74)**	**0.20 (0.77)**
REDDIT-B	**GIN + Degree**	**0.20 (0.58)**	0.00 (0.01)	0.00 (0.00)	1.00 (0.00)	-	1.00 (0.00)
REDDIT-5K	**GIN + Degree**	0.00 (0.03)	0.04 (0.35)	0.33 (0.18)	1.00 (0.00)	-	0.74 (0.24)
COLLAB	**GIN + Degree**	**0.07 (0.92)**	**0.01 (0.96)**	**0.41 (0.59)**	1.00 (0.00)	-	0.79 (0.21)

As can be seen, in many cases the best model performs equivalently to the competitors, especially in the DD, PROTEINS, and ENZYMES chemical datasets, and for social datasets whenever the model leverage the degree feature. This provides evidence that small fluctuation in the performance accuracy must not be interpreted as a performance boost provided by the specific architecture, but can probably be attributed to statistical or dataset noise. Following, we summarize other relevant findings of the study.

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 yet able to fully exploit the structure on such datasets. This contrasts with the current literature of chemistry, where structural properties of the molecular graph are strongly believed to correlate with molecular properties [Ros63]. 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 caused by other factors, such as random initialization, 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 4.4: 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)	**87.0**(4.4)	**53.8**(5.9)	**75.9**(1.9)
	GraphSAGE	**69.9**(4.6)	**47.2**(3.6)	86.1(2.0)	49.9(1.7)	71.6(1.5)
WITH DEGREE	Baseline	70.8(5.0)	**49.1**(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	**71.2**(3.9)	48.5(3.3)	**89.9**(1.9)	**56.1**(1.7)	**75.6**(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 DGNs 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 4.5. Given the benefit of adding of the node degree as a feature, we hypothesize that models such as GIN 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, even when the node degree is not provided explicitly.

TABLE 4.5: 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 4.1 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[1].

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

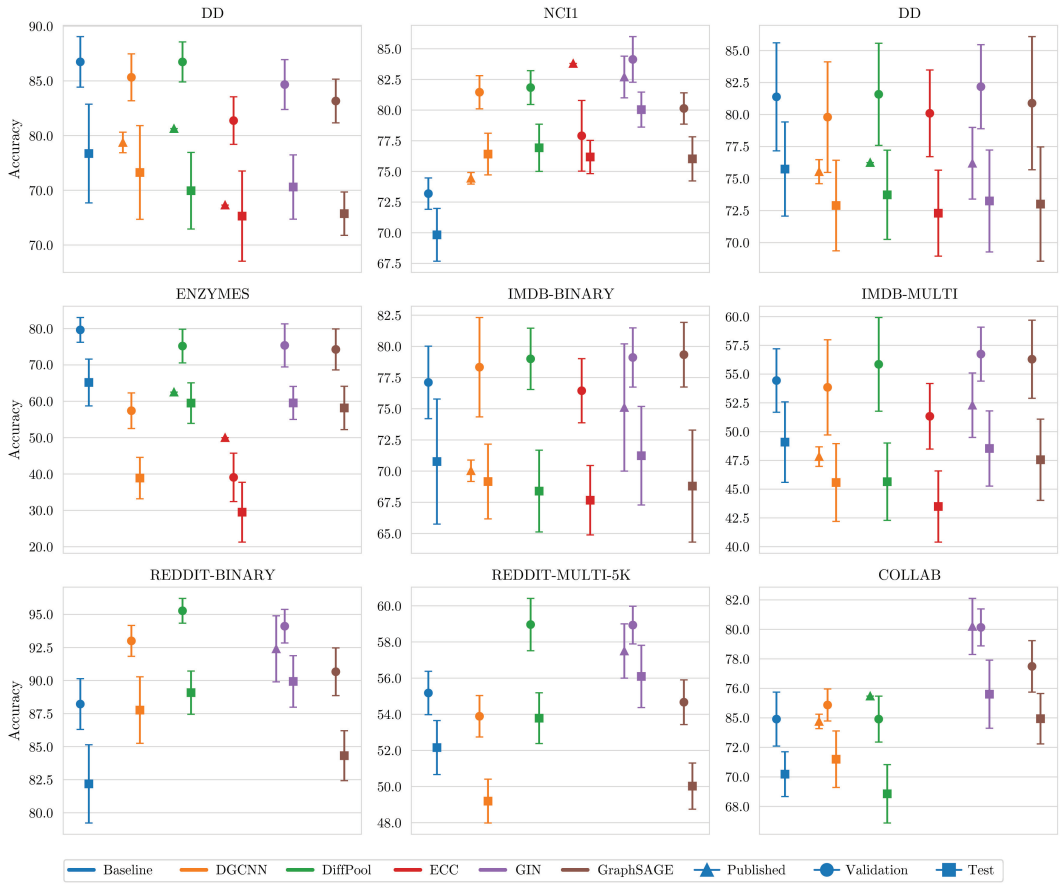
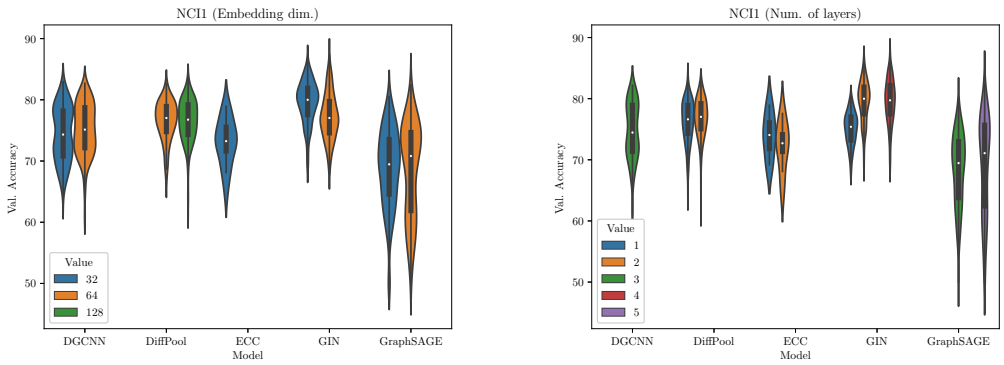


FIGURE 4.1: Results.

4.5.1 Insights from Hyper-Parameter Analysis

As a follow-up study, we analyze how sensitive the DGNs are to changes in the hyper-parameters. In other words, we study how drastically, and if at all, performances vary when varying some relevant hyper-parameters. This would give insights on several practical matters such as which network one can train in the initial phases of an experimental study, in order to get a sensible baseline to build from. To do so, we use validation accuracy as a proxy for test accuracy. First, we collect *post-hoc* all the validation scores obtained by the different models; then, we group the model performances obtained with a fixed choice of a hyper-parameter. We focus on two hyper-parameters that are of relevance for DGNs: number of GCLs, and dimension of the representation obtained by the network. Figure 4.2a shows a violin plot displaying how the validation performance changes while fixing the number of GCLs and optimizing the rest of the hyper-parameters. It can be observed how the performances of GIN are relatively stable with respect to the change in number of layers, while obtaining the highest performances. The figure shows that GIN obtains better performances when the number of GCLs is higher than one. The other model performances are fairly stable to the change in number of layers, but we notice an extreme variability in performances (such as with GraphSAGE) which indicates that proper hyper-parameter tuning is essential to get good performances. Figure 4.2a displays

the change in performance while fixing the embedding dimension. From the figure, it appears that GIN performs better with a smaller embedding dimension. This might be partially explained by the theoretical power of GIN, which however makes it prone to overfitting.



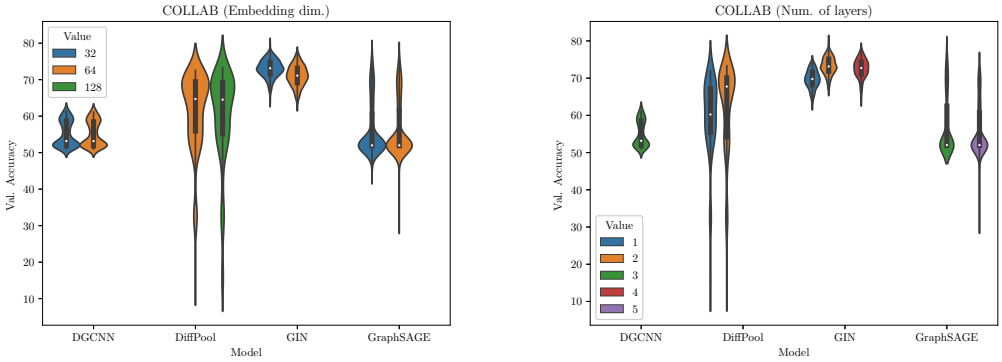
(A) Embedding dimension. (B) Number of layers.

FIGURE 4.2: The change in validation performances observed on the NCI1 dataset, while optimizing with a fixed number of layers (A) or fixed graph embedding dimension.

In Figure 4.3a and Figure 4.3a, we repeat a similar analysis on the COLLAB dataset without using degree information. We notice the same trend with GIN, which requires at least two GCLs to obtain good performances. However, there seems not to be a relevant change in performance when using four instead of two GCLs. On the contrary, increasing the number of DiffPool layers from one to two has the effect of boosting the accuracy by approximately 10 percentage points on average. However, we also notice from the elongation of the violin that DiffPool is extremely challenging to optimize on this dataset. In Figure 4.3b, we show the same analysis but focusing on the embedding dimension. Once again, the trend of Figure 4.2b appears from GIN, where a larger embedding dimension results in a decreased performance. As regards the other models, it seems that there is no effect obtained by varying the embedding dimension, as the violin for the different choices look specular.

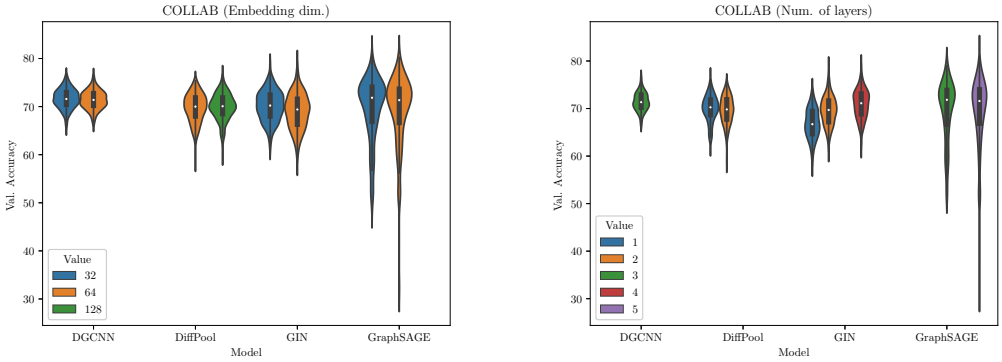
Figure 4.4a and Figure 4.4b show the same results when the degree features is added to the training graphs. In particular, in Figure 4.4a we see that adding the degree might allow GIN to use a higher number of layers to increase performance. For the other models, it appears like changing the number of layers does not provide and substantial change in performance. Finally, the plot of Figure 4.4b looks very similar to that of Figure 4.3b, but the effect of the embedding dimension for GIN is less noticeable.

These results show once again how it is easy to reason about the behavior of DGNs once the experimental setup is transparent and the models are evaluated fairly.



(A) Embedding dimension. (B) Number of layers.

FIGURE 4.3: The change in validation performances observed on the COLLAB dataset, while optimizing with a fixed number of layers (A) or fixed graph embedding dimension.



(A) Embedding dimension. (B) Number of layers.

FIGURE 4.4: The change in validation performances observed on the COLLAB dataset (using the degree feature), while optimizing with a fixed number of layers (A) or fixed graph embedding dimension.

Chapter 5

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.

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 [Gil77]. The use of these modelling tools allows investigating 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 amount 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 graphical notations of pathways exist in the literature (see, e.g., Karp et al. [KP94], Reddy et al. [RML+93], and Le Novere et al. [Le +09]),

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 DGNs 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 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* [Kit04] which is to some extent correlated with the notion of global sensitivity [Zi11]. 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 [Li+10], 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

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

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 Section 4.4. 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 than shallow ones 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:

$$c_1\mathsf{S}_1 + c_2\mathsf{S}_2 + \ldots \xrightarrow{k} c'_1\mathsf{P}_1 + c'_2\mathsf{P}_2 \ldots,$$

where $\mathsf{S}_i, \mathsf{P}_i$ 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 $k \in \mathbb{R}_+$ are *kinetic constants* 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

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

(A) Reactions

$$\begin{aligned}
\frac{d\mathsf{A}}{dt} &= -k_1\mathsf{A}B + k_2\mathsf{B} \\
\frac{d\mathsf{B}}{dt} &= k_1\mathsf{AB} - k_2\mathsf{B} \\
\frac{d\mathsf{C}}{dt} &= -k_3\mathsf{CDA} \\
\frac{d\mathsf{D}}{dt} &= -k_3\mathsf{CDA} \\
\frac{d\mathsf{E}}{dt} &= k_3\mathsf{CDA} - k_4\mathsf{E} + k_5\mathsf{F} \\
\frac{d\mathsf{F}}{dt} &= k_4\mathsf{E} - k_5\mathsf{F} \\
\frac{d\mathsf{G}}{dt} &= -\frac{k_6\mathsf{G}}{1+2\mathsf{F}} + k_7\mathsf{H} \\
\frac{d\mathsf{H}}{dt} &= \frac{k_6\mathsf{G}}{1+2\mathsf{F}} - k_7\mathsf{H}
\end{aligned}$$

(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.

language [Huc+18], 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 of 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 $\mathsf{r}i$ with $i = 1, \ldots, 7$. 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 [Pet77], 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 [GHL07] 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 labeled 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 as the one shown in Figure 5.1b. The state of a PPN (called *marking*) is 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 $\wp = \langle \mathcal{P}, \mathcal{T}, \mathcal{A}_S, \mathcal{A}_P, \mathcal{A}_I, \varpi, \varsigma, m_0 \rangle$ 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;
- $\varpi : \mathcal{A}_S \to \mathbb{N}_+$ is a function that assigns a non-negative multiplicity standard edges;
- $\varsigma : \mathcal{T} \to (\mathcal{M} \to \mathbb{R}_+)$, 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.

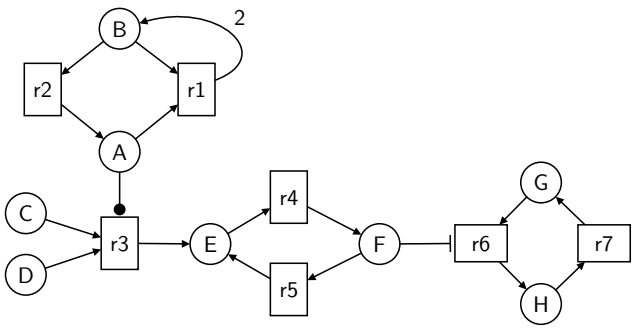


FIGURE 5.2: The Pathway Petri Net corresponding to the reactions described in Figure 5.1a.

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);
- the initial marking m_0.

Basically, for our purposes, a PPN is a tuple $\wp = \langle \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{A}}_S, \boldsymbol{\mathcal{A}}_P, \boldsymbol{\mathcal{A}}_I \rangle$ 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** $\mathrm{g} = \langle \mathcal{V}_\mathrm{g}, \mathcal{E}_\mathrm{g} \rangle$. To do so, we first define the following nodes and edges subsets:

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

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

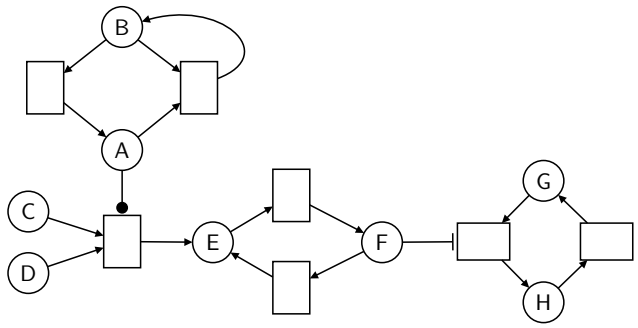


FIGURE 5.3: The pathway graph obtained by omitting kinetic formulas and edge labels from the one in Figure 5.2. Notice that the names of the molecules are displayed for visual aid, but are not included of the pathway graph.

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 [Kit04], a property observed in many biological systems. A general notion of robustness has been formalized by Kitano [Kit07]. 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 $\mathfrak{S}$, with respect to a specific functionality a and against a set of perturbations $\mathcal{U}$ is expressed as:

$$R^{\mathfrak{S}}_{a,\mathcal{U}} = \int_{\nu} p(\nu)\, V^{\mathfrak{S}}_{a}(\nu)\, d\nu$$

In the above definition, $p(\nu)$ is the probability a perturbation ν, and $V_a(\nu)$ evaluates the functionality a of the system $\mathfrak{S}$ under the perturbation ν. More precisely, function $V_a(\nu)$ gives the viability of a under perturbation ν, in relation to the viability of a under normal conditions. In the absence of perturbations, $V_a(\nu) = 1$, 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, $V_a(\nu) = 0$ if perturbations cause the system to fail completely in performing a, and $0 < V_a(p) < 1$ in the case of relevant perturbations.

An improvement to Kitano's formulation of robustness has been proposed by Rizk et al. [Riz+09], 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 [Lar+11].

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 by Kitano [Kit07] and Rizk et al. [Riz+09]. In particular, the *absolute concentration robustness* proposed by Shinar et al. [SF10] 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 by Nasti et al. [NGM18], 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 [Zi11], 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 performing exhaustive numerical simulations in parameter space [Riz+09; IL15] (here, we use the word *parameters* to refer to the kinetic parameters, or the initial concentrations). In some particular cases, one can exploit the biological network structure to avoid performing simulations altogether [SF10]. Moreover, in cases where the dynamics of the network are monotonic, the number of such simulations can be significantly reduced [GMN19].

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. In other words, given an input/output pair, this subgraph corresponds to the portion of the pathway that is relevant for the assessment of the robustness. The idea is to extract these subgraphs for each pathway graph and for each possible input/output pair of molecules. In order to do so, let us first introduce a helper data structure which we call **enriched pathway graph**. Given a pathway graph $\mathtt{g}$, its enriched version $\mathtt{g}+$ is defined as follows: initially, $\mathcal{V}_{\mathtt{g}+} = \mathcal{V}_{\mathtt{g}}$, $\mathcal{E}_{\mathtt{g}+} = \mathcal{E}_{\mathtt{g}}$. Then, for every standard edge $(u, v) \in \mathcal{E}^{\text{std}}_{\mathtt{g}}$ where $u \in \mathcal{V}^{\text{mol}}_{\mathtt{g}}$ and $v \in \mathcal{V}^{\text{rx}}_{\mathtt{g}}$, we update the standard edges of $\mathtt{g}+$ with a reverse standard edge (v, u), setting $\mathcal{E}^{\text{std}}_{\mathtt{g}+} = \mathcal{E}^{\text{std}}_{\mathtt{g}+} \bigcup \{(v, u)\}$. Note that standard edges from reactions to molecules, nor promotion and inhibition edges are not reversed. 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 [FS08]. In practice, the nodes defining the subgraphs are identified through $\mathtt{g}+$. More specifically, given $\mathtt{g}$, $\mathtt{g}+$, and a pair of nodes $v_I, v_O \in \mathcal{V}^{\text{mol}}_{\mathtt{g}}$, we define the subgraph induced by (v_I, v_O) in $\mathtt{g}$ as the graph $\mathtt{g}_{[\mathtt{p},(v_I,v_O)]} = \langle \mathcal{V}_{\mathtt{p}}, \mathcal{E}_{\mathtt{p}} \rangle$ where $\mathcal{V}_{\mathtt{p}} = \left\{ v \in \mathcal{V}_{\mathtt{g}+} \mid v_I \xrightarrow{\mathtt{g}+} v \wedge v \xrightarrow{\mathtt{g}+} v_O \right\}$ and $\mathcal{E}_{\mathtt{p}} = \left\{ (u, v) \in \mathcal{E}_{\mathtt{g}} \mid u, v \in \mathcal{V}_{\mathtt{p}} \right\}$. In practice, $\mathtt{p}$ is a subgraph of $\mathtt{g}$ where the node set contains v_I, v_O, as well as nodes in every possible oriented path from v_I to v_O in

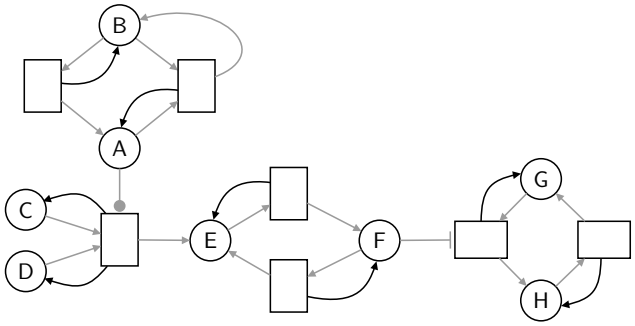
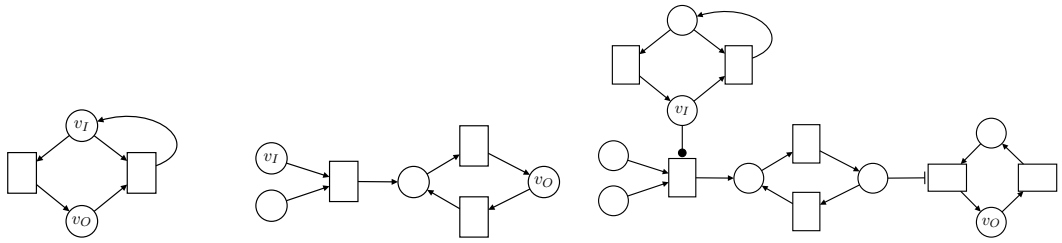


FIGURE 5.4: The enriched pathway graph obtained from the pathway graph of Figure 5.3. The edges added with respect to the original graph are shown in black.

g+. Notice that p 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. Notice how, in the subgraph of Figure 5.5b, the node D (below node $v_I = \mathsf{C}$) is included in the subgraph because, even though there is no oriented path between them in the pathway graph, there is one in the enriched graph (see Figure 5.4).



(A) $v_I = \mathsf{B}$, $v_O = \mathsf{A}$ (B) $v_I = \mathsf{C}$, $v_O = \mathsf{F}$ (C) $v_I = \mathsf{A}$, $v_O = \mathsf{H}$

FIGURE 5.5: Examples of subgraphs of the pathway graph in Figure 5.3 induced by different input/output node pairs (v_I, v_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 by Somogyi et al. [Som+15]) 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 $\mathsf{g}_{[\mathsf{p},(v_I,v_O)]}$, we computed the width α of the range of concentrations reached by the output molecules v_O by varying the

[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.

input v_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 $r_{(v_I,v_O)} \in [0,1]$ 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:

$$r_{(v_I,v_O)} = 1 - min(1, \frac{\overline{\alpha}}{0.4}).$$

5.3.3 Data Preprocessing

Our initial data collection consisted of 706 SBML models of biochemical pathways, downloaded from the BioModels database [Le +06]. 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], after removing empty models (not containing any node) and discarding duplicates. These models were first transformed into PPNs 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 $\mathcal{G} = \{\mathtt{g}_i\}_{i=1}^{484}$ 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:

$$\mathbb{D} = \bigcup_{\mathtt{g} \in \mathcal{G}} \left\{ (\mathtt{g}_{[\mathtt{p},(v_I,v_O)]}, \overline{r}_{(v_I,v_O)}) \mid \forall (v_I, v_O) \in \mathcal{V}_{\mathtt{g}}^{\text{mol}} \times \mathcal{V}_{\mathtt{g}}^{\text{mol}} \right\},$$

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

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. 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. Specifically, Given an induced subgraph $\mathtt{g}_{[\mathtt{p},(v_I,v_O)]}$ and one of its nodes $v \in \mathcal{V}_{\mathtt{p}}$, we

[3] May 2019.

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

define its 3-dimensional vector of node features $\boldsymbol{x}_{[v]} \in \boldsymbol{x}_{\mathtt{p}}$ component by component as follows:

$$\boldsymbol{x}_{[v,1]} = \begin{cases} 1 & \text{if } v \in \mathcal{V}_{\mathtt{p}}^{\text{mol}} \\ 0 & \text{if } v \in \mathcal{V}_{\mathtt{p}}^{\text{rx}}, \end{cases} \qquad \boldsymbol{x}_{[v,2]} = \begin{cases} 1 & \text{if } v = v_I \\ 0 & \text{otherwise,} \end{cases} \qquad \boldsymbol{x}_{[v,3]} = \begin{cases} 1 & \text{if } v = v_O \\ 0 & \text{otherwise,} \end{cases}$$

where the notation $\boldsymbol{x}_{[v,i]}$ indicates the i-th component of the node feature vector $\boldsymbol{x}_{[v]}$. 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 $(u, v) \in \mathcal{E}_{\mathtt{p}}$, we define its 3-dimensional vector of edge features $\boldsymbol{e}_{[u,v]} \in \boldsymbol{e}_{\mathtt{p}}$ component by component as follows:

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

where the notation $\boldsymbol{e}_{[u,v,i]}$ identifies the i-th component of the edge feature vector $\boldsymbol{e}_{[u,v]}$. 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.

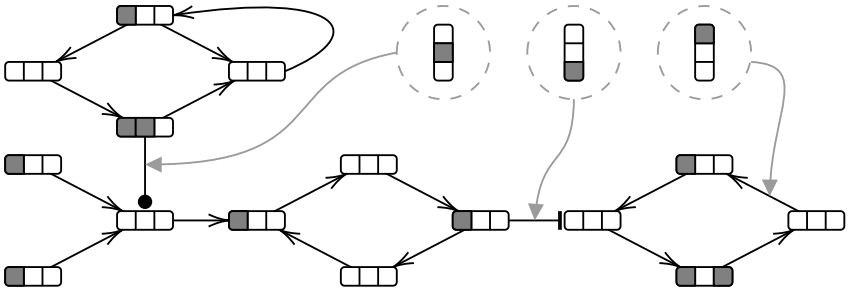


FIGURE 5.6: A representation of the node and edge feature vectors of the induced subgraph shown in Figure 5.5c. Each node is represented as a 3-dimensional binary vector, where 1 is indicated with dark gray and 0 with white. To avoid visual cluttering, we only show one example of edge feature vector 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 DGN architecture for robustness prediction, shown at a high level in Figure 5.7, consists of 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 hyper-parameters of the

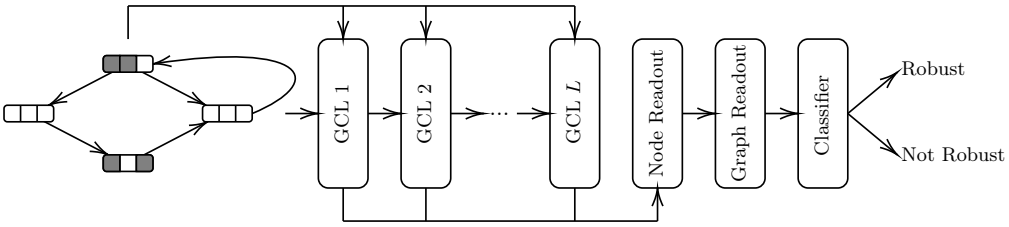


FIGURE 5.7: Model Architecture.

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 use 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 use 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 compare a solution with two baselines: one is simply a "dummy" 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. Following, we term this model DUMMY. The other baseline corresponds to the one presented in Section 4.3, where the graph representation is computed as follows: first, a one hidden-layer MLP, followed by a ReLU, is applied to the node features; then, the transformed node features and summed to obtain the final graph representation, which is passed to the final classifier. We term this model MLPSUM from now on. Notice once again that this model does not operate on the graph structure, taking into account the node features but not their local relationships.

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 final evaluation on the entire dataset;

- in the second phase, termed **E2**, we perform a final evaluation of the architecture on the larger 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 in 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 on a subset of induced subgraphs, specifically those with number of nodes <= 40. This resulted in a dataset with a total of 7036 induced subgraphs. 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.

As regards the types of GCL, we evaluate the following GCL variants:

- Graph Convolutional Network (GCN), as described in Section 3.5.2;
- Graph Isomorphism Network (GIN), as described in Section 4.2;
- Weisfeiler-Leman Graph Convolution (WLGCN), *i.e.* the GCL layer presented in [Mor+19], whose implementation is the following:

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

  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 3.5.2, using three different weight matrices (one for each edge type) for the neighborhood aggregation. More specifically, given an induced subgraph $\mathtt{g}_{[\mathtt{p},(v_I,v_O)]}$ and one of its nodes $v \in \mathcal{V}_{\mathtt{p}}$, the edge-aware neighborhood function used to select neighbors of a certain type is the following:

$$\mathcal{N}_c(v) = \left\{u \in \mathcal{N}(v) \mid \mathbb{I}\left[(u,v) \in \mathcal{E}_{\mathtt{p}}^c\right]\right\},$$

where $c \in \mathcal{C} = \{\text{std}, \text{pro}, \text{inh}\}$. For the SumMLP baseline, we tune the node embedding size, choosing between 768 and 1024, to maintain a number of parameters comparable to the one of our model. Moreover, we tune the learning rate (choosing between 0.001 and 0.0001) and the weight decay penalty (choosing between 0.0001 and 0.00001).

Evaluation Metrics As performances metric, we use 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, considering that:

- 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 positives, *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 hyperparameters evaluated are the same as in **E1**, though the selection is now performed on the full dataset.

Evaluation Metrics In these experiments, there is a high imbalance in favor of the positive class ($\approx 83\%$) with respect to the negative class in the dataset, which is less dramatic for the small dataset ($\approx 73\%$ 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 such as:

- *sensitivity*, also known as True Positive Rate (TPR), defined as $\frac{TP}{TP + FN}$, which intuitively measures how well the classifier detects the positive (robust) class;
- *specificity*, defined as $\frac{TN}{TN + FP}$, which intuitively measures how well the classifier to detects the negative (not robust) class.

Another metric we evaluate, which do not relate 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 $\text{FPR} = 1 - \text{TPR}$ 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.

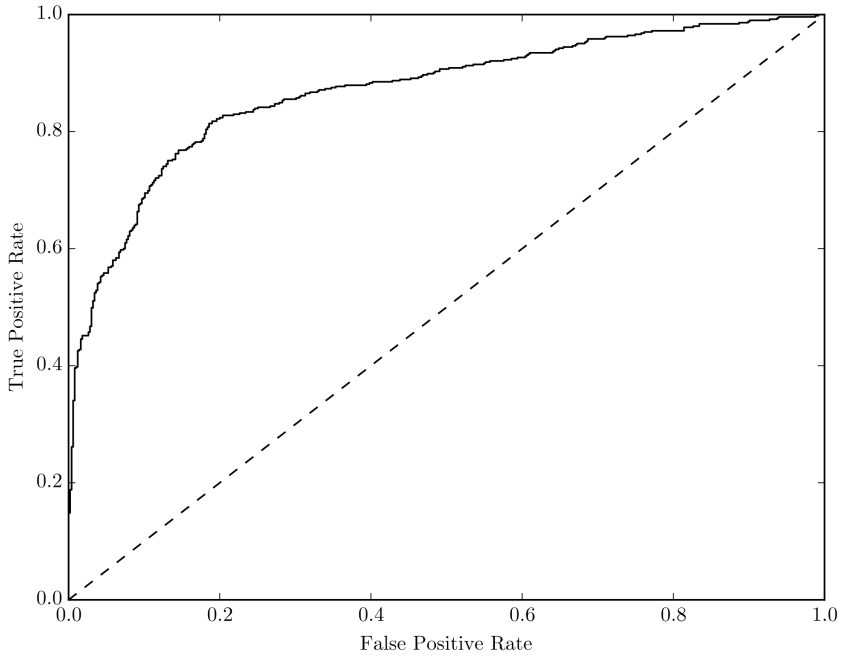


FIGURE 5.8: An example of ROC curve. The dashed line is the ROC curve of a random classifier.

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 connecting 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 (case 2), while an AUROC of 1 describes a perfect classifier (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 equal 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 the work of Fawcett [Faw06].

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 of 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 DUMMY 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 graph structure plays a crucial role in predicting the robustness. This is clear from the results obtained by the MLPSUM baseline, which performs very poorly (similar to the DUMMY one) in comparison;
- 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/without 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 (intended as number of DGN layers) is a good inductive bias for this task. To do so, we perform a *post-hoc* study of 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 overestimate of the true accuracy, the relative difference in performance as the number of

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**
DUMMY	0.732(0.000)
MLPSUM	0.749(0.003)
GCN-vanilla	0.857(0.009)
GIN-vanilla	0.857(0.014)
WLGCN-vanilla	0.862(0.009)
GCN	0.869(0.014)
GIN	0.868(0.008)
WLGCN	0.869(0.012)

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 (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.

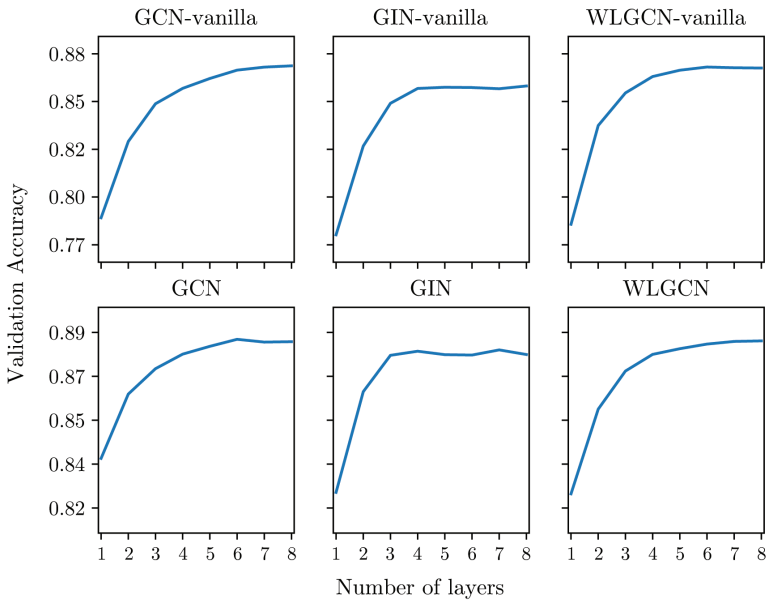


FIGURE 5.9: The effect of the number of DGN layers to the performances of the models.

5.5.2 Results of 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

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	**8985**(1)	**0.913**(0.003)	**0.965**(0.006)	**0.646**(0.042)	**0.948**(0.004)

by number of nodes, in order to also analyze the model performances in relation to 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 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 strata tested. In particular, it performs better when dealing on subgraphs with 21-80 nodes, reaching an average 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

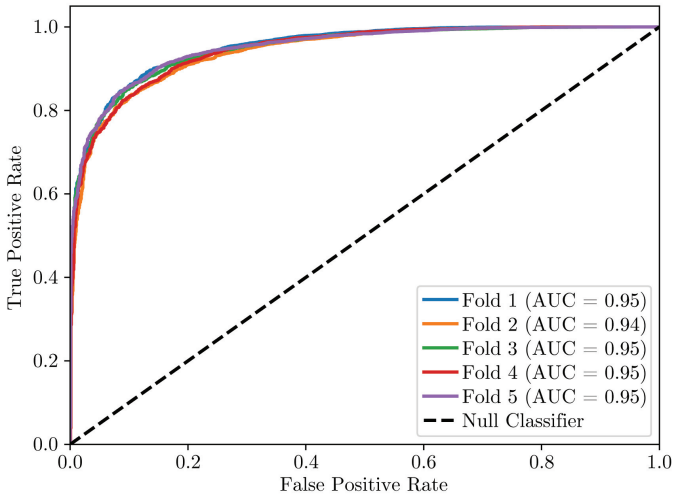
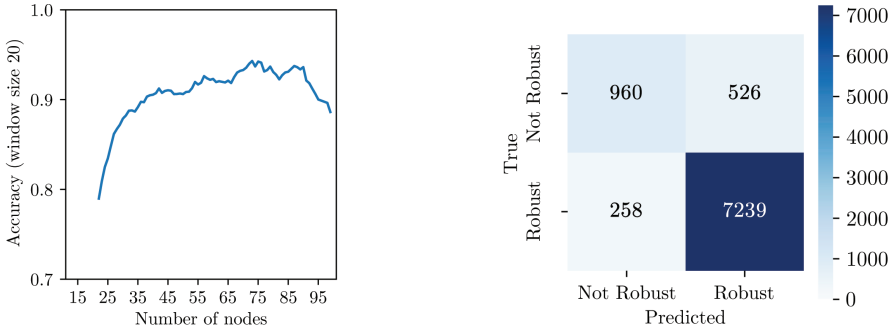


FIGURE 5.10: ROC curves for each of the five test folds. The black dashed line shows the performance of a baseline ("Null") classifier that always predicts one class for reference.

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 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 behavior due to class misproportion in the dataset.

Lastly, we emphasize the fact that, once trained, the time needed to obtain a prediction from the model is in the order of milliseconds, while performing numerical simulations can be orders of magnitude slower. For this specific case, the numerical simulations of the majority of the considered models (as detailed in Section 5.3.2) requires an amount of time in the order of minutes, while bigger models take many hours to produce an output value. While training and evaluating the model is expensive and can take hours, it needs to be performed only once. To conclude, we believe that, once perfected, methods inspired by our approach have the potential of enabling faster advances in understanding the functioning of cells through pathway modelling.



(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.

5.6 Case Studies

Following, we present two application cases of the model: the first is on the synthetic pathway shown in Figure 5.1, and serves as an explanation tool of why the model performs sub-par in cases of smaller subgraphs; the second is an example of how the model can correctly infer the robustness relations in a real-world pathway.

Application 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:

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

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 for each 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 not directly com-

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)

parable: those in Table 5.3a are exact robustness values, while those in Table 5.3b are probabilities of the robustness values being 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 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 (see Figure 5.1). 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.

Application on a Model of the EGF Pathway As a second use case, we consider the SBML model of the Epidermal Growth Factor (EGF) pathway proposed by Sivakumar et al. in [Siv+11]. This model corresponds to model `BIOMD0000000394` of the BioModels database, and is shown in Figure 5.12 as represented by the CellDesigner tool [Fun+03]. We adopt this visual representation style for this case study for readability, but the pathway can can be trivially translated into a PPN. The pathway

describes, in a simplified way, the transduction of the EGF signal, and the consequent activation of the mitogenesis and cell differentiation processes (modelled as an abstract species in the pathway). When the EGF signal protein is available in the cell environment, it can be perceived by the receptor protein EGFR (where R stands for Receptor), which then initiates a cascade of reactions inside the cell, ultimately leading to the activation of mitogenesis and differentiation. The initial steps of the pathway give rise to a rather big complex, involving an activated EGFR dimer and a number of other proteins (the big box in the upper-left corner of Figure 5.12). The formation of such complex is described in a very simplified way in this pathway model, by concentrating everything in only two reactions. The big complex then promotes a cascade of reactions inside the cell, which are modelled more in detail.

Considering the Mitogenesis Differentiation (abstract) species as output (in pink, bottom left corner of Figure 5.12), with respect to EGF and EGFR as input (in green and yellow, respectively, top left corner of Figure 5.12), numerical simulations assign a very high robustness (> 0.995) in both cases. This is actually expected in a signal transduction pathway, since it behaves as an amplifier that must be able to activate the target cell process despite perturbations in the signal and receptor concentrations. On the other hand, if we look at the robustness of the first portion of the pathway up to the creation of the big complex, we can then observe a different behavior. Specifically, if we consider the big complex as output and EGF as input, we still obtain a very high robustness (> 0.999). However, when the input is EGFR, we obtain a robustness value of only 0.19. Again, this is not surprising, since EGF is modelled in the pathway as a promoter of the first reaction (*i.e.* , it is not consumed), while EGFR is a reactant, and it will be included in the big complex.

In this case, the model correctly captures the different roles of EGF and EGFR. Indeed, using EGF as input and the big complex as output, the model gives 0.9474 ± 0.014 as probability of robustness, whereas it gives 0.145 ± 0.121 when the input is with EGFR. The model also captures the robustness of the whole pathway, namely the previous case where either EGF or EGFR were considered as input, and the abstract Mitogenesis Differentiation as output. Precisely, it gives probability 0.973 ± 0.005 with EGF as input and 0.970 ± 0.008 with EGFR as input.

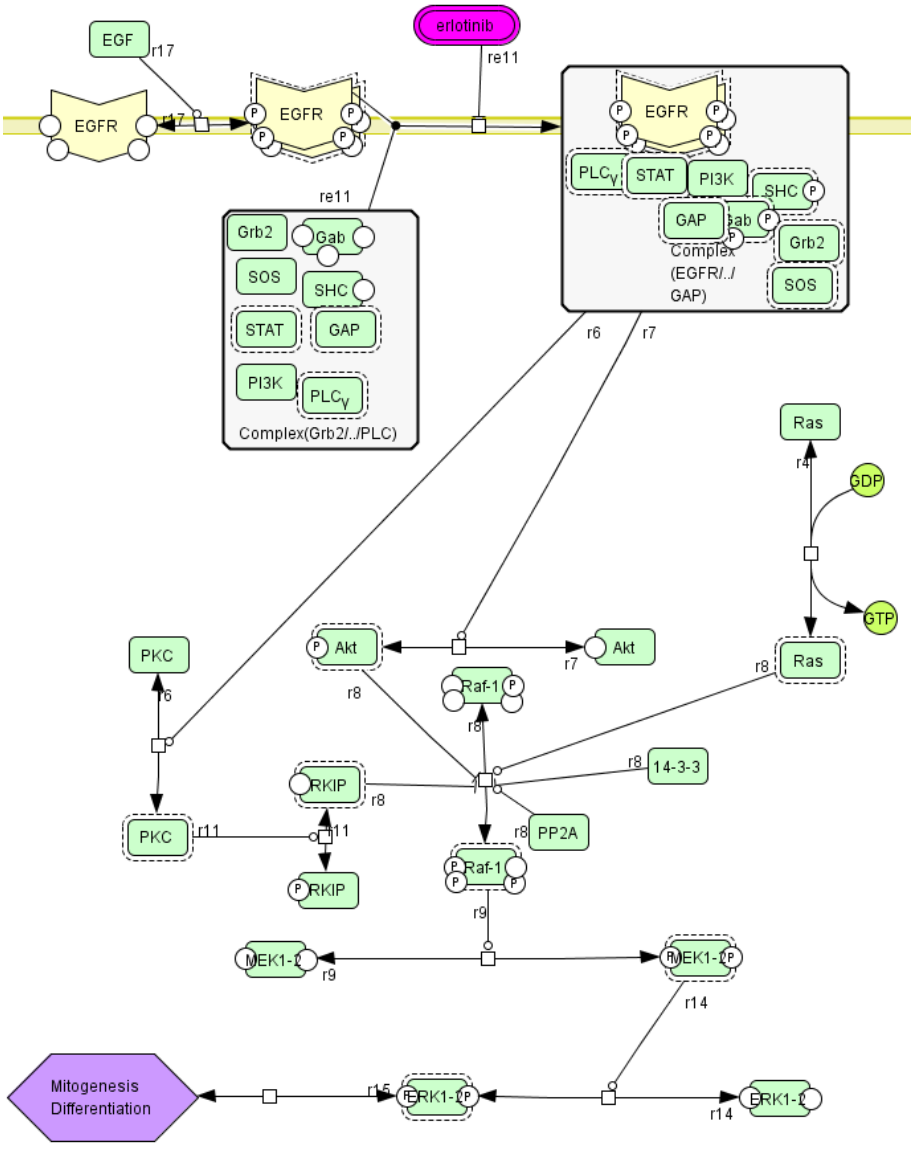


FIGURE 5.12: A model of the EGF pathway by Sivakumar et al. [Siv+11]. Image generated form the SMBL file in the BioModels database using the CellDesigner tool.

Part III

Deep Generative Learning on Graphs and Applications to Computational Chemistry

Chapter 6

A Model for Edge-Based Graph Generation

In this chapter, we introduce a novel generative model for graphs, capable of generating unattributed graphs coming from very different graph distributions. We transform graphs into sequences of ordered edges, from which we extract two sequences derived from the edge endpoints. The proposed model consists of two RNNs to learn the probability distribution of such sequences: the first is an autoregressive network which generates a specification of the graph to produce, which is completed into a graph by the second network. We experiment extensively with the proposed model, comparing its performances with a pool of baselines, one of which is a DGM of graphs that holds state-of-the-art performances at the generative task. The experimental framework has been designed to evaluate the proposed model on concerning both quantitative and qualitative aspects, as discussed in Section 3.6.4. Our experiments demonstrate that, under our evaluation framework, the proposed model is able to perform at, and sometimes surpass, the state of the art in the task of generating graphs coming from very different distributions. Furthermore, we study the effect of changing the order of the edge sequence by experimenting with different node orderings. We show that the chosen node ordering strategy is more effective for learning complex dependencies than the alternatives, and produces graphs of better quality.

6.1 Methods

In this section, we present the methodologies used to develop the model. In particular, we formally introduce the concept of ordered edge sequences, we describe the model, and we show how it is trained and how graph generation is achieved.

6.1.1 Ordered Edge Sequences

Let $\mathbf{g} = \langle \mathcal{V}_\mathbf{g}, \mathcal{E}_\mathbf{g} \rangle$ be a fully connected unattributed graph with n nodes and m edges. We assume $\mathbf{g}$ is undirected for simplicity, without loss of generality. Let $\gamma : \mathcal{V}_\mathbf{g} \to \mathbb{N}_+$ be a bijective node labelling function which assigns a unique positive integer (which we call node ID) to each node in the graph; thus, γ defines a total order over the

nodes of g. The *ordered edge sequence* $\boldsymbol{\mathcal{S}}$ of graph g is the sequence of pairs:

$$\boldsymbol{\mathcal{S}}_{\mathsf{g}} = ((s_1, e_1), (s_2, e_2), \dots, (s_m, e_m)),$$

where $(\gamma^{-1}(s_i), \gamma^{-1}(e_i)) \in \boldsymbol{\mathcal{E}}_{\mathsf{g}}$. Moreover, $\boldsymbol{\mathcal{S}}_{\mathsf{g}}$ is ordered lexicographically according to the IDs assigned to the nodes, *i.e.* $(s_i, e_i) \leq (s_j, e_j)$ if and only if $s_i < s_j$, or $s_i = s_j$ and $e_i \leq e_j$. Given a generic pair $(s_i, e_i) \in \boldsymbol{\mathcal{S}}_{\mathsf{g}}$, we call $s_i \in \mathbb{N}_+$ its *starting node* and $e_i \in \mathbb{N}_+$ its *ending node*. Finally, let us define the *starting sequence* $\tau_s(\boldsymbol{\mathcal{S}}_{\mathsf{g}}) = (s_1, s_2, \dots, s_m)$, the sequence corresponding of starting nodes ordered as in $\boldsymbol{\mathcal{S}}_{\mathsf{g}}$, and analogously, the *ending sequence* $\tau_e(\boldsymbol{\mathcal{S}}_{\mathsf{g}}) = (e_1, e_2, \dots, e_m)$, corresponding to the ending nodes ordered as in $\boldsymbol{\mathcal{S}}_{\mathsf{g}}$. For conciseness, let us omit the dependence of $\boldsymbol{\mathcal{S}}_{\mathsf{g}}$ on the graph g, and of the starting and ending sequences from $\boldsymbol{\mathcal{S}}_{\mathsf{g}}$ whenever they are clear from the context. Clearly, the choice of the labelling function γ is critical in determining the ordered sequence of a graph. Given the graph g, we choose to implement γ with the following algorithm:

- first, select a node v_1 at random from its set of nodes $\mathcal{V}_{\mathsf{g}}$, and set its node ID as $\gamma(v_1) = 1$;
- then, traverse the graph in breadth-first order. Let $V = (v_2, v_3, \dots, v_n)$ be the ordered sequence of nodes visited during the traversal, excluding v_1. Assign node ID $\gamma(v_i) = \mathsf{i}$, $\forall v_i \in V$, $i = 2, \dots, n$.

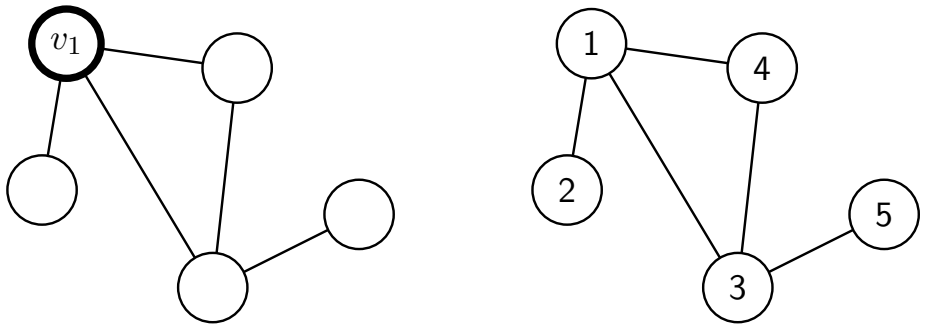
Once the ordering is established, the ordered edge sequence is obtained by sorting the graph edges (now labelled by the ID of their endpoints) according to the lexicographic order. Assuming graph g has the structure shown in Figure 6.1a, and that node v_1 is chosen as the root node for the visit, an example of how the graph nodes are labeled by γ is shown in Figure 6.1b. Notice that γ is trivially bijective, since it assigns a different integer to each node. Once the nodes are labeled, the ordered edge sequence of g is $\boldsymbol{\mathcal{S}} = ((1,2), (1,3), (1,4), (3,4), (3,5))$, with $\tau_S = (1,1,1,3,3)$ and $\tau_E = (2,3,4,4,5)$. Notice that the graph can be readily reconstructed from its ordered edge sequence by first applying the inverse function γ^{-1} to each element of its pairs to obtain $\boldsymbol{\mathcal{E}}_{\mathsf{g}}$, which in turn gives $\mathcal{V}_{\mathsf{g}}$ since we assumed that g is fully connected.

6.1.2 Model

Our goal is to model $p(\boldsymbol{\mathcal{S}})$, the probability of ordered edge sequences, using a dataset $\mathbb{D} = \left\{\boldsymbol{\mathcal{S}}_{(i)}\right\}_{i=1}^{n}$. Our key observation is that any ordered edge sequence $\boldsymbol{\mathcal{S}}$ is uniquely defined by its starting and ending sequences. Therefore, instead of working on the ordered edge sequence directly, we work on their starting and ending sequences. Specifically, we model the probability of sampling $\boldsymbol{\mathcal{S}}$ from $p(\boldsymbol{\mathcal{S}})$ as follows:

$$p(\boldsymbol{\mathcal{S}}) = p(\tau_S, \tau_E) = p(\tau_E \mid \tau_S)\, q(\tau_S) = \prod_{i=1}^{|\boldsymbol{\mathcal{S}}|} p(e_i \mid s_i) \prod_{j=1}^{|\boldsymbol{\mathcal{S}}|} q(s_j \mid s_{<j}).$$

Intuitively, the generative process specified by the distribution is the following:



(A) (B)

FIGURE 6.1: (A): an example graph, where the starting node v_1 of the labelling algorithm is marked with a thicker border. (B): the same graph, where nodes are labeled according to a breadth-first visit of the graph rooted at v_1.

- first, one samples τ_S from a prior probability $q(\tau_S)$, which is modeled autoregressively;
- once τ_S is available, it is used to condition the prediction of the ending sequence τ_E.

Once both τ_S and τ_E are available, the ordered edge sequence (and thus, the corresponding graph) can be reconstructed simply pairing their elements. Here, we make an important remark. Notice that the probability of $q(\tau_S)$ is modeled autoregressively, while the conditional $p(\tau_E \mid \tau_S)$ is not. This means that to generate a sequence, the only stochastic part in the model relates to $q(\tau_S)$, while predicting τ_E is deterministic once τ_S is known. One possible interpretation of $q(\tau_S)$ is that it acts as a "soft prior" of the structure of the graph: in fact, it specifies a subset of relevant nodes for the generation and their degree, expressed as the number of times they appear in the starting sequence. The conditional $p(\tau_E \mid \tau_S)$ uses this information to "complete" the graph based on the information provided by the prior.

6.1.3 Training

The model specified above is implemented as two RNNs in cascade, which we refer to as $q_{\boldsymbol{\psi}}$ and $p_{\boldsymbol{\phi}}$ respectively. The two sequences (starting and ending) are first encoded as sequences of $(k+2)$-dimensional one-hot vectors, where k is the largest node ID assigned by γ to any node in the dataset, and $+2$ is added for the start of sequence and end of sequence tokens. We indicate the one-hot encoded starting sequence with the notation $\boldsymbol{s} = (\boldsymbol{s}_{[1]}, \dots, \boldsymbol{s}_{[|\tau_S|]}, \langle \mathsf{E} \rangle)$, where $\boldsymbol{s}_{[i]} \in \mathbb{R}^{k+2}$, and analogously, we use $\boldsymbol{e} = (\boldsymbol{e}_{[1]}, \dots, \boldsymbol{e}_{[|\tau_E|]})$ for the one-hot encoded ending sequence, with $\boldsymbol{e}_{[i]} \in \mathbb{R}^{k+2}$. Notice that an end of sequence token is added to $\boldsymbol{s}$: when it is predicted, the autoregressive process that yields $\boldsymbol{s}$ is interrupted. Our dataset has thus the following form: $\mathbb{D} = \left\{(\boldsymbol{s}_{(i)}, \boldsymbol{e}_{(i)})\right\}_{i=1}^{n}$. Focusing on a single training pair $(\boldsymbol{s}, \boldsymbol{e})$, we explain the feed-forward phase of the network. Firstly, the sequence $\boldsymbol{s}$ is processed by $q_{\boldsymbol{\psi}}$. Given an element

$\boldsymbol{s}_{[i]} \in \boldsymbol{s}$, the per-element output of $q_{\boldsymbol{\psi}}$ is obtained as follows:

$$\begin{aligned}\boldsymbol{h}_{[i]} &= \mathrm{GRU}_{\boldsymbol{\psi}}(\boldsymbol{E}^{\mathsf{T}}\boldsymbol{s}_{[i-1]}, \boldsymbol{h}_{[i-1]})\\ \boldsymbol{o}_{[i]} &= \mathrm{softmax}(\boldsymbol{V}_{\boldsymbol{\psi}}\boldsymbol{h}_{[i]} + \boldsymbol{b}_{\boldsymbol{\psi}}).\end{aligned}$$

In the formula above, $\boldsymbol{E} \in \mathbb{R}^{h\times(k+2)}$ is an *embedding matrix* whose entries are learnable, and the dot product selects the column of $\boldsymbol{e}$ corresponding to the embedding of the node ID assigned to $\boldsymbol{s}_{[i]}$. Furthermore, $\boldsymbol{h}_{[i]} \in \mathbb{R}^h$ is a hidden state, $\mathrm{GRU}_{\boldsymbol{\psi}}$ is a multi-layer GRU network, $\boldsymbol{V}_{\boldsymbol{\psi}} \in \mathbb{R}^{(k+2)\times h}$ and $\boldsymbol{b}_{\boldsymbol{\psi}} \in \mathbb{R}^{k+2}$ are the weights of the softmax output layer, and $\boldsymbol{o}_{[i]}$ is an output distribution over all the possible node IDs. The process is initialized by setting $\boldsymbol{s}_{[0]} = \langle \mathsf{S} \rangle$ and $\boldsymbol{h}_{[0]} = \boldsymbol{0}$. We use teacher forcing, meaning that, at each step, the input of the network is the ground truth value taken from $\boldsymbol{s}$, rather than the output predicted by the network. Lastly, the output $\boldsymbol{o}_{[i]}$ is compared to the ground truth value $\boldsymbol{s}_{[i]}$ using the CE loss function. For the whole starting sequence, the computed loss is the following:

$$\mathcal{L}(\boldsymbol{\psi}, \boldsymbol{s}) = \frac{1}{|\boldsymbol{s}|+1}\sum_{i=1}^{|\boldsymbol{s}|+1} q_{\boldsymbol{\psi}}(\boldsymbol{s}_{[i]} \mid \boldsymbol{s}_{[<i]}) = \frac{1}{|\boldsymbol{s}|+1}\sum_{i=1}^{|\boldsymbol{s}|+1} \mathrm{CE}(\boldsymbol{s}_{[i]}, \boldsymbol{o}_{[i]}).$$

After the starting sequence $\boldsymbol{s}$ has been processed by $q_{\boldsymbol{\psi}}$, the control flow passes onto $p_{\boldsymbol{\phi}}$. The function computed by $p_{\boldsymbol{\phi}}$ is similar to the one implemented by the first RNN. Specifically, it is the following:

$$\begin{aligned}\boldsymbol{h'}_{[i]} &= \mathrm{GRU}_{\boldsymbol{\phi}}(\boldsymbol{E}^{\mathsf{T}}\boldsymbol{s}_{[i]}, \boldsymbol{h'}_{[i-1]})\\ \boldsymbol{o'}_{[i]} &= \mathrm{softmax}(\boldsymbol{V}_{\boldsymbol{\phi}}\boldsymbol{h'}_{[i]} + \boldsymbol{b'}_{\boldsymbol{\phi}}),\end{aligned}$$

where $\boldsymbol{h'}_{[0]} = \boldsymbol{h}_{[|\tau_S|]}$, *i.e.* the second network is initialized with the last hidden state of the first. Moreover, $\boldsymbol{h'}_{[i]} \in \mathbb{R}^h$ is a hidden state, $\mathrm{GRU}_{\boldsymbol{\phi}}$ is a multi-layer GRU network, $\boldsymbol{V}_{\boldsymbol{\phi}} \in \mathbb{R}^{(k+2)\times h}$ and $\boldsymbol{b'}_{\boldsymbol{\theta}} \in \mathbb{R}^{k+2}$ are the weights of the softmax output layer, $\boldsymbol{o}_{[i]}$ is an output distribution over all the possible node IDs, and $\boldsymbol{E}$ is the same embedding matrix used by $q_{\boldsymbol{\psi}}$. Lastly, the output $\boldsymbol{o'}_{[i]}$ is compared to the ground truth value $\boldsymbol{e}_{[i]}$ using the CE loss function, similarly as before. For the whole ending sequence, the computed loss is the following:

$$\mathcal{L}(\boldsymbol{\phi}, (\boldsymbol{s}, \boldsymbol{e})) = \frac{1}{|\boldsymbol{e}|}\sum_{i=1}^{|\boldsymbol{e}|} -\log p_{\boldsymbol{\phi}}(\boldsymbol{e}_{[i]} \mid \boldsymbol{s}_{[i]}) = \frac{1}{|\boldsymbol{e}|}\sum_{i=1}^{|\boldsymbol{e}|} \mathrm{CE}(\boldsymbol{e}_{[i]}, \boldsymbol{o'}_{[i]}).$$

The whole network is trained with MLE by minimizing the following objective function:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{(\boldsymbol{s},\boldsymbol{e})\in\mathbb{D}} \mathcal{L}(\boldsymbol{\psi}, \boldsymbol{s}) + \mathcal{L}(\boldsymbol{\phi}, (\boldsymbol{s}, \boldsymbol{e})),$$

where $\boldsymbol{\theta} = (\boldsymbol{\phi}, \boldsymbol{\psi})$. Figure 6.2 shows the architecture during training.

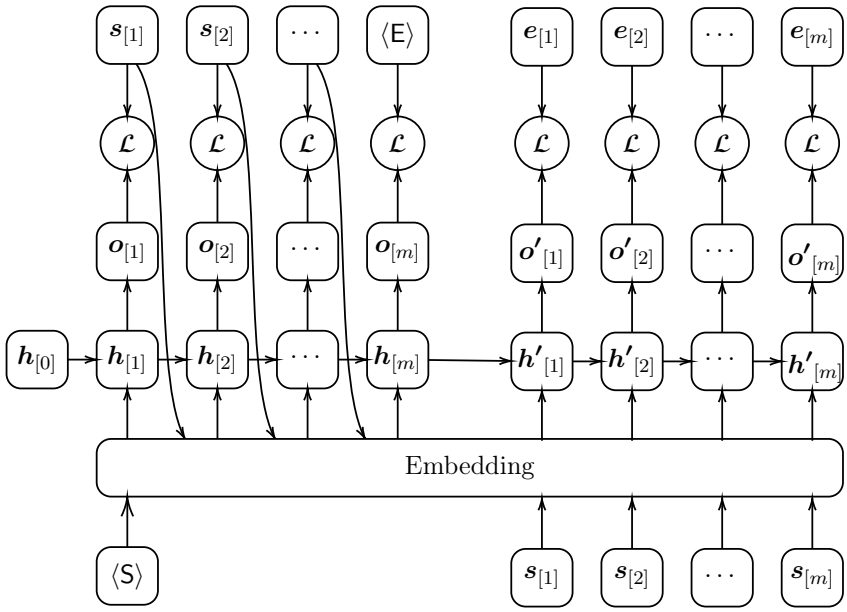


FIGURE 6.2: A depiction of the proposed architecture during training. We set $|\tau_S| = |\tau_E| = m$ to avoid cluttering. Notice that the embedding matrix is shared between the two networks.

6.1.4 Generation

To generate an ordered edge sequence, one first generates a starting sequence $\tilde{\boldsymbol{s}}$ from the first network in autoregressive sampling mode. The generative process is started by passing the ⟨S⟩ token to the network. The categorical distribution corresponding to all the possible Node IDs is sampled at each step, and the resulting node ID is used as input for the next step. The autoregressive sampling is interrupted once the ⟨E⟩ token is sampled. At this point, the sampled starting sequence $\tilde{\boldsymbol{s}}$ is used as input for the second network, which predicts the ending sequence. This time, the output distribution is not sampled, but the token with the highest probability $\hat{\boldsymbol{e}}_{[i]}$ is chosen at each step. This method is also known as *greedy sampling*, and corresponds to applying the arg max operator to the distribution vector, which gives the ID of the chosen node. After all elements have been predicted, we end up with a starting sequence $\tilde{\boldsymbol{s}}$, and an ending sequence $\hat{\boldsymbol{e}}$. The two sequences are paired together to obtain the ordered edge sequence of the generated graph. Figure 6.3 shows the generative process visually.

6.1.5 Implementation Details

The model is implemented using the `PyTorch`[1] [Pas+17] library; after an initial exploratory phase (not reported), some hyper-parameters (such as the number of recurrent layers) were fixed in advance. The other hyper-parameters are selected with an 80%-20% internal model selection. We choose among 32 and 64 for the embedding dimension d, 128 and 256 for the recurrent state size h. We apply a dropout layer to the embedding layer, whose rate is chosen between 0.25 and 0.35. As recurrent cells,

[1] https://github.com/marcopodda/grapher

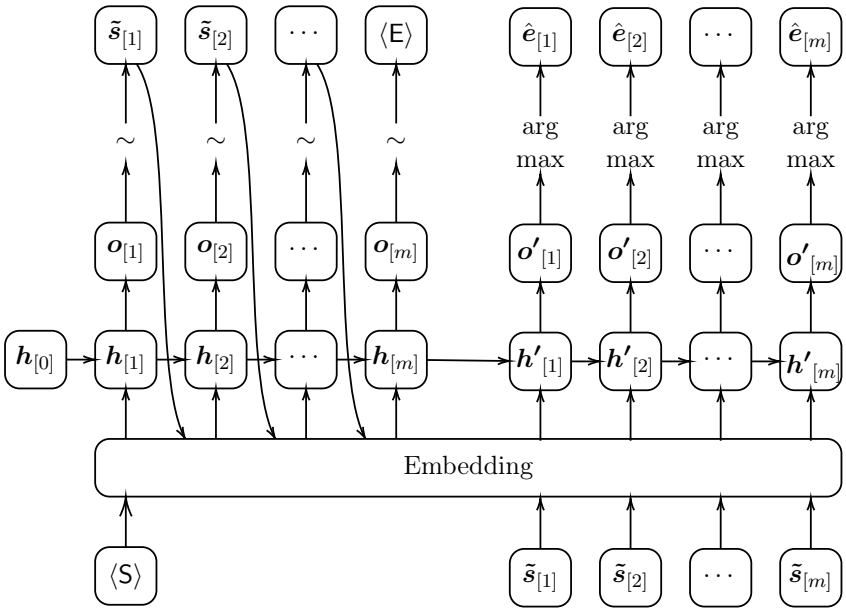


FIGURE 6.3: Graph generation using the proposed model. The output of the network are two sequences: one is a stochastic sequence $\tilde{\boldsymbol{s}}$ representing the starting sequence of the graph. The other is a deterministic ending sequence $\hat{\boldsymbol{e}}$, predicted using $\tilde{\boldsymbol{s}}$ as input. The two sequences are ultimately combined to produce the ordered edge sequence of a novel graph (not shown).

we use 2 layers of Gated Recurrent Units (GRU) [Cho+14]. As regards the learning parameters, we use the Adam [KB15] optimizer with learning rate of 0.001, halved every 200 epochs. We train all models for a maximum of 2000 epochs, applying early stopping whenever the loss starts to plateau (we found a suitable empirical threshold after running a number of exploratory training instances).

6.2 Experiments

Following, we detail the experiments to assess our model, describing the datasets used for learning, the baselines we compared to, and the framework used for evaluation.

6.2.1 Datasets

In our experiments, we evaluate our model on 3 synthetic datasets and 2 real-world datasets of molecular graphs. Every synthetic dataset represents graphs with particular node/edge dependencies. The rationale is to assess whether our approach is general enough to give good performances on very different graph distributions. Concretely, we use the following datasets:

- LADDERS, a synthetic dataset of ladder graphs. We generate all possible ladder graphs having a number of nodes equal to $2n, n = 2, \ldots, 20$, and replicate them 10 times, resulting in a total size of 180 graphs. This dataset is inspired from the GRIDS dataset used in [You+18b]. In our case, we use ladder graphs because

they have a similar node degree distribution, while being computationally manageable. In this dataset, the model has to capture very strong dependencies: the nodes of a ladder graph have degree of 3, except nodes at the four corners, which have degree of two. Any graph that does not respect these dependencies is not a ladder graph. An example of ladder graph is shown in Figure 6.4a;

- COMMUNITY, a synthetic dataset of graphs with two-community structure. Community graphs are composed of two densely connected clusters of nodes, which are weakly connected between themselves. Here, the model has to capture the community structure. Community dependencies are very common in biological settings: for example, densely connected communities in metabolic networks often represent functional groups (see *e.g.* [GN02]). To create this dataset, we firstly generate two clique graphs of random size between $8 \leq N \leq 20$. We then remove some edges randomly from the two clusters with probability 0.4, and then connect the two communities with 1 or 2 edges at random. The generated dataset is composed of 1000 graphs. This dataset is similar to the COMMUNITY dataset used in [You+18b], which unfortunately we could not reproduce. An example of community graph is shown in Figure 6.4b;

- EGO, a dataset of ego networks extracted from the Citeseer dataset [GBL98]. In this case, the model has to capture the presence of a focal node (the "ego" node), which is connected to the majority of nodes in the graph, while the other nodes (the "alter" nodes) have weaker connectivity. This dependency is typical of social networks, and can be modeled with the BA model for preferential attachment. To create the dataset, we extract all possible ego networks of radius 2 from the Citeseer dataset. Thus, the path length between the ego node and the alter nodes is at most 2. The total number of graphs in the dataset is 1719. An example of ego graph is shown in Figure 6.4c;

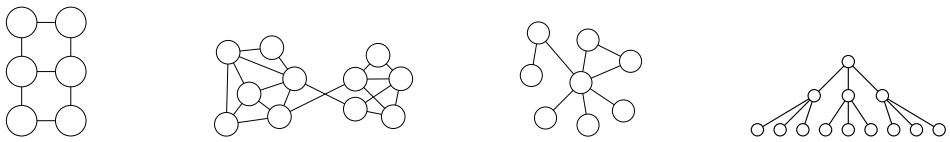
- TREES, a synthetic dataset of balanced and unbalanced ternary trees presented in [BC19]. The dataset is composed of 780 ternary trees, of which one third is symmetric (*i.e.* the number of nodes is almost identical in each subtree at the same level), one third left-asymmetric, and one third right-asymmetric. A left-asymmetric (right-asymmetric) tree is one where if the number of nodes in the leftmost (rightmost) position is greater than the number of nodes in the opposite position, whilst symmetric trees, have an almost equivalent number of nodes for each position. The dataset is divided into 600 trees for training (200 symmetric, 200 left-symmetric, 200 right-symmetric), and 180 trees for testing (60 symmetric, 60 left-symmetric, and 60 right-symmetric);

- ENZYMES, a subset of 436 graphs taken from the ENZYMES dataset [Sch+04] (see Section 4.1 for details);

- PROTEINS, a subset of 794 graphs taken from the Dobson and Doig dataset [DD03] (see Section 4.1 for details). In these two last datasets, the model should

capture patterns of connectivity typical of molecules, such as functional groups or aromatic rings.

All datasets have a number of nodes comprised between 4 and 40 (the only exception is TREES, where the number of nodes is comprised between 34 and 229), and a number of edges comprised between 2 and 130 (33 and 228, respectively, for TREES). Before training, we held out a portion of the dataset for testing purposes (except for TREES, where the split was provided in advance). In particular, for all datasets except LADDERS, we held out 30% of graphs, and used them for evaluation. This held-out set acts as a random sample drawn from the corresponding unknown graph distribution. We also considered experimenting with two other graph families, namely



(A) LADDERS. (B) COMMUNITY. (C) EGO. (D) TREES.

FIGURE 6.4: Examples of the synthetic graphs used for the evaluation.

cycles and complete graphs. However, we ultimately decided not to for two main reasons. Firstly, the number of unlabeled cycles and complete graphs is fixed, (exactly 1 for a given number of nodes). In the case of complete graphs especially, the connectivity of each graph is quadratic in the number of nodes, which would have made a dataset of reasonable size computationally unmanageable. Secondly, cyclic and pseudo-complete sub-structures are already represented in the pool of datasets chosen: for example, ladder graphs are made of overlapping cycles, while the community graphs in the COMMUNITY dataset are densely connected (and thus very similar to complete graphs). For these reasons, we decided to carry on the experiments only with the above-mentioned datasets.

For the LADDERS dataset, we held out 10% of graphs in a stratified fashion. In practice, the held-out set for LADDERS is composed of the same 18 unique ladder graphs found in the training set. To motivate this choice, let us clarify the role of the LADDERS dataset: its purpose is not to evaluate generalization capability, but to show that adaptive models can overfit hard node/edge dependencies among nodes, while non-adaptive models (such as the ER and BA models) cannot. The statistics of the datasets used for evaluation are presented in Table 6.1.

6.2.2 Baselines

We compare against four baseline models. Two of them are classical generative models of graphs coming from graph theory literature, namely the ER and BA models. The rationale behind this choice is to assess whether our model is able to perform better than random models that model graph connectivity independently (ER), or consider just one simple edge dependency (BA, where the probability of an edge is a function of the node degree).

Table 6.1: Statistics of datasets used in the experiments.

Dataset	No. of graphs	Test size	Avg. no. of nodes	Avg. no. of edges
LADDERS	180	18	21.00	29.50
COMMUNITY	1000	300	24.38	88.58
EGO	1719	516	13.02	18.51
TREES	780	180	109.14	108.14
ENZYMES	436	130	26.14	51.16
PROTEINS	794	238	20.49	38.67

The ER model has two parameters: n, the number of nodes of the graph to be generated, and P, the probability of adding an edge between a pair of nodes. The generative process of the ER model can be described informally as follows: first, pick n, then, for each possible pair of nodes, sample a Bernoulli random variable with probability P, and connect the two nodes according to the sampled value. We sample n from the empirical distribution of the number of nodes in the datasets, and choose P from a grid of values. The best value of P is obtained by minimizing the earth mover distance between the empirical distribution and the generated distribution of graph properties.

Similarly, the BA model has two parameters: n, the number of nodes of the graph to be generated, and M, a maximum number of nodes a node can be connected to. The generative process for a BA graph proceeds incrementally: given an already formed graph, add a new node and connect it (or not) to at most M nodes with probability proportional to the nodes' degree. In our experiments, the two parameters of the BA model are optimized similarly to the ER model.

Besides graph theory baselines, we also compare to a strong Deep-Learning based generative baseline. In particular, we choose the GraphRNN model of You et al. [You+18b], which holds state-of-the-art performances in the graph generation task. We implemented the model according to the original code repository provided by the authors, following their implementation and their hyper-parameters configuration closely.

Lastly, we introduce a third baseline model, a recurrent neural network with GRU cells that is trained to predict the adjacency matrix of the graph one entry at a time. We call this baseline GRU-B(aseline) from now on. It is arguably the simplest model one can come up with to model graph generation in a recurrent fashion, with the limitation that it has to sample $n(n-1)/2$ entries to fully specify the adjacency matrix of an undirected graph with n nodes, making it susceptible to learning issues induced by long-term dependencies [BSF94]. Clearly, even though Deep-Learning based, this baseline has purposely limited expressiveness with respect of our model and GraphRNN.

6.2.3 Evaluation Framework

We assess our model against the baselines following the quantitative and qualitative evaluation principles described in Section 3.6.4. The experiments are described as follows:

- the first experiment consists in evaluating the model quantitatively. To confidently detect overfitting, we choose to generate samples larger than the training set. Specifically, we generate two samples of size 1000 and 5000 from all the candidates. For each generated sample, we measure their novelty (with respect to the training set) and their uniqueness (with respect to the same generated graphs). Despite using chemical datasets (ENZYMES and PROTEINS), we do not evaluate chemical validity because we are concerned about generating unlabeled graphs; thus, chemical validity cannot be assessed in our case. Finally, we also measure the time that each model takes to generate the 5000 graph sample;
- the second experiment is of qualitative nature. In practice, we assess how much the generated samples resemble a random sample from the graph distribution of reference. The assessment consists of extracting local and global statistics from the generated graphs, and comparing their distribution to that of the test graphs. The local statistics evaluated are Degree Distribution (DD), Clustering Coefficient (CC), and Orbit Counts (OC). For the global statistics, we choose Betweenness Centrality (BC) and Neighborhood Subgraph Pairwise Distance Kernel (NSPDK) [CG10]. Given a graph statistics, the discrepancy between the generated and reference distributions is computed as follows. Initially, a vector of statistics is extracted for each graph, and its length is equalized to 100 by fitting a histogram to its values. Then, the histograms are summed across all the samples of the generated graphs and the reference graphs, respectively. Finally, the KLD between the two resulting vectors is computed. The only exception is NSPDK, where we compute the Maximum Mean Discrepancy [Gre+12] instead. Since the NSPDK only works with labelled graphs, we use node degrees as labels. All the calculations are repeated 10 times, each time using a different generated sample. In the following tables, the mean and the standard deviation of these 10 runs is reported.

6.3 Results

Here, we present the results of our experiments. We divide the analysis of the results into a quantitative and qualitative sections for readability purposes.

6.3.1 Quantitative Analysis

Table 6.2 shows the results of our quantitative experiments. The ER model achieves the best performances in the LADDERS and COMMUNITY datasets: this was expected, since the model produces random graphs that are very likely to be different

from the training sample. Similarly, the BA model generates graphs whose structure is radically different from graphs coming from these two datasets by construction. In the EGO dataset, both models score poorly with respect to the competitors. We argue that this result is due to the nature of the EGO dataset, which is composed of graphs with very weak connectivity (except for the ego nodes) and a very low number of cycles. With such characteristics, it is easier for a random model to produce duplicates or overfit the training sample, just by setting the parameters that regulate connectivity to small values. In contrast, our model and GraphRNN consistently

TABLE 6.2: Results of the quantitative analysis of the generated samples. In the leftmost column, both the metric of interest as well as the sample size (either 1000 or 5000) is specified. Best performances of models based on RNNs are bolded.

Dataset	**Metric**	**ER**	**BA**	**GRU-B**	**GraphRNN**	**Ours**
LADDERS	Novelty@1000	0.98	1.00	0.66	0.97	0.99
	Novelty@5000	0.99	1.00	0.71	0.96	0.99
	Uniqueness@1000	0.98	0.86	0.17	0.26	0.22
	Uniqueness@5000	0.86	0.72	0.06	0.08	0.07
	Time@5000	<1s	1s	6m32s	17m06s	3m38s
COMMUNITY	Novelty@1000	1.00	1.00	1.00	1.00	1.00
	Novelty@5000	1.00	1.00	1.00	1.00	1.00
	Uniqueness@1000	1.00	1.00	1.00	1.00	1.00
	Uniqueness@5000	1.00	1.00	0.99	1.00	1.00
	Time@5000	3s	5s	7m25s	52m26s	8m58s
EGO	Novelty@1000	0.69	0.74	0.62	0.74	0.96
	Novelty@5000	0.73	0.82	0.58	0.76	0.92
	Uniqueness@1000	0.76	0.72	0.85	0.94	0.97
	Uniqueness@5000	0.65	0.61	0.64	0.87	0.91
	Time@5000	1s	1s	6m01s	1h10m16s	3m23s
TREES	Novelty@1000	1.00	1.00	1.00	0.99	1.00
	Novelty@5000	1.00	1.00	1.00	0.99	1.00
	Uniqueness@1000	1.00	1.00	0.89	0.94	0.88
	Uniqueness@5000	0.98	1.00	0.64	0.89	0.70
	Time@5000	8s	8s	22m43s	2h38m53s	16m39s
ENZYMES	Novelty@1000	1.00	1.00	0.99	0.95	1.00
	Novelty@5000	1.00	1.00	0.99	0.95	1.00
	Uniqueness@1000	1.00	1.00	0.98	1.00	1.00
	Uniqueness@5000	1.00	1.00	0.81	0.97	0.92
	Time@5000	3s	3s	7m19s	52m24s	4m41s
PROTEIN	Novelty@1000	0.98	1.00	0.90	0.75	0.95
	Novelty@5000	0.98	1.00	0.91	0.76	0.96
	Uniqueness@1000	0.98	0.93	0.95	0.93	0.83
	Uniqueness@5000	0.97	0.92	0.77	0.84	0.65
	Time@5000	<1s	4s	6m29s	54m07s	3m44s

generate graphs with high novelty and uniqueness rates in almost all scenarios. The only exception to this trend is the LADDERS dataset, where both our model and

GraphRNN overfit (they tend to generate the same graphs over and over), while the ER and BA models perform excellently. Again, these high scores imply that the ER and BA are unable to replicate the structures of these datasets.

The GRU-B model greatly under-performs in several datasets. From this result alone, one could legitimately infer that the model is not generalizing to unseen adjacency matrices. However, the qualitative analysis and the high value of the training loss (not shown) suggest that the model cannot perform any better. This provides evidence that a simple recurrent model such as GRU-B, at least in this form, is not well suited for the task of graph generation.

Our model and GraphRNN obtain excellent novelty and uniqueness scores in every dataset except LADDERS. Again, the poor scores in the LADDERS dataset indicate that they are able to overfit ladder graphs. Our model is the most consistent across all datasets, obtaining a score of at least 0.88 (except for the LADDERS dataset). When the sample size is large, both models tend to produce higher rates of duplicate graphs, which was again expected since the largest sample size exceeds the size of the training set.

Note how all models obtain a perfect score in the COMMUNITY dataset. This can be explained by considering the nature of the COMMUNITY dataset, whose graphs are essentially composed by two random graphs weakly connected among each other. Thus, since generating a graph from that distribution is very similar to generating two strongly connected components weakly connected together, samples are highly likely to be different from each other.

As regards sampling time, random models are the fastest during generation; this was expected, since they have only 2 parameters, while all RNN-based models have a larger number of parameters. Among the RNN-based models, our model is the fastest at generating new samples. One exception is the COMMUNITY dataset, where however it elapses only 1.5 minutes more than the winning model, GRU-B. In contrast, the GraphRNN model has sampling times 5 to 20 times slower than our model. For completeness, however, we report that while we sampled from the GraphRNN model with batch size of 1 to achieve a fair comparison, its implementation allows to drawing samples in batches, greatly speeding up the process.

6.3.2 Qualitative Analysis

Table 6.3 shows the results of the qualitative evaluation of the models. Since we are measuring discrepancies between distributions of graph statistics, we remark that lower values indicate better fit. It can be clearly seen how random models are unable to learn complex dependencies, scoring poorly on all datasets in every considered metric. One exception is the TREES dataset, where the ER model obtains the best KLD for the CC metric. The GRU-B model struggles in almost every metric across all datasets. This reinforces the argument made with the quantitative analysis, *i.e.* that this model suffers from limited expressiveness. Among all models, GraphRNN and ours perform consistently at the state of the art as regards the quality of generated graphs. In most

TABLE 6.3: Results of the qualitative analysis of the generated samples. The three metrics considered are KLDs calculated on Average Degree Distribution (ADD), Average Clustering Coefficient (ACC), and Average Orbit Count (AOD). Best performances are bolded.

Dataset	Metric	ER	BA	GRU-B	GraphRNN	Ours
LADDERS	DD	1.873 (0.134)	0.945 (0.047)	1.098 (0.162)	0.035 (0.003)	**0.013** (0.003)
	CC	**0.000** (0.000)	0.195 (0.029)	0.004 (0.005)	**0.000** (0.000)	**0.000** (0.000)
	OC	1.853 (0.026)	0.136 (0.022)	0.026 (0.006)	**0.000** (0.000)	**0.000** (0.000)
	BC	6.281 (0.877)	1.756 (0.584)	6.461 (1.447)	**0.566** (0.062)	0.762 (0.180)
	NSPDK	0.744 (0.007)	0.569 (0.020)	0.604 (0.110)	0.139 (0.007)	**0.056** (0.022)
COMMUNITY	DD	0.353 (0.002)	1.149 (0.080)	0.167 (0.004)	0.075 (0.000)	**0.013** (0.000)
	CC	0.312 (0.031)	1.643 (0.189)	0.181 (0.012)	**0.074** (0.001)	0.101 (0.011)
	OC	0.073 (0.000)	0.055 (0.001)	0.027 (0.007)	**0.001** (0.000)	0.013 (0.002)
	BC	1.543 (0.039)	2.642 (0.021)	0.228 (0.032)	0.311 (0.063)	**0.038** (0.005)
	NSPDK	0.072 (0.001)	0.156 (0.003)	0.056 (0.001)	**0.009** (0.000)	**0.009** (0.000)
EGO	DD	0.723 (0.005)	0.587 (0.004)	0.244 (0.000)	0.046 (0.001)	**0.029** (0.000)
	CC	0.414 (0.002)	0.698 (0.069)	0.411 (0.005)	0.194 (0.001)	**0.090** (0.011)
	OC	0.121 (0.001)	0.384 (0.000)	0.124 (0.001)	0.009 (0.001)	**0.002** (0.000)
	BC	1.100 (0.021)	0.735 (0.020)	0.127 (0.010)	0.282 (0.024)	**0.096** (0.001)
	NSPDK	0.098 (0.001)	0.038 (0.000)	**0.009** (0.000)	0.014 (0.001)	0.016 (0.001)
TREES	DD	0.729 (0.011)	0.810 (0.004)	0.797 (0.005)	**0.036** (0.002)	0.078 (0.002)
	CC	**0.000** (0.000)	0.026 (0.001)	0.128 (0.003)	**0.000** (0.000)	**0.000** (0.000)
	OC	0.418 (0.023)	0.017 (0.000)	0.081 (0.004)	**0.000** (0.000)	0.001 (0.000)
	BC	0.201 (0.003)	0.754 (0.055)	0.137 (0.003)	0.170 (0.009)	**0.056** (0.008)
	NSPDK	0.597 (0.004)	0.718 (0.003)	0.605 (0.011)	**0.022** (0.001)	0.042 (0.004)
ENZYMES	DD	1.124 (0.046)	1.761 (0.045)	0.475 (0.005)	**0.026** (0.003)	0.040 (0.002)
	CC	0.781 (0.003)	1.680 (0.076)	0.531 (0.051)	**0.113** (0.019)	**0.128** (0.037)
	OC	0.218 (0.005)	0.197 (0.001)	0.022 (0.004)	**0.008** (0.001)	**0.005** (0.002)
	BC	2.784 (0.066)	5.753 (0.470)	0.201 (0.031)	0.086 (0.008)	**0.062** (0.006)
	NSPDK	0.177 (0.006)	0.235 (0.005)	0.104 (0.003)	**0.015** (0.001)	0.021 (0.001)
PROTEINS	DD	1.224 (0.041)	1.643 (0.033)	0.532 (0.014)	**0.012** (0.003)	**0.018** (0.003)
	CC	0.829 (0.010)	1.264 (0.030)	0.413 (0.026)	**0.043** (0.006)	**0.056** (0.007)
	OC	0.256 (0.005)	0.226 (0.004)	0.020 (0.002)	**0.007** (0.004)	**0.002** (0.000)
	BC	2.084 (0.255)	3.735 (0.156)	0.167 (0.040)	**0.023** (0.008)	0.048 (0.005)
	NSPDK	0.176 (0.004)	0.172 (0.007)	0.102 (0.002)	**0.008** (0.006)	**0.017** (0.004)

cases, they perform indistinguishably. Notably, our model outperforms GraphRNN on every metric in the EGO dataset. Notice the low scores achieved by the two models on the global graph statistics (BC and NSPDK), which indicate that they are both able to learn the global structure of the graph, independently from the dataset they are trained on. To summarize the performances of the models, in Table 6.4 we report the mean rank of all models. To rank the models, we first order the metrics by increasing score, collect the position of each model in the rank, and average across the datasets. Our model and GraphRNN perform almost indistinguishably when compared across all datasets. In Figure 6.5 we show graphically how well the generated samples match the empirical distribution of graph statistics across the datasets. We only consider four metrics (DD, CC, OC, and BC), for which histograms are available, fitting them with a Kernel Density Estimator (KDE) for visual comparison. The plots show that our model and GraphRNN well approximate the test sample statistics. Finally, in Figure

TABLE 6.4: Mean ranks obtained on the evaluated qualitative metrics by the examined models over all datasets. Best performances are bolded.

Metric	ER	BA	GRU-B	GRNN	Ours
DD	4.16	4.50	3.50	1.33	**1.16**
CC	3.00	4.83	2.83	**1.16**	**1.16**
OC	4.50	4.16	3.16	**1.16**	1.33
BC	4.16	4.33	3.00	2.00	**1.33**
NSPDK	4.33	4.33	3.00	1.33	**1.50**

6.6 we show graphs drawn from our model against real graphs, on three datasets. Indeed, visual inspection confirms that generated graphs display connectivity patterns typical of real graphs, for example ego nodes in EGO-like graphs, long chains in PROTEINS-like graphs, and dense clusters in COMMUNITY-like graphs.

6.3.3 Effect of Node Ordering

In a follow-up study, we investigate if the way nodes are ordered during the graph preprocessing phase affects performances. To do so, we compare the proposed model with ablated models, where graphs are transformed into ordered edge sequences using different node orderings. Specifically, we focus on five different ablations:

- *Random*: the ordered edge sequence is constructed with a random permutation of the nodes. This strategy is expected to perform very poorly, since the model does not "see" consistent connectivity patterns during training;
- *BFS Random*: the ordered edge sequence is constructed using the strategy proposed by You et al. [You+18b]; that is, the ordered edge sequence is constructed using the BFS order, but each time a graph is picked from the training set at different epochs, the root node for the visit can be different. The strategy corresponds to train the model on every possible node permutation induced by a breadth-first search (BFS) visit of the graph. On one hand, this strategy acts as a regularizer, since at each epoch the training set changes completely. On the other hand, however, changing the node ordering at each epoch could in principle prevent our model from focusing on useful connectivity patterns, simply because they are "masked" away by different node permutations;
- *DFS Random*: the same strategy as BFS random, but using depth-first search (DFS) visit instead;
- *DFS*: a strategy equivalent to the one proposed in this work, which uses depth-first traversal instead of breadth-first;
- *SMILES*: for the two molecular datasets only, the node ordering imposed by the SMILES linearization of the graph.

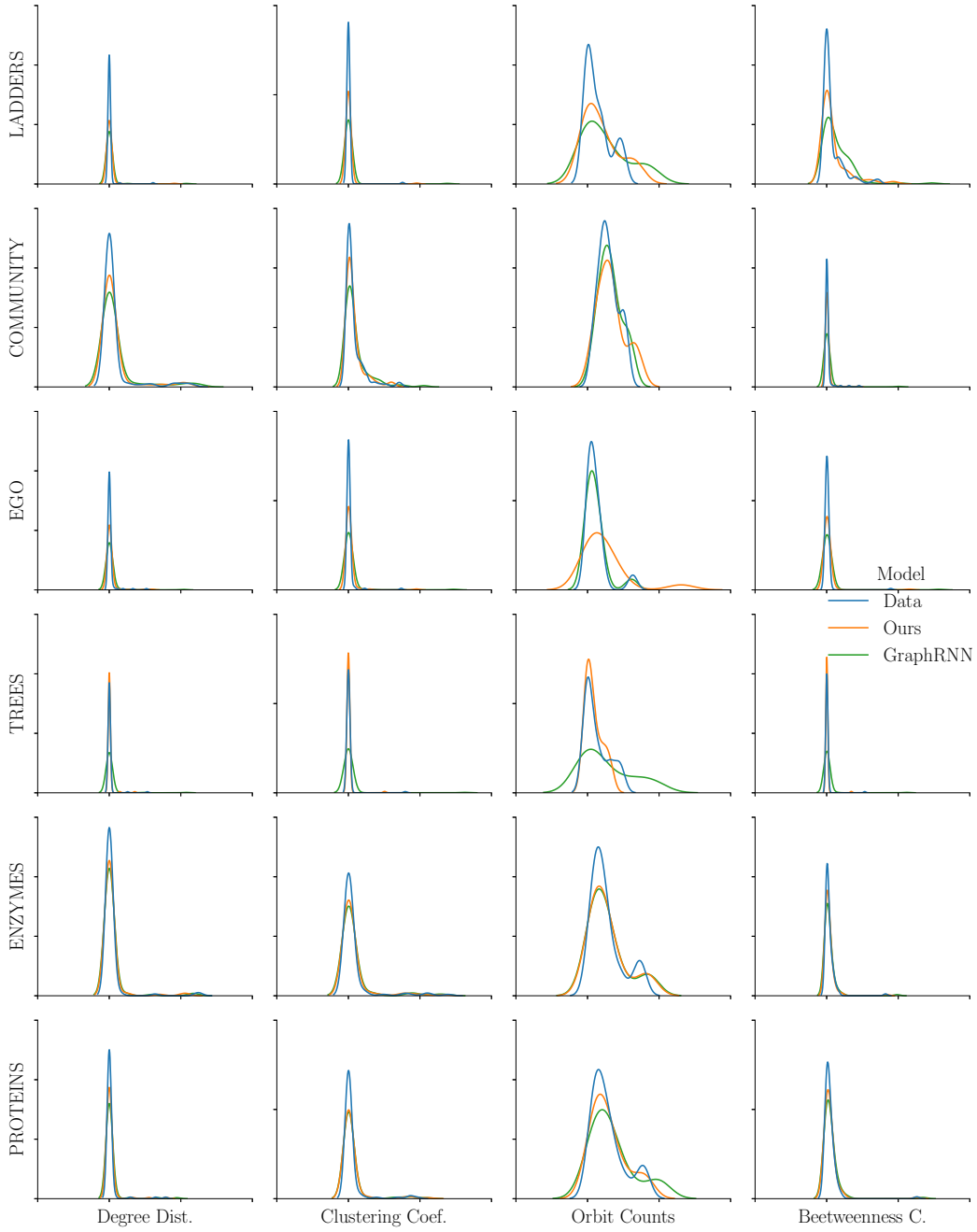


FIGURE 6.5: We plot the KDE fitted on the histograms of graph statistics, taken both from the test and generated samples. Scales are omitted since values are normalized.

The ablation experiment consists of training the models with the alternative node ordering strategies for 1000 epochs without regularization. In practice, we try to overfit the dataset on purpose. The idea is to assess whether the alternative node ordering strategies are able to memorize connectivity patterns (hence, given proper regularization, are able to generalize to unseen ones). As a further experiment, we perform a qualitative evaluation similarly to Section 6.3.2.

In Figure 6.7, we plot the loss obtained by the different ablations across all the

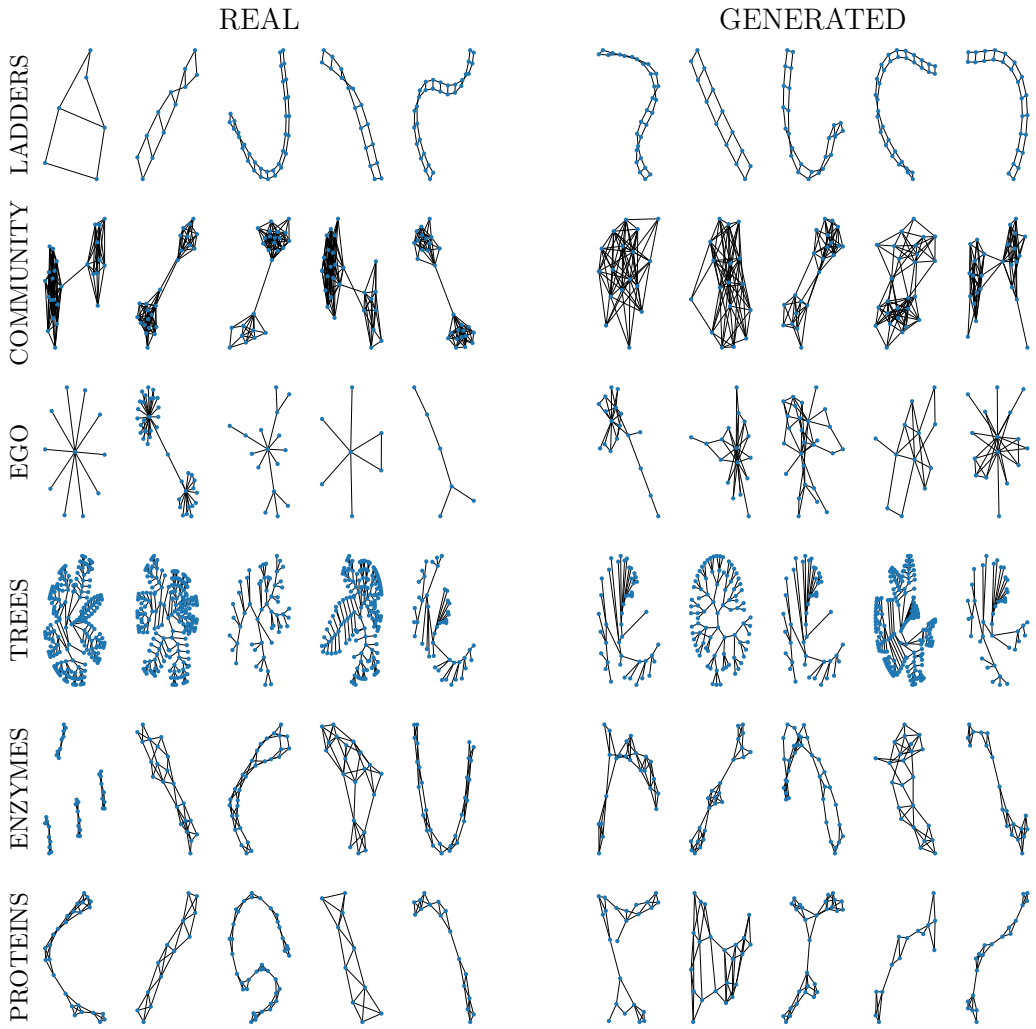


FIGURE 6.6: Five randomly chosen graphs, both from the training sample (left) and generated by our model (right), on each considered dataset (displayed by row). Notice how our generative model has picked up all the characterizing connectivity patterns.

datasets. The figure shows that the models trained with the BFS, DFS and SMILES ordering strategies are able to reach a lower loss than the random ablations, which in contrast plateau at higher loss values. This experiment supports the choice of the BFS node ordering, which couples well with the architecture of our model, and provides an effective inductive bias to learn connectivity patterns from very different graph datasets.

The results of the qualitative analysis are detailed in Table 6.5. It can be easily noticed that the proposed node ordering strategy yields superior results in every chosen metric, compared to the ablations. The only competitive node ordering strategy is DFS, which basically differs from ours in the way graph nodes are visited. However, despite its similarity to our choice, this strategy seems to underperform in some cases, for example in all the local metrics in the COMMUNITY dataset. To explain this phenomenon, we hypothesize that the choice of DFS traversal instead of BFS is less

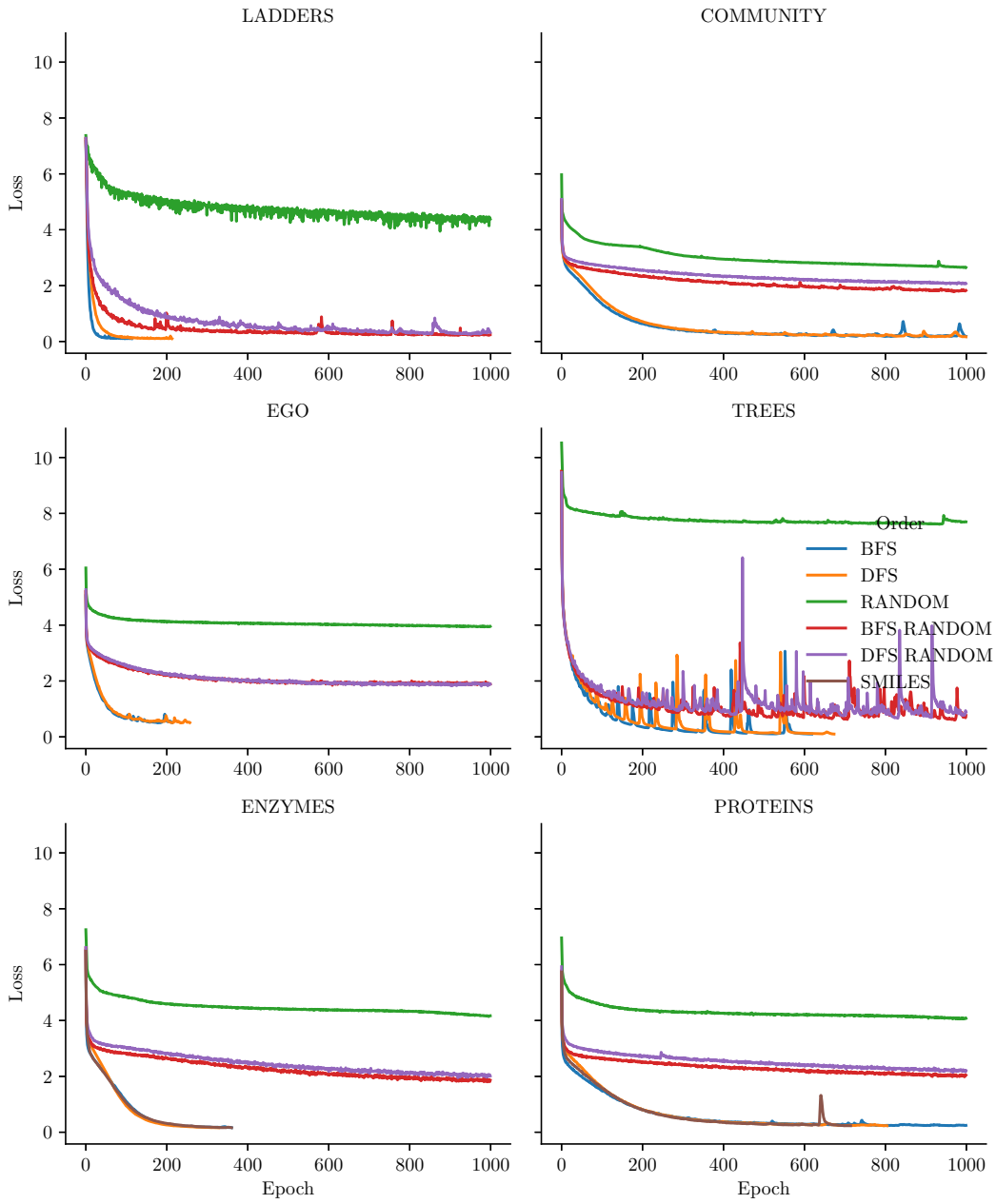


FIGURE 6.7: Plot of the loss on dataset Protein of variants of our approach trained under different node orderings. Notice how the models trained with our proposed node ordering (blue), and SMILES (orange) are able to reach a lower training loss in a smaller number of epochs, compared to variants trained with random ordering (red) and BFS random ordering (green).

suited to capture local dependencies. In conclusion, these results further extend the evidences supporting the robustness of our approach.

TABLE 6.5: We show the results of training the model with different node ordering strategies. Best performances are bolded.

Dataset	**Metric**	**Random**	**Rand. BFS**	**Rand. DFS**	**DFS**	**SMILES**	**Ours**
LADDERS	DD	0.881 (0.111)	0.162 (0.010)	0.079 (0.011)	0.073 (0.012)	-	**0.013** (0.003)
	CC	0.317 (0.019)	0.025 (0.006)	0.015 (0.007)	0.002 (0.003)	-	**0.000** (0.000)
	OC	0.136 (0.015)	0.007 (0.001)	0.003 (0.001)	0.002 (0.001)	-	**0.000** (0.000)
	BC	1.111 (0.080)	0.411 (0.053)	0.283 (0.070)	**0.202** (0.018)	-	0.762 (0.180)
	NSPDK	0.549 (0.006)	0.230 (0.032)	0.089 (0.018)	0.114 (0.016)	-	**0.056** (0.022)
COMMUNITY	DD	0.204 (0.002)	0.027 (0.001)	0.073 (0.004)	0.025 (0.002)	-	**0.013** (0.000)
	CC	11.066 (0.990)	2.858 (0.340)	0.368 (0.025)	0.471 (0.022)	-	**0.101** (0.011)
	OC	1.142 (0.020)	0.276 (0.002)	0.045 (0.002)	0.088 (0.002)	-	**0.013** (0.002)
	BC	0.876 (0.132)	1.229 (0.012)	0.232 (0.046)	0.177 (0.045)	-	**0.038** (0.005)
	NSPDK	0.062 (0.001)	0.031 (0.000)	0.024 (0.001)	0.012 (0.001)	-	**0.009** (0.000)
EGO	DD	0.099 (0.002)	0.039 (0.004)	0.052 (0.002)	**0.026** (0.002)	-	**0.029** (0.001)
	CC	0.241 (0.004)	0.301 (0.002)	0.233 (0.002)	**0.073** (0.007)	-	**0.090** (0.011)
	OC	0.109 (0.002)	0.055 (0.002)	0.039 (0.000)	**0.004** (0.001)	-	**0.002** (0.001)
	BC	0.139 (0.012)	0.141 (0.001)	0.139 (0.001)	**0.052** (0.013)	-	0.096 (0.001)
	NSPDK	0.010 (0.000)	0.011 (0.001)	0.018 (0.000)	**0.007** (0.000)	-	0.016 (0.001)
TREES	DD	0.819 (0.004)	0.320 (0.010)	0.261 (0.007)	0.143 (0.008)	-	**0.078** (0.002)
	CC	0.065 (0.001)	0.004 (0.001)	0.003 (0.000)	0.004 (0.001)	-	**0.000** (0.000)
	OC	0.028 (0.001)	0.005 (0.001)	0.004 (0.000)	0.003 (0.000)	-	**0.001** (0.000)
	BC	0.416 (0.008)	0.075 (0.002)	0.067 (0.002)	0.056 (0.003)	-	**0.056** (0.008)
	NSPDK	0.653 (0.003)	0.328 (0.004)	0.240 (0.005)	0.142 (0.009)	-	**0.042** (0.004)
ENZYMES	DD	0.499 (0.014)	0.140 (0.005)	0.135 (0.007)	0.064 (0.004)	0.077 (0.007)	**0.040** (0.002)
	CC	0.581 (0.026)	1.107 (0.218)	0.566 (0.032)	0.138 (0.009)	0.230 (0.011)	**0.128** (0.037)
	OC	0.096 (0.002)	0.179 (0.007)	0.102 (0.001)	0.019 (0.003)	0.034 (0.003)	**0.005** (0.002)
	BC	2.484 (0.038)	1.192 (0.022)	0.293 (0.029)	0.138 (0.005)	0.479 (0.088)	**0.062** (0.006)
	NSPDK	0.113 (0.002)	0.045 (0.000)	0.041 (0.001)	0.026 (0.001)	0.030 (0.002)	**0.021** (0.001)
PROTEINS	DD	0.209 (0.007)	0.160 (0.002)	0.153 (0.008)	0.092 (0.006)	0.104 (0.008)	**0.018** (0.003)
	CC	0.947 (0.025)	1.105 (0.019)	0.568 (0.034)	0.163 (0.003)	0.277 (0.007)	**0.056** (0.005)
	OC	0.200 (0.006)	0.220 (0.001)	0.120 (0.003)	0.032 (0.003)	0.067 (0.006)	**0.002** (0.000)
	BC	1.161 (0.144)	0.345 (0.015)	0.157 (0.019)	0.071 (0.003)	0.368 (0.124)	**0.048** (0.005)
	NSPDK	0.059 (0.001)	0.045 (0.001)	0.038 (0.001)	0.026 (0.001)	0.032 (0.002)	**0.017** (0.000)

Chapter 7

A Deep Generative Model for Fragment-Based Drug Discovery

The term *de novo* Drug Design (DD) refers to a plethora of methods for the production of novel chemical compounds endowed with desired pharmaceutical properties. It is an important step of the *drug discovery* pipeline, the process that goes from the identification of a biological target for which treatment is needed, to a novel drug hitting the markets. The cost of producing a new drug is now over 1 billion USD, and the average time taken to develop one is approximately 12 years [DGH16]. In this scenario, computational (or *in-silico*) methodologies are gradually substituting more traditional *in-vitro* solutions, as they allow expediting almost every aspect of the discovery process, while also being ethically more sustainable (for example, avoiding experimenting on animals). In this chapter, we focus on Deep Learning-based generative methods for *de novo* drug design.

7.1 Background

Here, we provide the necessary background to understand our contribution. We start with a brief primer on molecules and how they are represented for generative tasks. We follow by explaining the typical generative tasks in the context of drug discovery. Then, we briefly review current approaches for modeling molecule generation and how they are evaluated. Lastly, we provide a primer on chemical fragments and Fragment-Based Drug Design (FBDD), which is the main paradigm that inspired this work.

7.1.1 Molecules and their Representation

For generative tasks involving molecules, two kinds of data representations are usually employed. Below, we review both in more detail.

SMILES

One convenient representation of molecules is SMILES, which is essentially a linearization of the molecular graph as a string of ASCII characters, formed by variations of the following general algorithm:

- the hydrogen atoms are removed (they can be inferred back using *valency* rules);
- the molecular graph is transformed into a spanning tree by removing cycles (disconnecting rings);
- the disconnected bonds are marked with unique integers, to allow their reconstruction;
- the spanning tree is visited in depth-first order; each time a new atom is visited, the corresponding atomic symbol (and a symbol indicating its chemical bond, if different from a single bond) is added to the SMILES string. Tree branches are marked by parentheses.

Even though SMILES strings are essentially sequences of characters, they are generally considered structured representations of the molecular graph, since they are constructed upon its traversal. Unfortunately, SMILES strings are not unique, i.e. different SMILES strings can be associated with the same molecule by changing the root node of the spanning tree or starting the traversal from a different node. More precisely, molecules with n atoms can be represented with at least n different SMILES strings. To mitigate this issue, most chemical packages that implement SMILES parsers include canonicalization algorithms to ensure that uniqueness of the SMILES strings [WWW89] (except a few degenerate cases). Despite these issues, the SMILES encoding is widely adopted in Cheminformatics, for example to index molecules in large databases.

Molecular Graph

A more expressive and natural representation of the molecule is given by the molecular graph, also called Lewis structure. In a molecular graph, atoms are nodes and bonds are links between them. The molecular graph can carry additional information through the features attached to the nodes and edges of the associated molecular graph. Typical features added to the atoms are:

- one-hot encoding of the atom type;
- number of Hydrogen atoms attached to a given atom;
- whether the atom is part of a ring or not;
- charge of the atom, *i.e.* its number of protons and electrons;
- *stereochemistry* and *chirality* information, *i.e.* its 3-D structure. A typical example are coordinates representing the position of the atom in 3-D space.

As regards molecular bonds, the features added are:

- a one-hot encoding of the bond type;
- whether the bond is part of a ring or not;
- stereochemistry information.

7.1.2 Generative Tasks for Drug Design

We identify three, different but connected, generative tasks in the context of *de novo* drug design where DGMs can be used:

- *molecular generation* corresponds to the usual unconditional generation task. Given a dataset $\mathbb{D} = \{\mathtt{G}_{(i)}\}_{i=1}^{n}$, where $\mathtt{G}_{(i)}$ are attributed molecular graphs, the objective is to approximate $p(\mathtt{G})$, their underlying probability, or a mechanism to sample from it;

- *molecular optimization* concerns optimizing some complex chemical property of compounds in a dataset of molecular graphs $\mathbb{D}$, which might be given or generated. In practice, a molecular optimization task often complements one of molecular generation;

- *molecular translation* is in some sense a supervised task of conditional generation. Here, the dataset is composed of pairs of attributed molecular graphs $\mathbb{D} = \{(\mathtt{G}_{(i)}, \mathtt{H}_{(i)})\}_{i=1}^{n}$, where the targets $\mathtt{H}_{(i)}$ are structurally similar to the inputs $\mathtt{G}_{(i)}$, but with enhanced pharmaceutical properties. The task is to learn their relationship such that, when given an unseen compound $\mathtt{G}'$, the model generates a similar and chemically enhanced compound $\mathtt{H}'$.

7.1.3 Deep Generative Models of Molecules

Broadly speaking, there are two major lines of research as regards *de novo* drug design with DGMs, which are distinguished by the type of decoder used to generate a molecule:

- *SMILES-based* approaches generate molecules through their SMILES strings with autoregressive decoders. In this case, the architecture learns a *language model* of SMILES strings, *i.e.* a model that predicts the next SMILES character, given the previous characters;

- *graph-based* approaches learn to generate the molecular graph directly (with one-shot decoders) or incrementally (with autoregressive decoders);

- *substructure-based* approaches learn to generate the molecular graph compositionally, by combining smaller substructures together.

Below, we provide a brief overview of the literature in DGMs models of molecules based on this taxonomy.

SMILES-based Approaches

A first class of SMILES-based approaches model the generation with a simple RNN decoder [Seg+17; BT17]. These models do not use a structured latent space, and hence are used only for molecular generation. In some cases, reinforcement learning techniques are adopted to perform molecular optimization [Oli+17; Nei+18]: in

practice, the model is first pretrained autoregressively with MLE; then, the molecules are optimized with reinforcement learning, by giving higher rewards to molecules that respect validity constraints, or whose properties are above a desired threshold. Historically, one of the first SMILES-based generative models using a latent space is that of Gómez-Bombarelli et al. [Góm+18], which we term ChemVAE. Basically, the model is a seq2seq language model for SMILES strings, where the latent space is shaped through a VAE. However, with respect to pure RNN approaches, the ChemVAE has been shown to produce a fairly high number of invalid molecules. A first improvement has been proposed by Kusner et al. [KPH17], which uses a rule-based generative model called GrammarVAE, which learns to generate productions of the SMILES grammar which yield syntactically valid molecules. Subsequently, Dai et al. [DTD+18] proposed a Syntax-Directed (SD)-VAE, which uses a SMILES attributed grammar to enforce semantic constraints to the generation. These models are adapted to perform molecular optimization by jointly training a property predictor on the latent space; then, starting from a random point, a novel molecule with optimized properties is found in latent space using Bayesian optimization. Finally, some models use *transfer learning* to optimize molecular properties [Mar+20; Mer+18]. In this case, the models are pretrained on a large dataset of molecules, and fine-tuned on a smaller dataset with optimized molecules.

Graph-based Approaches

Among graph-based approaches that generate the molecule in one shot, one of the first attempts was that of Simonovsky et al. [SK18]. The model is a VAE, whose decoder generates three matrices: a probabilistic adjacency matrix, a matrix of node features (where the features are the atom types), and a matrix of edge features (where the features are the bond types). The sampled graph is aligned with the one in input with approximate graph matching. The approach of De Cao et al. [DK18] uses a GAN, whose generator produces one probabilistic adjacency matrix for each bond type, and a node feature matrix, which are sampled to produce an actual graph. The graph is then used to train the discriminator, and also passed to a reward network that is used to optimize molecular properties with a reinforcement learning objective. Both the discriminator and the reward network are implemented as DGNs. Among autoregressive graph decoders, one approach is that of Popova et al. [Pop+19], which extends GraphRNN [You+18b] to work with molecular generation (*i.e.* to output labeled graphs). Molecular optimization is achieved using a policy gradient optimization objective. The Graph Convolutional Policy Network (GCPN) by You et al. [You+18a] formulates the molecule generation as a Markov Decision Process, where actions consist in adding nodes to an existing graph. The Constrained Graph VAE by Liu et al. [LAB+18] uses a different approach to generate a molecule: first, a set of node representations is sampled from latent space; then, the algorithm picks one node at a time, samples its atom type, and connects it to the remaining nodes, sampling the bond types. Optimization is achieved with gradient ascent (on the property value) in latent

space. A similar generative approach is used in the work by Samanta et al. [Sam+19], whilst optimization is achieved using Bayesian optimization as in Gómez-Bombarelli et al. [Góm+18].

Substructure-based Approaches

One of the first work that used substructure generation is that of Jin et al. [JBJ18], called Junction Tree (JT)-VAE. In the work, a junction tree of the molecule is built by considering some substructures such as rings as a single component. The model learns to generate junction trees, which guide the generation of the molecular graph. Thus, the model generates valid molecules by design. Optimization in latent space is achieved with Bayesian optimization. The model has been extended by the same authors in [Jin+19] to a hierarchical model, where different levels of coarseness of the molecular graph are used to connect the different substructures. This work is also the first to introduce the molecular translation problem. Similarly, Fu et al. [FXS20] proposed CoRe, a JT-VAE enhanced with a Copy and Refine mechanism that learns how to copy some substructures from the input molecule, rather than sample them. Molecule Chef, by Bradshaw et al. [Bra+19], operates under a different paradigm: the architecture is still a VAE (with Wasserstrain distance regularization instead of KLD), but trained to generate a sequence of reactants, rather than proper substructures of the desired molecule. A novel molecule is then synthesized by mapping the predicted reactants to a desired compound through a reaction predictor. The model is also extended to allow retro-synthesis, *i.e.* mapping a molecule to the most probably reactants that generated it. The model has later been extended by Bradshaw et al. [Bra+20], where the procedure has been modified to synthesize molecule through DAGs in a multi-step fashion, similarly to the actual procedure used in laboratories. Finally, the model by Ståhl et al. [Stå+19] works on chemical fragments, similarly to the contribution we present in Section 7.2. Specifically, they encode each fragment using a balanced binary tree, where similar fragments are placed in nearby leaves. The model is then trained to learn to replace fragments in an input molecule, to produce an optimized one.

7.1.4 The Evaluation of Deep Generative Models of Molecules

There are several metrics under which the molecular generators can be evaluated. Ideally, generated molecules should be at the same time:

- structurally "different" enough from training molecules to span the largest possible chemical subspace;
- structurally "close" enough to possess similar interesting chemical properties.

Thus, besides chemical validity, novelty and uniqueness, the *diversity* of the generated samples plays a major role. One measure of diversity requires to evaluate the average distance between all the possible pairs of generated molecules. This is called *internal* diversity. Conversely, *external* diversity measures the distance of

each molecule in the generated sample against its nearest neighbor molecule in the training sample. The distance function usually employed to calculate diversity is the *Tanimoto distance*, which is a bit-wise distance function defined over the molecular fingerprints associated with a molecule. Another, more recent, diversity metric is the Frechét ChemNet Distance [Pre+18], which evaluates the distance between the training and generated samples by first feeding them to a pre-trained Deep Neural Network (ChemNet [Goh+17]) for property prediction, and then comparing several statistics computed on the respective activations at the penultimate layer. For molecular generation tasks, other useful metrics to evaluate the generative performance are based on comparing the distributions of chemical properties between the training sample and the generated sample, as shown in Section 3.6.4. Typical properties evaluated to this aim are number of atoms, number of bonds, and number of rings distribution of the generated molecules (compared to the training sample). To evaluate performances on the optimization tasks, one intuitive metric to compute is the *improvement*, which measures the average increase (or decrease) of the property obtained in the generated sample, with respect to the training sample. In molecular translation tasks, often *success* is measured. It is the ratio of molecules in the generated sample that were correctly "translated", meaning that their molecular properties are above some predefined threshold. Notice that success and improvement should be evaluated after having cleaned the generated sample from duplicates, to avoid biased results. In general, improvement and success cannot be calculated directly, as the ground truth value of the property may be not known or hardly computable for the generated molecule. This is often the case, for example, when the target is biological activity. In these cases, it is often useful to compare against an already trained predictor, to obtain an approximation of the ground truth property.

7.1.5 A Primer on Fragments

Fragments are very-small-weight compounds, typically composed of < 20 non-hydrogen atoms. Small size has several advantages: firstly, they are easier to manipulate chemically than larger fragments. Secondly, the chemical space of fragments is narrower than, for example, the one of drug-like molecules typically generated from other DD approaches such as HTS. Thus, it is easier to explore and characterize. Thirdly, the small size makes fragments weakly interact with a broader spectrum of target proteins than larger compounds (higher molecular complexity translates into stronger interaction, albeit not necessarily beneficial). A typical FBDD experiment begins with the identification of a suitable collection of fragments, from which a subset with desired interactions with the target (hits) is identified. Subsequently, fragments are optimized into higher affinity compounds that become the starting points (leads) for subsequent drug discovery phases. Optimization is commonly carried out according to three different strategies: (a) linking, which optimizes a given fragment by connecting it with another fragment; (b) growing, where the fragment is functionally and structurally enriched to optimize binding site occupation; (c) merging, which involves combining

the structure of two overlapping fragments into a new one with increased affinity. Since its inception in 1996, FBDD accounts for two clinically approved drugs, and more than thirty undergoing clinical trials at various stages [DR17].

7.2 Methods

Here, we present the methodologies adopted in our study. Specifically, how the molecules have been broken into sequences of fragments, how these fragments have been embedded in vectorial form, and the architecture used during training and generation. Finally, we describe Low-Frequency Masking, the technique employed to boost the number of unique molecules produced by the model. We start off with a set $\mathcal{G} = \{\mathtt{G}_{(i)}\}_{i=1}^{n}$ of molecular graphs, where each node is labeled with its atom type, and each edge is labeled with its bond type.

7.2.1 Molecule Fragmentation

Given a dataset of molecules, the first step of our approach entails breaking them into an ordered sequence of fragments. To do so, we leverage the Breaking of Retrosynthetically Interesting Chemical Substructures (BRICS) algorithm [DWZ+08], which breaks strategic bonds in a molecule that match a set of chemical reactions. "Dummy" atoms (with atomic number 0) are attached to each end of the cleavage sites, marking the position where two fragments can be joined together. BRICS cleavage rules are designed to retain molecular components with valuable structural and functional content, e.g. aromatic rings and side-chains, breaking only single bonds that connect among them. Our fragmentation algorithm works by scanning atoms in the order imposed by the SMILES encoding. As soon as a breakable bond (according to the BRICS rules) is encountered during the scan, the molecule is broken in two at that bond, applying a matching chemical reaction. After the cleavage, we collect the leftmost fragment, and repeat the process on the rightmost fragment in a recursive fashion. Note that fragment extraction is ordered from left to right according to the SMILES representation; this makes the process fully reversible, i.e. it is possible to reconstruct the original molecule from a sequence of fragments. In Figure 7.1, we show a practical example of the fragmentation process on the molecule of aspirin. After the fragmentation phase, the set of molecular graphs $\mathcal{G}$ is transformed in a set of fragment sequences $\mathbb{D} = \{s_{(i)} = (\mathtt{f}_{[1]}, \mathtt{f}_{[2]}, \ldots, \mathtt{f}_{[|s_{(i)}|]}) \mid \mathtt{G}_{(i)} \in \mathcal{G}\}_{i=1}^{n}$. The unique fragments found during the fragmentation process are stored in a *vocabulary* V as one-hot vectors. We indicate with $V(\mathtt{f}_{[i]}) \in \mathbb{R}^{|V|}$ the one-hot vector in the vocabulary corresponding to fragment $\mathtt{f}_{[i]}$, and with $|V|$ the size of the vocabulary.

7.2.2 Skip-Gram Embedding of Fragments

In this phase, we transform the one-hot vectors specifying the fragments into continuous vectors enriched with context semantics. In analogy with the work of Bowman

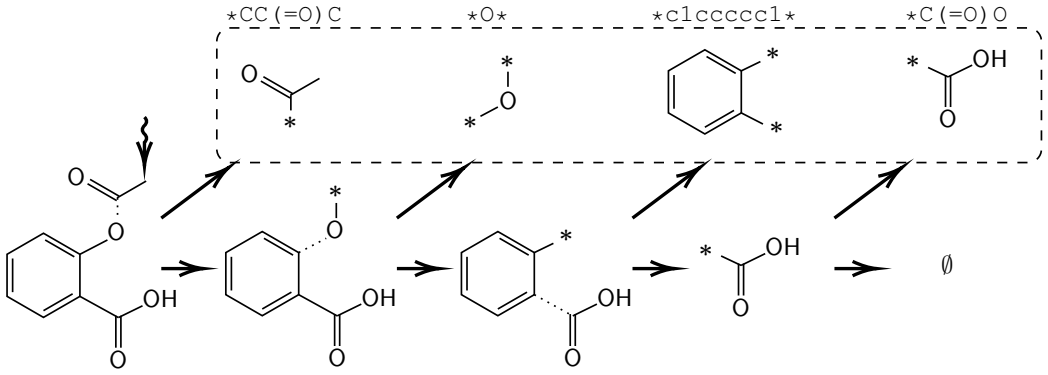


FIGURE 7.1: Fragmentation of the aspirin molecule (SMILES string `CC(=O)OC1=CC=CC=C1C(=O)O`). In the bottom row, the wavy arrow indicates the first atom according to the SMILES ordering, and dashed bonds are broken according to the BRICS rules. At each iteration, the leftmost fragment is stripped out of the molecule until the remainder cannot be broken further. The derived fragment sequence is displayed in the top row, delimited by a dashed box.

et al. [Bow+16], we view a sequence of fragments as a "sentence"; therefore, each fragment is a "word". Given a sequence $s = (\mathtt{f}_{[1]}, \mathtt{f}_{[2]}, \ldots, \mathtt{f}_{[|s|]})$ of fragments, this objective corresponds to optimizing the following Skipgram loss function [LM14]:

$$\mathcal{L}((\boldsymbol{E}, \boldsymbol{E}_{\text{out}}), s) = -\sum_{i=1}^{|s|} \sum_{-w \le j \le w} \log p_{\boldsymbol{E}, \boldsymbol{E}_{\text{out}}}(\mathtt{f}_{[i+j]} | \mathtt{f}_{[i]}), \;\; j \neq 0,$$

where w is called context window, $\mathtt{f}_{[i]}$ is an input fragment, and $\mathtt{f}_{[i+j]}$ are fragments that occur in the context of $\mathtt{f}_{[i]}$. We implement $p_{\boldsymbol{E}, \boldsymbol{E}_{\text{out}}}$ as the following neural network:

$$\begin{aligned} \boldsymbol{f}_{[i]} &= \boldsymbol{E}\, V(\mathtt{f}_{[i]}) \\ \boldsymbol{o}_{[i]} &= \text{softmax}(\boldsymbol{E}_{\text{out}} \boldsymbol{f}_{[i]}), \end{aligned}$$

where $\boldsymbol{E} \in \mathbb{R}^{d \times |V|}$ is an embedding matrix and $\boldsymbol{E}_{\text{out}} \in \mathbb{R}^{d \times |V|}$ is an output matrix. The model is pretrained with negative sampling [LM14]. After pretraining, the columns of the embedding matrix $\boldsymbol{E}$ contain d-dimensional embeddings of each fragment in the vocabulary. As a result, fragment embeddings are endowed with context semantics: precisely, embeddings that are close in a sequence of fragments are represented by nearby vectors in embedding space.

7.2.3 Model

Here, we describe the proposed model, detailing how it is trained, and how it can be used for inference and molecular generation.

Training

The model architecture is a standard VAE seq2seq model for graph generation, composed of an encoder $q_{\boldsymbol{\psi}}(\boldsymbol{z} \mid s)$ and an autoregressive graph decoder $p_{\boldsymbol{\phi}}(s \mid \boldsymbol{z})$, where $s = (\mathtt{f}_{[1]}, \mathtt{f}_{[2]}, \ldots, \mathtt{f}_{[|s|]})$ is a generic input fragment sequence. The encoder is a RNN that operates at fragment level and computes the following:

$$\begin{aligned} \boldsymbol{f}_{[i]} &= \boldsymbol{E}\, V(\mathtt{f}_{[i]}) \\ \boldsymbol{h}_{[i]} &= \mathrm{GRU}_{\boldsymbol{\psi}}\left(\boldsymbol{h}_{[i-1]}, \boldsymbol{f}_{[i]}\right), \end{aligned}$$

where $\boldsymbol{E}$ is the pretrained embedding matrix, and $\boldsymbol{h}_{[0]} = \mathbf{0}$ as usual. The final hidden state $\boldsymbol{h}_{[|s|]} \in \mathbb{R}^h$ is used as a representation of the whole fragment sequence. The sequence representation is then mapped to two vectors as follows:

$$\begin{aligned} \boldsymbol{\mu} &= \boldsymbol{W}_{\boldsymbol{\psi}_\mu} \boldsymbol{h}_{[|s|]} \\ \boldsymbol{\sigma}^2 &= \boldsymbol{W}_{\boldsymbol{\psi}_\sigma} \boldsymbol{h}_{[|s|]}, \end{aligned}$$

$\boldsymbol{\mu} \in \mathbb{R}^h$ and $\boldsymbol{\sigma}^2 \in \mathbb{R}^h$ represent the mean and variance of the encoding distribution, and $\boldsymbol{W}_{\boldsymbol{\psi}_\mu}, \boldsymbol{W}_{\boldsymbol{\psi}_\sigma} \in \mathbb{R}^h$ are weight matrices. Subsequently, a latent sample from the encoding distribution is drawn with reparameterization as $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\varepsilon}\boldsymbol{\sigma}^2$, with $\boldsymbol{\varepsilon} \in \mathbb{R}^h \sim \mathtt{N}(\mathbf{0}, \boldsymbol{I})$, where $\boldsymbol{I} \in \mathbb{R}^{h \times h}$ is a unit covariance matrix. The encoder is trained to minimize the KLD between the distribution parameterized by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ and a standard Gaussian prior:

$$\mathcal{L}(\boldsymbol{\psi}, s) = \mathrm{KLD}(q_{\boldsymbol{\psi}}(\boldsymbol{z} \mid s) \parallel \mathtt{N}(\mathbf{0}, \boldsymbol{I})).$$

The decoder is initialized with the latent vector, *i.e.* by setting $\boldsymbol{h'}_{[0]} = \boldsymbol{z}$. Specifically, the decoder is a RNN that computes the following:

$$\begin{aligned} \boldsymbol{f}_{[i-1]} &= \boldsymbol{E}\, V(\mathtt{f}_{[i-1]}) \\ \boldsymbol{h'}_{[i]} &= \mathrm{GRU}_{\boldsymbol{\phi}}(\boldsymbol{h}_{[i-1]}, \boldsymbol{f}_{[i-1]}) \\ \boldsymbol{o'}_{[i]} &= \mathrm{softmax}\left(\boldsymbol{V}_{\boldsymbol{\phi}} \boldsymbol{h'}_{[i]} + \boldsymbol{b'}_{\boldsymbol{\phi}}\right), \end{aligned}$$

where $\boldsymbol{f}_{[0]} = \boldsymbol{E}\, V(\langle \mathsf{S} \rangle)$, $\boldsymbol{E}$ is the shared embedding matrix, and $\boldsymbol{V} \in \mathbb{R}^{|V| \times h}$, $\boldsymbol{b'}_{\boldsymbol{\phi}} \in \mathbb{R}^{|V|}$ are the parameters of the softmax output layer. The vector $\boldsymbol{o'}_{[i]} \in \mathbb{R}^{|V|}$ is the output distribution over all possible fragments in the vocabulary. Each output of the network is compared with the ground truth fragment $\mathtt{f}_{[i]}$ with the CE function as follows:

$$\mathcal{L}(\boldsymbol{\phi}, s) = \frac{1}{|s|+1} \sum_{i=1}^{|s|+1} -\log p_{\boldsymbol{\phi}}(\mathtt{f}_{[i]} \mid \mathtt{f}_{[<i]}) = \frac{1}{|s|+1} \sum_{i=1}^{|s|+1} \mathrm{CE}\left(V(\mathtt{f}_{[i]}), \boldsymbol{o'}_{[i]}\right),$$

where +1 is added to account for the end of sequence token. The whole model is trained with MLE to minimize the following objective function:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{s \in \mathbb{D}} \mathcal{L}(\boldsymbol{\psi}, s) + \mathcal{L}(\boldsymbol{\phi}, s),$$

where $\boldsymbol{\theta} = (\boldsymbol{\psi}, \boldsymbol{\phi})$ as usual. The architecture is shown visually in Figure 7.2.

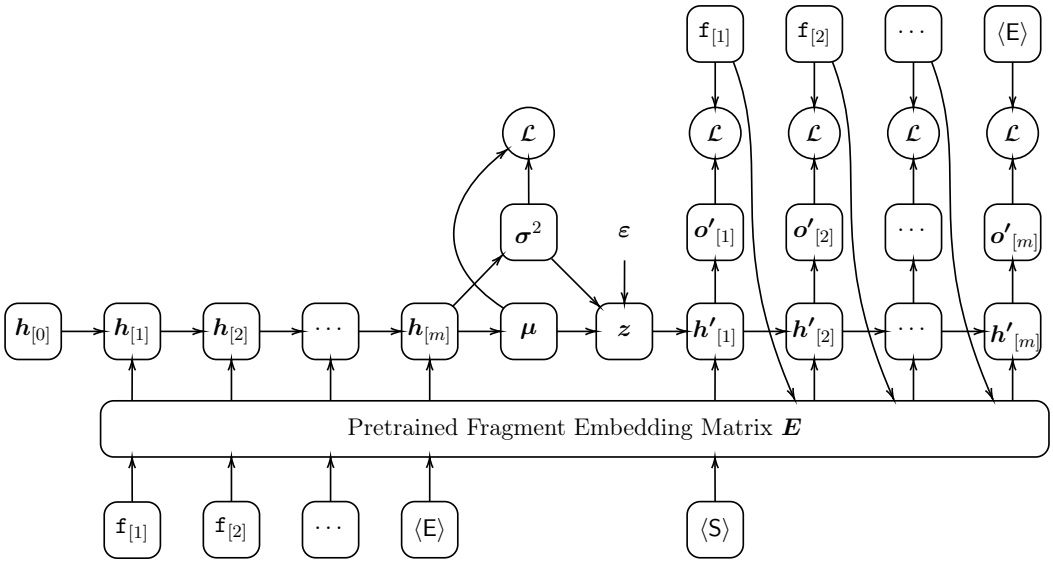


FIGURE 7.2: The architecture of the proposed generative model of fragment sequences.

Generation

Generation starts by sampling a latent point $\boldsymbol{z} \in \mathbb{R}^h$ from the standard Gaussian $\mathbb{N}(\mathbf{0}, \boldsymbol{I})$, and setting $\boldsymbol{z} = \boldsymbol{h}_{[0]}$ to initialize the decoder. The initial hidden state is fed to the network together with the starting token $V(\langle \mathsf{S} \rangle)$. At each step, the network produces a conditional output distribution on every possible word in the vocabulary, from which a fragment $\hat{\mathtt{f}}_{[i]}$ is taken using greedy sampling. The sampled token then becomes the input of the next decoding step, together with the updated hidden state. The process is repeated until the end of sequence token $\langle \mathsf{E} \rangle$ is sampled, at which point the generation is interrupted. The fragments of the sampled sequence $(\hat{\mathtt{f}}_{[1]}, \hat{\mathtt{f}}_{[2]}, \ldots, \langle \mathsf{E} \rangle)$ are then joined together in the same order to form a novel molecule. Figure 7.3 shows the process visually.

7.2.4 Low-Frequency Masking

To foster molecule diversity, we start from the observation that the distribution of fragments in the data can be roughly approximated by a power law distribution. In fact, usually few fragments occur with very high frequency, while the majority of fragments occur rarely. Hence, infrequent fragments are unlikely to be sampled

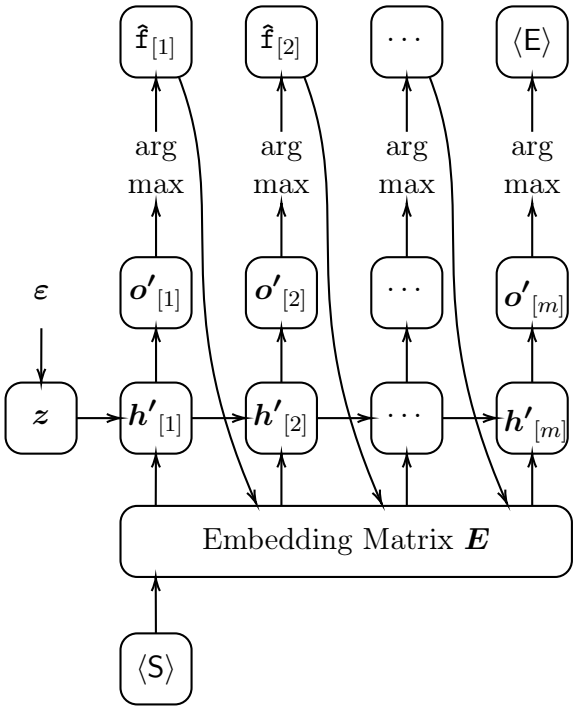


FIGURE 7.3: Fragment sequence generation using the proposed model.

during generation. To counter this issue, we develop a strategy which we term Low-Frequency Masking (LFM). During training, we mask fragments with frequency below a certain threshold k with a token composed of its frequency and the number of attachment points. As an example, suppose that fragment `*Nc1ccc(O*)cc1` occurs 5 times in the dataset, and the threshold is $k = 10$. Thus, this fragment is masked with the token `5_2`, where 5 denotes its frequency, and 2 denotes the number of attachment points. Similarly, fragment `*C(=O)N1CCN(Cc2ccccc2)CC1` with frequency of 3 is masked with the token `3_1`. In contrast, fragment `*c1ccccc1OC` with a frequency of 200 is left unmasked, since its frequency is above the threshold. A reverse mapping from the masking tokens to the masked fragments is kept. During sampling, whenever a masking token is sampled, we replace it with a fragment sampled with uniform probability from the corresponding set of masked fragments. This strategy serves a double purpose. Firstly, it greatly reduces vocabulary size during training, speeding up the computations. Secondly, it fosters molecule diversity by indirectly boosting the probability of infrequent fragments, and injecting more randomness in the sampling process at the same time. From another point of view, LFM forces the model to generate molecules mostly composed of very frequent fragments, but with infrequent substructures that may vary uniformly from molecule to molecule.

7.3 Experiments

Following, we review our experimental setup, namely how the experiments are conceived, which datasets and evaluation metrics are used, which baselines we compare

to, as well as details about the hyper-parameters of our model. In our experiments, we try to provide an empirical answer to the following questions:

- Q1: is our fragment-based language model able to increase validity rates?
- Q2: is our LFM strategy beneficial to increase uniqueness rates?

To answer the first question, we compare our model against character-based baselines, which generate molecules atom by atom. As regards the second question, we perform an ablation study of performances with and without LFM. We also compare against graph-based approaches, to assess performances in relation to models that use more expressive molecule representations.

7.3.1 Data

We experiment on the ZINC dataset [IS05], consisting of $\approx$ 250k drug-like compounds. ZINC is a common benchmark for the generative task; as such, it is used to compare against several baselines. To assess the impact of LFM further, in our ablation study we also test our model variants in the PubChem BioAssay (PCBA) dataset [GSZ+16], which comprises $\approx$ 440k small molecules. Dataset statistics are presented in Table 7.1. We applied some common preprocessing steps before training. In the PCBA dataset, we found 10822 duplicate or invalid molecules, which were removed. During an initial preprocessing phase, we discarded molecules composed of < 2 fragments. After the preprocessing, our training samples were 227946 (ZINC) and 383790 (PCBA). For completeness, we report that we tried to test our model on the QM9 dataset [RDR+14] as well, but found out that approximately 70% of its molecules are composed of a single fragment, making assessment poorly informative due to the small sample size.

Table 7.1: Dataset statistics.

	ZINC	**PCBA**
Total number of molecules	249455	437929
Molecules with no. fragments ≥ 2	227946	383790
Mean number of fragments	2.24±0.45	2.25±0.48
Vocabulary size	168537	199835
Vocabulary size (LFM)	21085	35949
Average number of atoms	23.52±4.29	26.78±6.76
Average number of bonds	25.31±5.07	28.98±7.44
Average number of rings	2.75±1.00	3.16±1.05

7.3.2 Performance Metrics

Following the standards to evaluate molecular generators, we compare our model with the baselines quantitatively, measuring validity, novelty and uniqueness. Validity

is assessed by checking the SMILES of the generated graph through the SMILES validator of the `rdkit` library in Python [Lan13]. To assess the quality of our samples, we compare the distributions of number of atoms, number of bonds and number of rings between the sampled and generated molecules. Moreover, we compare the distributions of the following molecular properties:

- octanol/water Partition coefficient (logP), which measures solubility;
- Quantitative Estimate of Drug-likeness [BPB+12] (QED), which measures drug-likeness;
- Synthetic Accessibility Score [ES09] (SAS), which measures ease of synthesis.

In both cases, we remove duplicates before the valuation, to unbias the results.

Baselines

We compare to baselines found in the literature, representing both SMILES and graph-based generative models described in Section 7.1.3. As regards SMILES-based approaches, we consider ChemVAE [Góm+18], GrammarVAE [KPH17] and SD-VAE [DTD+18], whereas as regards graph-based models, we compare against GraphVAE [SK18], CGVAE [LAB+18] and NeVAE [Sam+19].

7.3.3 Hyper-Parameters

We evaluate our model using the same hyper-parameters for both variants, in order to isolate the effect of our contribution from improvements due to hyper-parameter tuning. The embedding dimension is set to 64, the number of recurrent layers to 2, the number of GRU units per layer to 128 and the latent space size to 100. We use the Adam optimizer with an initial learning rate of 0.00001, annealed every epoch by a multiplicative factor of 0.9, a batch size of 128, and a dropout rate of 0.3 applied to the recurrent layers to prevent overfitting. The model is trained for 4 epochs: after that, we found empirically that the model started to severely overfit the training set. We use $k = 10$ as LFM threshold. The stopping criteria for training is the following: after each epoch, we sample 1000 molecules and measure validity, novelty and uniqueness rates of the sample, stopping whenever the uniqueness rate starts to drop (we found out empirically that samples were stable in terms of validity and novelty rates). After training, we sample 20k molecules for evaluation. We publicly release code and samples for reproducibility[1]. Baseline results are taken from literature[2].

[1] `https://github.com/marcopodda/fragment-based-dgm`
[2] We found no results in the literature for the PCBA dataset.

7.4 Results

The main results of our experiments are summarized in Table 7.2, and provide the answers to the experimental questions posed in Section 7.3. As regards Q1, we observe that our model achieves perfect validity scores in the ZINC dataset, greatly outperforming LM-based models and performing on par with the state of the art. This is true also as regards the PCBA dataset. Since both our variants improve over the LM-based competitors, it is safe to argue that our fragment-based approach can effectively increase validity rates. As regards Q2, we observe an improvement in uniqueness by both our variants, with respect to the LM-based competitors. However, the improvement is noticeably higher whenever the LFM strategy is employed. In the PCBA dataset, this trend is even more apparent. Compared to graph-based models, we see how the model with LFM is now competitive with the state of the art. Lastly, we notice that using LFM yields a small improvement in novelty with respect to the vanilla variant. In Figure 7.4 (for the ZINC dataset) and Figure 7.5 (as

TABLE 7.2: Scores obtained by our model against SMILES-based and graph-based baselines. LFM indicates that the model has been trained with Low-Frequency Masking. Performances of our LFM variant are shown in bold.

Model	Model Family	Dataset	Valid	Novel	Unique
ChemVAE	SMILES	ZINC	0.170	0.980	0.310
GrammarVAE	SMILES	ZINC	0.310	1.000	0.108
SD-VAE	SMILES	ZINC	0.435	-	-
GraphVAE	Graph	ZINC	0.140	1.000	0.316
CGVAE	Graph	ZINC	1.000	1.000	0.998
NeVAE	Graph	ZINC	1.000	0.999	1.000
Ours	SMILES	ZINC	1.000	0.992	0.460
Ours (LFM)	SMILES	ZINC	**1.000**	**0.995**	**0.998**
Ours	SMILES	PCBA	1.000	0.981	0.108
Ours (LFM)	SMILES	PCBA	**1.000**	**0.991**	**0.972**

regards the PCBA dataset), our samples against training compounds are compared, as regards the distribution of the three structural features, and molecular properties listed above. Notice that even without the help of an explicit supervision, generated molecules are qualitatively similar to the training data. Finally, Figure 7.6 shows two random samples of 30 molecules taken from the ZINC dataset and generated by our model for visual comparison.

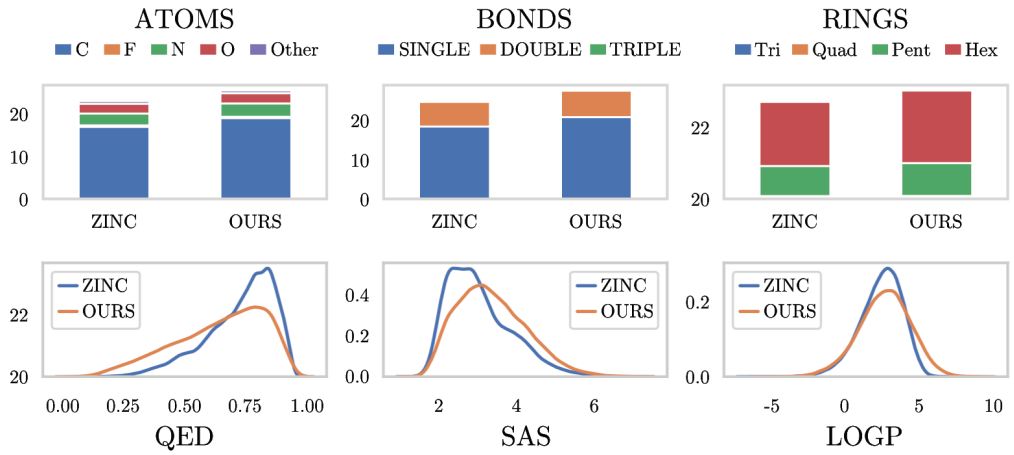


FIGURE 7.4: Comparison of structural and molecular properties between the training and generated samples on the ZINC dataset.

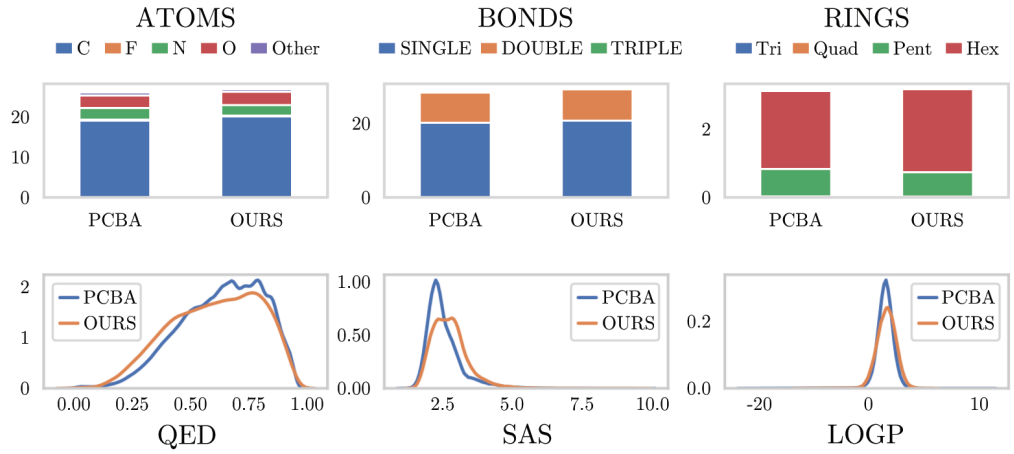
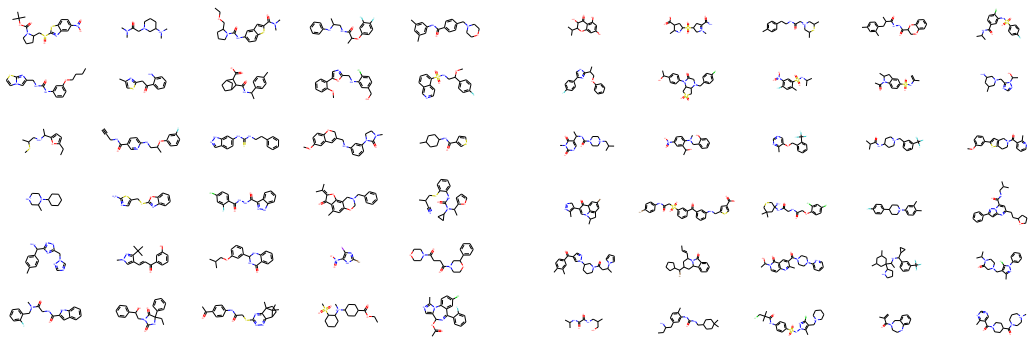


FIGURE 7.5: Comparison of structural and molecular properties between the training and generated samples on the PCBA dataset.



(A) ZINC (B) Generated

FIGURE 7.6: A random sample of 30 molecules taken from the ZINC dataset (A) and generated by our model (B).

Part IV

Conclusions

Chapter 8

Conclusions and Future Works

We live in a world where the availability of rich, relational data increases at an astonishing rate. In this context, Deep Learning methodologies that learn from structured data offer an arguably unique combination of flexibility to handle heterogeneous data types, generality in terms of different learning paradigms covered, and powerful universal approximation capabilities. These characteristics match those of the life sciences domains, where data are generally relational, and learning problems arise from very complex and diverse processes. The main objective of this thesis has been to show this compatibility in a selection of real-world applications. Our study has concerned two somewhat different but equally relevant sub-domains of the life sciences: that of computational biology, where the focus has been on predictive learning of dynamical properties of biochemical pathways; and that of computational chemistry, where the focus has been on generative learning of molecules for *de novo* drug design. In the following, we review our contributions in retrospect and highlight future research directions for each of them.

The first contribution of this thesis has been an extensive and fair evaluation of Deep Graph Network models for graph classification tasks. Our work has been motivated by how the empirical comparison among models for graphs has been conducted in the literature over the last few years. We identified several flaws in the evaluation protocols, such as poor standardization across different works in terms of data splits used, ambiguous or under-specified selection of hyper-parameters, and unfair comparisons in terms of features used by the different models under evaluation. The major part of the study is a re-evaluation of 5 state-of-the-art models in the literature across 9 different datasets, of which 4 are related to the classification of chemical properties of graphs, and 5 concern predictive tasks on social networks. We devised a standardized evaluation framework consisting of an internal hold-out based model selection and an external 10-fold cross-validation for model assessment. Importantly, we took all the necessary precautions to design the framework as rigorously and fairly as possible: all models have been trained on the same stratified data splits and assigned a congruent amount of hyper-parameters to optimize during model selection. Moreover, we release our code and data splits to allow fellow researchers to replicate our experiments and use it for further comparison (see Appendix B). One fundamental aspect of this work is the comparison against structure-agnostic baselines, which helps identify situations

where the models do not fully exploit relational knowledge during the learning process. Our results differ from those reported in the literature and are more consistent with what would be expected from a robust assessment. Concerning the comparison with the baselines, we have shown that Deep Graph Networks do not perform better than structure-agnostic baselines in some cases. This result suggests care to not overemphasize small performance improvements, which are more likely to be attributed to chance. Finally, we have shown how a fair comparison allows a reasoned analysis of specific properties of Deep Graph Networks. We have done so by demonstrating how using node degrees as features translates to a performance improvement across the social datasets; and which DGNs seem to be most stable in performance responding to a change in hyper-parameters. In principle, our analysis can be extended indefinitely by further works: by applying the same fair evaluation to different datasets, such as those in the Stanford Large Network Dataset Collection [LK14]; by comparing more Deep Graph Networks variants; by enlarging the grids of hyper-parameters each model optimizes; and by refining the internal model selection phase using cross-validation for a less unbiased estimation. Most importantly, after our work on the subject has been published, we have started witnessing a collective effort to standardize model assessment for Deep Graph Networks by the research community [Hu+20; Mor+20].

Our second contribution has been a novel application of Deep Graph Networks for graph classification to the problem of predicting the dynamical properties of biological pathways represented as graphs. Our focus has been on the property of robustness, which quantifies the stability of the associated dynamical system under perturbations of the initial conditions. Unlike traditional methods, which require to compute the robustness via expensive numerical simulations, we hypothesized that the structure of the graph representing the pathway correlates to its robustness and that this correlation could be exploited for predictive purposes. The potential implication of this assumption is that, to assess robustness or other dynamical properties, one is relieved from performing costly numerical simulations. Guided by this motivation, we developed a learning framework in which the structure of a biological pathway is given as input, and an indicator of whether the pathway is robust or not is predicted as output. The framework comprises a Deep Graph Network that processes the pathway structure and a downstream Multi-Layer Perceptron classifier, which takes care of the actual robustness prediction. We tested this framework on a dataset of real-world biological pathways taken from the BioModels database. The experimental results validate our initial assumption to a remarkable extent, paving the way for the use of learning methodologies to assess the dynamical properties of biochemical pathways. We have also shown that the pathway structure plays a major role to perform well in this task, since models that do not consider the pathway structure score not better than random. In subsequent works, it might be interesting to extend this approach further to the assessment of other dynamical properties of biochemical pathways, such

as, for example, monotonicity, oscillatory, and bi-stability properties. Another fascinating research avenue concerns the development of novel Deep Generative Models of biochemical pathways. The potential of such development would be two-fold. Firstly, a generative model would endow researchers with an explanatory model of how pathway structure relates to the dynamical property. Secondly, it could allow the study of dynamical properties even in cases where not enough relevant pathways are available.

The third contribution of our thesis has concerned the problem of graph generation. This problem is now being actively researched because of its connection to several domains such as network science or computational chemistry. Specifically, we developed a novel autoregressive model for unlabeled graph generation. Existing autoregressive approaches model graph generation in terms of sequences of nodes. We adopt a related but different formulation, which produces a graph by generating its sequence of edges ordered lexicographically. The autoregressive distribution is modeled by splitting the ordered edge sequence into two related sequences. The first is obtained from the edges' starting nodes, while the second is obtained from the ending nodes. To generate a graph, one firstly samples a sequence of starting nodes with an autoregressive RNN. This initial sequence is ultimately completed into an ordered edge sequence by a second RNN which predicts the end nodes sequence. We assess the performances of the proposed models on 6 different datasets, of which 4 are synthetic datasets of graphs with strong node/edge dependencies, and 2 are chemical datasets of molecules. The experimental evaluation has been conducted by taking into account quantitative (*i.e.* how many useful graphs the model can produce) and qualitative (*i.e.* how well the generated graphs resemble the training samples, both locally and globally) aspects. The results show that the model can approximate the desired graph distributions in all cases, performing on par with state-of-the-art approaches. The model also has limitations, in that it is not able (in its current form) to handle labeled generation, which is the next logical step to improve in subsequent works. Another interesting direction to take in the next future is to extend the model with attention mechanisms to integrate better the information provided by the first RNN into the second. More generally, generative models of graphs still need to be researched further and in-depth. Among the many topics, one with arguably the highest impact on the field's advancement is the development of efficient permutation-invariant decoders.

Lastly, we have presented an application of Deep Generative models of graphs in the context of molecular generation. Generating molecules is a critical phase of the drug discovery process, as it can potentially cut down the failures of the subsequent screening phases by selecting more suitable candidates. Approaches to the generation of molecules are based on two prominent model families. One approach is to linearize the molecular graphs in SMILES notation, learning a language model of strings in the SMILES language. The learned language model can be sampled to produce novel SMILES strings one character at a time. A different approach operates directly

on the molecular graph, generating its structure either at once or sequentially by incrementally adding nodes to an existing graph. This approach is arguably considered more powerful, as it leverages a more expressive representation of the molecule. In comparison, the SMILES-based approach grants superior speed in the training and generation phases. However, it also generates chemically invalid and duplicate molecules at discouraging high rates. Our work has addressed these two limitations. Specifically, to solve the problems related to chemical validity, we have adopted a sequential generative approach based on chemical fragments. Chemical fragments are small compounds that can be combined to form more useful molecules. The proposed model generates molecules as sequences of fragments, rather than SMILES characters. This solution has two positive effects. Firstly, it reduces the length of the sequence to be generated, which implies faster computation and avoidance of long-term dependency problems. Secondly, it lowers the model's chances of making wrong predictions that can undermine the chemical validity of the graph being generated. To solve the problems regarding the generation of duplicate (*i.e.* not unique) molecules, we have proposed a strategy termed Low-Frequency Masking. This strategy aims to encourage the model to generate molecules with fragments that appear less frequently in a dataset. We have tested our model on two major chemical benchmarks for molecule generation, evaluating it both quantitatively and qualitatively. The results show that our model dramatically improves upon SMILES-based models in the generation of valid and unique molecules, reaching a level of performances typical of the more expressive family of graph-based generative models. However, we also acknowledge that the model in its current form might have limited applicability to subsequent generative tasks in the drug discovery pipeline, such as molecular optimization. Overcoming this limitation constitutes an important research direction to pursue in subsequent works by devising more effective strategies to avoid the production of duplicates. A second improvement might be obtained by observing that the sequence of fragments is fully reversible: this means that one could try different strategies to construct the encoder, for example leveraging Bidirectional LSTMs [SP97]. Lastly, a third improvement might be achieved by merging the fragment-based approach with graph-based methodologies. We believe this combination has a clear "best of both worlds" potential. On the one hand, it could exploit the advantages of fragment-based generation in terms of speed of training and sampling. On the other hand, it could leverage the expressiveness of a graph-based representation of the chemical fragments, similarly to other interesting approaches in the field [Jin+19; Bra+20].

In conclusion, we believe that the results presented in this thesis demonstrate the flexibility of Deep Learning to (a) handle complex, variable-sized data structures, and (b) leverage the expressiveness of these data to give important contributions in the two domains of application. On a more general level, we believe that this thesis gives a glimpse of the role that Deep Learning for structured data might play in the years to come. Some problems in the life sciences, like DNA analysis, require an

astounding amount of computing power and memory to be tackled. Others, like the enumeration of all possible molecules, are simply impossible to solve naively. In these scenarios, one of the main roles of Deep Learning is likely that of a replacement or a complement for expensive numerical simulations or *in vivo* experiments, with the ultimate promise of advancing the state of knowledge more efficiently. This process has already started in fields like computational chemistry, where using predictive or generative models can save researchers a tremendous amount of time (and money) to conduct their studies. The computational biology application presented in this thesis is another example of how the same goal can be achieved in a different life sciences domain. We expect many more cases where Deep Learning models break through the barriers of a difficult biological problem in the near future. In the route towards understanding the mechanisms underlying biological life, Deep Learning can help not to cover the entire distance but to take faster steps towards the destination.

Appendix A

Hyper-Parameters Table

We report the grid of the hyper-parameters used in the experiments of Section 4.4. The hyper-parameters are the following:

- *layers*: number of DGN layers;
- *convs per layer*: number of GCL layers for DGN layer;
- *batch size*: size of the SGD minibatch;
- *learning rate*: SGD learning rate;
- *hidden size*: hidden state dimension of the GCL;
- *epochs*: number of training epochs;
- *l2*: L2 regularization parameter;
- *dropout*: dropout rate;
- *patience*: early stopping patience;
- *optimizer*: type of optimizer;
- *scheduler*: learning rate annealing scheduler;
- *dense dim.*: output layer dimension;
- *embed dim.*: size of the graph representation;
- *neighborhood aggregation*: size of neighborhood aggregation.

	Layers	Convs per layer	Batch size	Learning rate	Hidden units	Epochs	L2	Dropout	Patience	Optimizer	Scheduler	Dense dim	Embed. dim	Neighbors Aggregation
Baseline chemical	-	-	32 128	1e-1 1e-3 1e-6	32 128 256	5000	1e-2 1e-3 1e-4	-	500, loss 500, acc	Adam	-	-	-	sum
Baseline IMDB	-	-	32 128	1e-1 1e-3 1e-6	32 128 256	3000	1e-2 1e-3 1e-4	-	500, loss 500, acc	Adam	-	-	-	sum
Base. COLLAB and REDDIT	-	-	32 128	1e-1 1e-3	32 128	3000	1e-2 1e-3 1e-4	-	500, loss 500, acc	Adam	-	-	-	sum
Baseline ENZYMES	-	-	32	1e-1 1e-3 1e-6	32 64 128 256	5000	1e-2 1e-3 1e-4	-	1000, loss 1000, acc	Adam	-	-	-	sum
DGCNN	2 3 4	1	50 (cpu) 16 (gpu)	1e-4 1e-5	32 64	1000	-	0.5	500, loss 500, acc	Adam	-	128	-	mean
DiffPool	1 2	3	20 (cpu) 8 (gpu)	1e-3 1e-4 1e-5	32 64	3000	-	-	500, loss 500, acc	Adam	-	50	64 128	mean
ECC	1 2	3	32 (cpu) 8 (gpu)	1e-1 1e-2	32 64	1000	-	0.05 0.25	500, loss 500, acc	SGD	ECC-LR	-	-	sum
GIN	see hidden units	1	32 128	1e-2	32 (5 layers) 64 (5 layers) 64 (2 layers) 32 (3 layers)	1000	-	0 0.5	500, loss 500, acc	Adam	Step-LR (step: 50, gamma: 0.5)	-	-	sum
GraphSAGE	3 5	1	32 (cpu) 16 (cuda)	1e-2 1e-3 1e-4	32 64	1000	-	-	500, loss 500, acc	Adam	-	-	-	mean max sum

Appendix B

List of Publications

- F. Errica, M. Podda, D. Bacciu, A. Micheli. "A Fair Comparison of Graph Neural Networks for Graph Classification". *8th International Conference on Learning Representations (ICLR)*. (2020)
- M. Podda, D. Bacciu, A. Micheli. "A Deep Generative Model for Fragment-Based Molecule Generation". *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 108, pages 2240-2250. (2020)
- D. Bacciu, F. Errica, A. Micheli, M. Podda. "A Gentle Introduction to Deep Learning for Graphs". *Neural Networks*. Volume 129, Pages 203-221. ISSN 0893-6080, doi:10.1016/j.neunet.2020.06.006. (2020)
- P. Bove, P. Milazzo, A. Micheli, M. Podda. "Prediction of dynamical properties of biochemical pathways with Graph Neural Networks". *Proceedings of the 13th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS*. ISBN 978-989-758-398-8, pages 32-43. doi:10.5220/0008964700320043 (2020)
- M. Podda, D. Bacciu, A. Micheli, P. Milazzo. "Biochemical Pathway Robustness Prediction with Graph Neural Networks". *To appear in: ESANN 2020 Proceedings* (2020)
- D. Bacciu, A. Micheli, M. Podda. "Edge-based sequential graph generation with recurrent neural networks". *Neurocomputing*, Volume 416, pages 177-189, ISSN 0925-2312. doi:10.1016/j.neucom.2019.11.112. (2020)
- D. Bacciu, A. Micheli, M. Podda. "Graph generation by sequential edge prediction". *ESANN 2019 Proceedings*, pages 95-100, ISBN 978-287-587-065-0. (2019)
- M. Podda, D. Bacciu, A. Micheli, R. Bellù, G. Placidi, L. Gagliardi. "A machine learning approach to estimating preterm infants survival: development of the Preterm Infants Survival Assessment (PISA) predictor". *Sci Rep* 8, 13743. doi:10.1038/s41598-018-31920-6 (2018)

Appendix C

Contributed Code

- Code for the article: M. Podda, D. Bacciu, A. Micheli, R. Bellù, G. Placidi, L. Gagliardi. "A machine learning approach to estimating preterm infants survival: development of the Preterm Infants Survival Assessment (PISA) predictor". *Sci Rep* 8, 13743. doi:10.1038/s41598-018-31920-6 (2018)
  URL: https://github.com/marcopodda/inn

- Code for the article: D. Bacciu, A. Micheli, M. Podda. "Edge-based sequential graph generation with recurrent neural networks". *Neurocomputing*, Volume 416, pages 177-189, ISSN 0925-2312. doi:10.1016/j.neucom.2019.11.112. (2020)
  URL: https://github.com/marcopodda/grapher

- Code for the conference paper: M. Podda, D. Bacciu, A. Micheli. "A Deep Generative Model for Fragment-Based Molecule Generation". *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 108, pages 2240-2250. (2020)
  URL: https://github.com/marcopodda/fragment-based-dgm

- Code for the conference paper: F. Errica, M. Podda, D. Bacciu, A. Micheli. "A Fair Comparison of Graph Neural Networks for Graph Classification". *8th International Conference on Learning Representations (ICLR)*. (2020)
  URL: https://github.com/diningphil/gnn-comparison

Bibliography

[Aba+16] M. Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[AC10] S. Arlot and A. Celisse. "A Survey of Cross-Validation Procedures for Model Selection". In: *Statistics Survey* 4 (2010), pp. 40–79.

[Agu+16] S. Aguiñaga et al. "Growing Graphs from Hyperedge Replacement Graph Grammars". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, pp. 469–478.

[Air+08] E. M. Airoldi et al. "Mixed Membership Stochastic Blockmodels". In: *Journal of Machine Learning Research* 9 (2008), pp. 1981–2014.

[BA99] A.-L. Barabási and R. Albert. "Emergence of Scaling in Random Networks". In: *Science* 286 (1999), pp. 509–512.

[Bac+20] D. Bacciu et al. "A gentle introduction to deep learning for graphs". In: *Neural Networks* 129 (2020), pp. 203–221.

[Bal12] P. Baldi. "Autoencoders, Unsupervised Learning, and Deep Architectures". In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. Vol. 27. Proceedings of Machine Learning Research. JMLR Workshop and Conference Proceedings, 2012, pp. 37–49.

[BB12a] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305.

[BB12b] J. S. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.1 (2012), pp. 281–305.

[BC19] D. Bacciu and D. Castellana. "Bayesian mixtures of hidden tree Markov models for structured data clustering". In: *Neurocomputing* 342 (2019). Publisher: Elsevier, pp. 49–59.

[BCB15] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ICLR*. 2015.

[BCV13] Y. Bengio, A. Courville, and P. Vincent. "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (2013), pp. 1798–1828.

[BEM18] D. Bacciu, F. Errica, and A. Micheli. "Contextual Graph Markov Model: A Deep and Generative Approach to Graph Processing". In: *Proceedings of the International Conference on Machine Learning (ICML)*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 294–303.

[Ben+17] A. Benavoli et al. "Time for a Change: a Tutorial for Comparing Multiple Classifiers Through Bayesian Analysis". In: *Journal of Machine Learning Research* 18.77 (2017), pp. 1–36.

[Ben09] Y. Bengio. "Learning Deep Architectures for AI". In: *Found. Trends Mach. Learn.* 2.1 (Jan. 2009), pp. 1–127.

[Bia+00a] A. Bianucci et al. "Application of Cascade Correlation Networks for Structures to Chemistry". In: *Appl. Intell.* 12 (Jan. 2000), pp. 115–145.

[Bia+00b] A. M. Bianucci et al. "Application of cascade correlation networks for structures to chemistry". In: *Applied Intelligence* 12.1-2 (2000). Publisher: Springer, pp. 117–147.

[Bis06] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[BLC13] Y. Bengio, N. Léonard, and A. C. Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation". In: *ArXiv* abs/1308.3432 (2013).

[BM+76] J. A. Bondy, U. S. R. Murty, et al. *Graph theory with applications*. Vol. 290. Macmillan London, 1976.

[BMP19a] D. Bacciu, A. Micheli, and M. Podda. "Edge-based sequential graph generation with recurrent neural networks". In: *Neurocomputing* (2019).

[BMP19b] D. Bacciu, A. Micheli, and M. Podda. "Graph generation by sequential edge prediction". In: *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. 2019.

[BMP20] D. Bacciu, A. Micheli, and M. Podda. "Edge-based sequential graph generation with recurrent neural networks". In: *Neurocomputing* (2020).

[Bor+05] K. M. Borgwardt et al. "Protein function prediction via graph kernels". In: *Bioinformatics* 21.suppl_1 (2005), pp. i47–i56.

[Bov+20] P. Bove. et al. "Prediction of Dynamical Properties of Biochemical Pathways with Graph Neural Networks". In: *Proceedings of the 13th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS*. INSTICC. SciTePress, 2020, pp. 32–43.

[Bow+16] S. R. Bowman et al. "Generating Sentences from a Continuous Space". In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. 2016, pp. 10–21.

[BPB+12] G. R. J. Bickerton, G. V. Paolini, J. Besnard, et al. "Quantifying the chemical beauty of drugs." In: *Nature chemistry* 4 (2 2012), pp. 90–98.

[Bra+19] J. Bradshaw et al. "A model to search for synthesizable molecules". In: *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*. 2019, pp. 7935–7947.

[Bra+20] J. Bradshaw et al. "Barking up the right tree: an approach to search over molecule synthesis DAGs". In: *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*. 2020.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.

[BT17] E. Bjerrum and R. Threlfall. "Molecular Generation with Recurrent Neural Networks (RNNs)". In: *ArXiv* abs/1705.04612 (2017).

[CG10] F. Costa and K. Grave. "Fast Neighborhood Subgraph Pairwise Distance Kernel". In: Aug. 2010, pp. 255–262.

[Cho+14] K. Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734.

[Cos+] F. Costa et al. "Learning incremental syntactic structures with recursive neural networks". In: *KES'2000. Fourth International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technologies. Proceedings (Cat. No. 00TH8516)*. Vol. 2. IEEE, pp. 458–461.

[Cos+03] F. Costa et al. "Towards incremental parsing of natural language using recursive neural networks". In: *Applied Intelligence* 19.1-2 (2003), pp. 9–25.

[Cos17] F. Costa. "Learning an efficient constructive sampler for graphs". In: *Artificial Intelligence* 244 (2017). Combining Constraint Solving with Mining and Learning, pp. 217–238.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[DD03] P. D. Dobson and A. J. Doig. "Distinguishing enzyme structures from non-enzymes without alignments". In: *Journal of molecular biology* 330.4 (2003), pp. 771–783.

[DGH16] J. A. DiMasi, H. G. Grabowski, and R. W. Hansen. "Innovation in the pharmaceutical industry: New estimates of R&D costs". In: *Journal of Health Economics* 47 (2016), pp. 20–33.

[DGK07] I. S. Dhillon, Y. Guan, and B. Kulis. "Weighted graph cuts without eigenvectors a multilevel approach". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.11 (2007). Publisher: IEEE, pp. 1944–1957.

[DK18] N. De Cao and T. Kipf. "MolGAN: An implicit generative model for small molecular graphs". In: *Workshop on Theoretical Foundations and Applications of Deep Generative Models, International Conference on Machine Learning (ICML)* (2018).

[DR17] B. J. Davis and S. D. Roughley. "Fragment-Based Lead Discovery". In: *Platform Technologies in Drug Discovery and Validation*. Vol. 50. Annual Reports in Medicinal Chemistry. Academic Press, 2017. Chap. 11, pp. 371–439.

[DTD+18] H. Dai, Y. Tian, B. Dai, et al. "Syntax-Directed Variational Autoencoder for Structured Data". In: *ICLR*. 2018.

[Duv+15] D. K. Duvenaud et al. "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems 28*. 2015, pp. 2224–2232.

[DWZ+08] J. Degen, C. Wegscheid-Gerlach, A. Zaliani, et al. "On the Art of Compiling and Using 'Drug-Like' Chemical Fragment Spaces." In: *ChemMedChem* 3 (10 2008), pp. 1503–1507.

[Elm90] J. L. Elman. "Finding structure in time". In: *Cognitive Science* 14.2 (1990). Publisher: Wiley Online Library, pp. 179–211.

[EMO04] D. A. Erlanson, R. S. McDowell, and T. O'Brien. "Fragment-Based Drug Discovery". In: *Journal of Medicinal Chemistry* 47.14 (2004), pp. 3463–3482.

[ER59] P. Erdös and A. Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[Err+20] F. Errica et al. "A fair comparison of graph neural networks for graph classification". In: *Proceedings of the 8th International Conference on Learning Representations (ICLR)*. 2020.

[ES09] P. Ertl and A. Schuffenhauer. "Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions". In: *Journal of Cheminformatics* 1.1 (2009).

[Faw06] T. Fawcett. "An Introduction to ROC Analysis". In: *Pattern Recognition Letters* 26 (2006), pp. 861–874.

[FGS98] P. Frasconi, M. Gori, and A. Sperduti. "A general framework for adaptive processing of data structures". In: *IEEE Trans Neural Netw* 9.5 (1998), pp. 768–86.

[FL90] S. E. Fahlman and C. Lebiere. "The Cascade-Correlation learning architecture". In: *Proceedings of the 3rd Conference on Neural Information Processing Systems (NIPS)*. 1990, pp. 524–532.

[FS08] F. Fages and S. Soliman. "From Reaction Models to Influence Graphs and Back: A Theorem". In: *Formal Methods in Systems Biology*. Ed. by J. Fisher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 90–102.

[Fun+03] A. Funahashi et al. "CellDesigner: a process diagram editor for gene-regulatory and biochemical networks". In: *Biosilico* 1.5 (2003), pp. 159–162.

[FXS20] T. Fu, C. Xiao, and J. Sun. "CORE: Automatic Molecule Optimization Using Copy & Refine Strategy". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (Apr. 2020), pp. 638–645.

[GBB11] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[GBL98] C. L. Giles, K. D. Bollacker, and S. Lawrence. "CiteSeer: An Automatic Citation Indexing System". In: *Proceedings of the Third ACM Conference on Digital Libraries*. 1998, pp. 89–98.

[Ger+15] M. Germain et al. "MADE: Masked Autoencoder for Distribution Estimation". In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. JMLR Proceedings. 2015, pp. 881–889.

[GG16] Y. Gal and Z. Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research. PMLR, 2016, pp. 1050–1059.

[GHL07] D. Gilbert, M. Heiner, and S. Lehrack. "A Unifying Framework for Modelling and Analysing Biochemical Pathways Using Petri Nets". In: *Computational Methods in Systems Biology*. Ed. by M. Calder and S. Gilmore. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 200–216.

[Gil+17] J. Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the International Conference on Machine Learning (ICML)*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1263–1272.

[Gil77] D. T. Gillespie. "Exact stochastic simulation of coupled chemical reactions". In: *The journal of physical chemistry* 81.25 (1977), pp. 2340–2361.

[GJ20] N. Gruber and A. Jockisch. "Are GRU Cells More Specific and LSTM Cells More Sensitive in Motive Classification of Text?" In: *Frontiers in Artificial Intelligence* 3 (2020), p. 40.

[GJR20] N. Goyal, H. V. Jain, and S. Ranu. "GraphGen: A Scalable Approach to Domain-Agnostic Labeled Graph Generation". In: *Proceedings of The Web Conference 2020*. WWW '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1253–1263.

[GK96] C. Goller and A. Kuchler. "Learning task-dependent distributed representations by backpropagation through structure". In: *Proceedings of the International Conference on Neural Networks (ICNN)*. Vol. 1. IEEE, 1996, pp. 347–352.

[GM10] C. Gallicchio and A. Micheli. "Graph echo state networks". In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010, pp. 1–8.

[GM20] C. Gallicchio and A. Micheli. "Fast and deep graph neural networks". In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*. 2020.

[GMN19] R. Gori, P. Milazzo, and L. Nasti. "Towards an Efficient Verification Method for Monotonicity Properties of Chemical Reaction Networks". In: *Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS,* INSTICC. SciTePress, 2019, pp. 250–257.

[GN02] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826.

[Goh+17] G. Goh et al. "ChemNet: A Transferable and Generalizable Deep Neural Network for Small-Molecule Property Prediction". In: *arXiv preprint arXiv:1712.02734* (2017).

[Góm+18] R. Gómez-Bombarelli et al. "Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules". In: *ACS Central Science* 4.2 (2018), pp. 268–276.

[Goo+14] I. Goodfellow et al. "Generative Adversarial Nets". In: *Proceedings of the 28th Conference on Neural Information Processing Systems (NIPS)*. 2014, pp. 2672–2680.

[Gra13] A. Graves. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[Gre+12] A. Gretton et al. "A Kernel Two-Sample Test". In: *JOURNAL OF MACHINE LEARNING RESEARCH* 13 (2012), pp. 723–773.

[GSZ+16] A. Gindulyte, B. A. Shoemaker, J. Zhang, et al. "PubChem BioAssay: 2017 update". In: *Nucleic Acids Research* 45.D1 (2016), pp. D955–D963.

[GVS15] I. Goodfellow, O. Vinyals, and A. Saxe. "Qualitatively Characterizing Neural Network Optimization Problems". In: *International Conference on Learning Representations*. 2015.

[GZ20] X. Guo and L. Zhao. *A Systematic Survey on Deep Generative Models for Graph Generation*. 2020.

[Hay09] S. Haykin. *Neural Networks and Learning Machines (3rd Edition)*. Pearson, 2009.

[He+16] K. He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[HF99] B. Hammer and O. Februar. "Learning with Recurrent Neural Networks". In: *Lecture Notes in Control and Information Sciences 254*. Springer, 1999, pp. 357–368.

[HMS05] B. Hammer, A. Micheli, and A. Sperduti. "Universal Approximation Capability of Cascade Correlation for Structures". In: *Neural Computation* 17.5 (2005), pp. 1109–1159.

[HS97] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997). Publisher: MIT Press, pp. 1735–1780.

[HSS08] T. Hofmann, B. Schölkopf, and A. J. Smola. "Kernel methods in machine learning". In: *Ann. Statist.* 36.3 (June 2008), pp. 1171–1220.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2nd. Springer New York Inc., 2009.

[Hu+20] W. Hu et al. "Open Graph Benchmark: Datasets for Machine Learning on Graphs". In: *arXiv preprint arXiv:2005.00687* (2020).

[Huc+18] M. Hucka et al. "The Systems Biology Markup Language (SBML): language specification for level 3 version 2 core". In: *Journal of integrative bioinformatics* 15.1 (2018).

[HYL17] W. Hamilton, Z. Ying, and J. Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2017, pp. 1024–1034.

[IL15] B. Iooss and P. Lemaître. "A review on global sensitivity analysis methods". In: *Uncertainty management in simulation-optimization of complex systems*. Springer, 2015, pp. 101–122.

[IS05] J. J. Irwin and B. K. Shoichet. "ZINC – A Free Database of Commercially Available Compounds for Virtual Screening". In: *Journal of Chemical Information and Modeling* 45.1 (2005), pp. 177–182.

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. JMLR.org, 2015, pp. 448–456.

[Jae02] H. Jaeger. "Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach". In: *GMD-Forschungszentrum Informationstechnik, 2002.* 5 (Jan. 2002).

[JBJ18] W. Jin, R. Barzilay, and T. S. Jaakkola. "Junction tree variational autoencoder for molecular graph generation". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018, pp. 2328–2337.

[JC17] K. Janocha and W. M. Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: *Conference Theoretical Foundations of Machine Learning*. TFML. 2017.

[JGP17] E. Jang, S. Gu, and B. Poole. "Categorical reparametrization with gumbel-softmax". In: *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. 2017.

[Jin+19] W. Jin et al. "Learning Multimodal Graph-to-Graph Translation for Molecule Optimization". In: *International Conference on Learning Representations*. 2019.

[KB15] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations*. ICLR. 2015.

[Ker+16] K. Kersting et al. *Benchmark Data Sets for Graph Kernels*. 2016.

[Kit04] H. Kitano. "Biological robustness". In: *Nature Reviews Genetics* 5.11 (2004), p. 826.

[Kit07] H. Kitano. "Towards a theory of biological robustness". In: *Molecular systems biology* 3.1 (2007).

[Koh+95] R. Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 14. 2. 1995, pp. 1137–1145.

[KP94] P. D. Karp and S. M. Paley. "Representations of metabolic knowledge: pathways." In: *Ismb*. Vol. 2. 1994, pp. 203–211.

[KPH17] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato. "Grammar Variational Autoencoder". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. 2017, pp. 1945–1954.

[KSH17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *CACM*. 2017.

[KW14] D. P. Kingma and M. Welling. "Auto-encoding variational Bayes". In: *Proceedings of the 2nd International Conference on Learning Representations (ICLR)* (2014).

[KW17] T. N. Kipf and M. Welling. "Semi-supervised classification with graph convolutional networks". In: *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. 2017.

[LAB+18] Q. Liu, M. Allamanis, M. Brockschmidt, et al. "Constrained Graph Variational Autoencoders for Molecule Design". In: *Advances in Neural Information Processing Systems 31*. 2018, pp. 7795–7804.

[Lan13] G. Landrum. *RDKit: Open-source cheminformatics*. http://www.rdkit.org. [Online; accessed 11-April-2013]. 2013.

[Lar+11] A. Larhlimi et al. "Robustness of metabolic networks: a review of existing definitions". In: *Biosystems* 106.1 (2011), pp. 1–8.

[LB+95] Y. LeCun, Y. Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The Handbook of Brain Theory and Neural Networks* 3361.10 (1995), p. 1995.

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444.

[Le +06] N. Le Novere et al. "BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems". In: *Nucleic acids research* 34.suppl_1 (2006), pp. D689–D691.

[Le +09] N. Le Novere et al. "The systems biology graphical notation". In: *Nature biotechnology* 27.8 (2009), p. 735.

[LeC+98] Y. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 9–50.

[Les+10] J. Leskovec et al. "Kronecker Graphs: An Approach to Modeling Networks". In: *Journal of Machine Learning Research* 11 (2010), pp. 985–1042.

[LHW18] Q. Li, Z. Han, and X.-M. Wu. "Deeper insights into graph convolutional networks for semi-supervised learning". In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*. 2018.

[Li+10] C. Li et al. "BioModels Database: An enhanced, curated and annotated resource for published quantitative kinetic models." In: *BMC Systems Biology* 4 (June 2010), p. 92.

[Li+18] Y. Li et al. "Learning deep generative models of graphs". In: *CoRR* abs/1803.03324 (2018).

[LK14] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[LM14] Q. V. Le and T. Mikolov. "Distributed Representations of Sentences and Documents". In: *Proceedings of the 31th International Conference on Machine Learning, ICML*. 2014, pp. 1188–1196.

[LPB13] A. Lusci, G. Pollastri, and P. Baldi. "Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules". In: *Journal of chemical information and modeling* 53.7 (July 2013), pp. 1563–75.

[LS18] Z. C. Lipton and J. Steinhardt. "Troubling trends in machine learning scholarship". In: *arXiv preprint arXiv:1807.03341* (2018).

[LZB+16] Y. Li, R. Zemel, M. Brockschmidt, et al. "Gated Graph Sequence Neural Networks". In: *Proceedings of ICLR*. 2016.

[Mar+20] P. Maragakis et al. "A deep-learning view of chemical space designed to facilitate drug discovery". In: *arXiv preprint arXiv:2002.02948* (2020).

[Mer+18] D. Merk et al. "De Novo Design of Bioactive Small Molecules by Artificial Intelligence". In: *Molecular Informatics* 37.1-2 (2018), p. 1700153.

[MHN18] E. S. Marquez, J. S. Hare, and M. Niranjan. "Deep cascade learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.11 (2018). Publisher: IEEE, pp. 5475–5485.

[Mic09] A. Micheli. "Neural Network for Graphs: A Contextual Constructive Approach". In: *Trans. Neur. Netw.* 20.3 (2009), pp. 498–511.

[Mit97] T. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[ML16] S. Mohamed and B. Lakshminarayanan. "Learning in Implicit Generative Models". In: *CoRR* abs/1610.03483 (2016).

[Mor+19] C. Morris et al. "Weisfeiler and leman go neural: Higher-order graph neural networks". In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*. Vol. 33. 2019, pp. 4602–4609.

[Mor+20] C. Morris et al. "TUDataset: A collection of benchmark datasets for learning with graphs". In: *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*. 2020.

[MP07] S. A. Macskassy and F. Provost. "Classification in networked data: A toolkit and a univariate case study". In: *Journal of Machine Learning Research* 8.May (2007), pp. 935–983.

[MSS07] A. Micheli, A. Sperduti, and A. Starita. "An introduction to recursive neural networks and kernel methods for cheminformatics". In: *Current Pharmaceutical Design* 13.14 (2007), pp. 1469–1496.

[Nea92] R. M. Neal. "Connectionist Learning of Belief Networks". In: *Artif. Intell.* 56.1 (July 1992), pp. 71–113.

[Nei+18] D. Neil et al. "Exploring Deep Recurrent Models with Reinforcement Learning for Molecule Design". In: *International Conference of Learning Representations (ICLR) Workshop*. 2018.

[NGM18] L. Nasti, R. Gori, and P. Milazzo. "Formalizing a notion of concentration robustness for biochemical networks". In: *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer. 2018, pp. 81–97.

[Niu+20] C. Niu et al. "Permutation Invariant Graph Generation via Score-Based Generative Modeling". In: *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*. Vol. 108. Proceedings of Machine Learning Research. PMLR, 2020, pp. 4474–4484.

[Oli+17] M. Olivecrona et al. "Molecular de-novo design through deep reinforcement learning". In: *Journal of Cheminformatics* 9.1 (Sept. 2017).

[Pas+17] A. Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[PBM20] M. Podda, D. Bacciu, and A. Micheli. "A Deep Generative Model for Fragment-Based Molecule Generation". In: *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 108. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2240–2250.

[Pet77] J. L. Peterson. "Petri nets". In: *ACM Computing Surveys (CSUR)* 9.3 (1977), pp. 223–252.

[Pod+20] M. Podda et al. "Biochemical Pathway Robustness Prediction with Graph Neural Networks". In: *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*. 2020.

[Pop+18] R. Poplin et al. "A universal SNP and small-indel variant caller using deep neural networks". In: *Nature Biotechnology* 36.10 (Nov. 2018), pp. 983–987.

[Pop+19] M. Popova et al. "MolecularRNN: Generating realistic molecular graphs with optimized properties". In: *arXiv preprint arXiv:1905.13372* (2019).

[Pre+18] K. Preuer et al. "Fréchet ChemNet Distance: A Metric for Generative Models for Molecules in Drug Discovery". In: *Journal of Chemical Information and Modeling* 58.9 (2018), pp. 1736–1741.

[Pre98] L. Prechelt. "Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[Ral+05] L. Ralaivola et al. “Graph kernels for chemical informatics”. In: *Neural networks* 18.8 (2005), pp. 1093–1110.

[RDR+14] R. Ramakrishnan, P. O. Dral, M. Rupp, et al. “Quantum chemistry structures and properties of 134 kilo molecules”. In: *Scientific Data* 1 (2014).

[Rif+11] S. Rifai et al. “Contractive Auto-Encoders: Explicit Invariance during Feature Extraction”. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML’11. 2011, pp. 833–840.

[Riz+09] A. Rizk et al. “A general computational method for robustness analysis with applications to synthetic gene networks”. In: *Bioinformatics* 25.12 (2009), pp. i169–i178.

[RM15] D. Rezende and S. Mohamed. “Variational Inference with Normalizing Flows”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. PMLR, 2015, pp. 1530–1538.

[RM51] H. Robbins and S. Monro. “A Stochastic Approximation Method”. In: *Ann. Math. Statist.* 22.3 (Sept. 1951), pp. 400–407.

[RMH86] D. E. Rumelhart, J. L. McClelland, and G. E. Hinton. “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*. MIT Press, 1986, pp. 77–109.

[RML+93] V. N. Reddy, M. L. Mavrovouniotis, M. N. Liebman, et al. “Petri net representations in metabolic pathways.” In: *ISMB*. Vol. 93. 1993, pp. 328–336.

[Rob+07] G. Robins et al. “Recent developments in exponential random graph (p*) models for social networks”. In: *Social Networks* 29.2 (2007). Special Section: Advances in Exponential Random Graph (p*) Models, pp. 192–215.

[Ros58] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* 65.6 (1958), pp. 65–386.

[Ros63] J. van Rossum. “The Relation Between Chemical Structure and Biological Activity”. In: *Journal of Pharmacy and Pharmacology* 15.1 (1963), pp. 285–316.

[Rud16] S. Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[Sal+16] T. Salimans et al. “Improved Techniques for Training GANs”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. 2016, pp. 2234–2242.

[Sam+19] B. Samanta et al. "NeVAE: A deep generative model for molecular graphs". In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*. 2019, pp. 1110–1117.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[Sca+09] F. Scarselli et al. "The Graph Neural Network Model". In: *Trans. Neur. Netw.* 20.1 (2009), pp. 61–80.

[Sch+01] B. Schölkopf et al. "Estimating the Support of a High-Dimensional Distribution". In: *Neural Comput.* 13.7 (July 2001), pp. 1443–1471.

[Sch+04] I. Schomburg et al. "BRENDA, the enzyme database: updates and major new developments". In: *Nucleic acids research* 32.suppl_1 (2004), pp. D431–D433.

[Sch+18] M. Schlichtkrull et al. "Modeling relational data with graph convolutional networks". In: *Proceedings of the 15th European Semantic Web Conference (ESWC)*. Springer, 2018, pp. 593–607.

[Seg+17] M. H. S. Segler et al. "Generating Focused Molecule Libraries for Drug Discovery with Recurrent Neural Networks". In: *ACS Central Science* 4.1 (Dec. 2017), pp. 120–131.

[Sen+20] A. W. Senior et al. "Improved protein structure prediction using potentials from deep learning". In: *Nature* 577.7792 (2020), pp. 706–710.

[SF10] G. Shinar and M. Feinberg. "Structural sources of robustness in biochemical reaction networks". In: *Science* 327.5971 (2010), pp. 1389–1391.

[Shc+18] O. Shchur et al. "Pitfalls of graph neural network evaluation". In: *Relational Representation Learning Workshop, Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[Siv+11] K. C. Sivakumar et al. "A systems biology approach to model neural stem cell regulation by notch, shh, wnt, and EGF signaling pathways". In: *Omics: a journal of integrative biology* 15.10 (2011), pp. 729–737.

[SK17] M. Simonovsky and N. Komodakis. "Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 29–38.

[SK18] M. Simonovsky and N. Komodakis. "GraphVAE: Towards generation of small graphs using variational autoencoders". In: *Proceedings of the 27th International Conference on Artificial Neural Networks (ICANN)*. 2018, pp. 412–422.

[Soc+11] R. Socher et al. "Parsing natural scenes and natural language with recursive neural networks". In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. 2011, pp. 129–136.

[Soc+13] R. Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[Som+15] E. T. Somogyi et al. “libRoadRunner: a high performance SBML simulation and analysis library”. In: *Bioinformatics* 31.20 (2015), pp. 3315–3321.

[SP97] M. Schuster and K. Paliwal. “Bidirectional recurrent neural networks”. In: *Signal Processing, IEEE Transactions on* 45 (Dec. 1997), pp. 2673–2681.

[Sri+14] N. Srivastava et al. “Dropout: a Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[SS95] H. Siegelmann and E. Sontag. “On the Computational Power of Neural Nets”. In: *Journal of Computer and System Sciences* 50.1 (1995), pp. 132–150.

[SS97] A. Sperduti and A. Starita. “Supervised neural networks for the classification of structures”. In: *IEEE Transactions on Neural Networks* 8.3 (1997), pp. 714–735.

[Stå+19] N. Ståhl et al. “Deep Reinforcement Learning for Multiparameter Optimization in de novo Drug Design”. In: *Journal of Chemical Information and Modeling* 59.7 (2019), pp. 3166–3176.

[Sto+20] J. M. Stokes et al. “A Deep Learning Approach to Antibiotic Discovery”. In: *Cell* 180.4 (Feb. 2020), 688–702.e13.

[Stu+03] P. Sturt et al. “Learning first-pass structural attachment preferences with dynamic grammars and recursive neural networks”. In: *Cognition* 88.2 (2003), pp. 133–169.

[van+16] A. van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *CoRR* abs/1609.03499 (2016).

[Vap00] V. N. Vapnik. *The Nature of Statistical Learning*. Springer, 2000.

[Vas+17] A. Vaswani et al. “Attention is all you need”. In: *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*. 2017, pp. 5998–6008.

[Vel+18] P. Velickovic et al. “Graph attention networks”. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018.

[Vin+10] P. Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *J. Mach. Learn. Res.* 11 (2010), pp. 3371–3408.

[Von07] U. Von Luxburg. “A tutorial on spectral clustering”. In: *Statistics and Computing* 17.4 (2007). Publisher: Springer, pp. 395–416.

[Wer88] P. J. Werbos. "Generalization of backpropagation with application to a recurrent gas market model". In: *Neural Networks* 1.4 (1988), pp. 339–356.

[Wil92] R. J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256.

[WM97] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1 (Apr. 1997), pp. 67–82.

[WS98] D. J. Watts and S. H. Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393.6684 (June 1998), pp. 440–442.

[WWK08] N. Wale, I. A. Watson, and G. Karypis. "Comparison of descriptor spaces for chemical compound retrieval and classification". In: *Knowledge and Information Systems* 14.3 (2008), pp. 347–375.

[WWW89] D. Weininger, A. Weininger, and J. L. Weininger. "SMILES. 2. Algorithm for generation of unique SMILES notation". In: *Journal of Chemical Information and Computer Sciences* 29.2 (May 1989), pp. 97–101.

[WZ89] R. J. Williams and D. Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In: *Neural Comput.* 1.2 (1989), pp. 270–280.

[Xu+19] K. Xu et al. "How Powerful are Graph Neural Networks?" In: *International Conference on Learning Representations (ICLR)* (2019).

[Yin+18] Z. Ying et al. "Hierarchical Graph Representation Learning with Differentiable Pooling". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2018, pp. 4800–4810.

[You+18a] J. You et al. "Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018, pp. 6410–6421.

[You+18b] J. You et al. "GraphRNN: Generating realistic graphs with deep auto-regressive models". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018.

[YV15] P. Yanardag and S. V. N. Vishwanathan. "Deep Graph Kernels". In: *Proceedings of the 21th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 2015, pp. 1365–1374.

[Zah+17] M. Zaheer et al. "Deep Sets". In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2017, pp. 3391–3401.

[Zha+18] M. Zhang et al. "An End-to-End Deep Learning Architecture for Graph Classification". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2018, pp. 4438–4445.

[Zha+20] J. Zhang et al. "Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity". In: *International Conference on Learning Representations*. 2020.

[Zi11] Z. Zi. "Sensitivity analysis approaches applied to systems biology models". In: *IET systems biology* 5.6 (2011), pp. 336–346.