Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Deep Generative Models for Molecular Design

Luís Henrique Simplício Ribeiro

JUIZ DE FORA
SETEMBRO, 2021

Deep Generative Models for Molecular Design

Luís Henrique Simplício Ribeiro

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Saulo Moraes Villela

JUIZ DE FORA

SETEMBRO, 2021

Deep Generative Models for Molecular Design

Luís Henrique Simplício Ribeiro

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Saulo Moraes Villela
Doutor em Engenharia de Sistemas e Computação

Marcelo Bernardes Vieira
Doutor em Ciência da Computação

Luiz Maurílio da Silva Maciel
Doutor em Engenharia de Sistemas e Computação

JUIZ DE FORA
09 DE SETEMBRO, 2021

Aos meus pais, irmãos e amigos

Resumo

Grande parte dos esforços empregados hoje em ciência dos materiais e química vem da necessidade de se criar moléculas que possuam propriedades específicas, como por exemplo na medicina, onde a geração de moléculas com propriedades desejadas desempenha um papel fundamental na descoberta de novos fármacos. Porém, apesar de existir um número finito de moléculas, tal número é muito grande, tornando uma exploração ingênua desse espaço de moléculas em um problema de otimização combinatória de difícil solução. Sendo assim, se faz necessário encontrar uma maneira robusta e eficiente de se explorar esse espaço de moléculas. Nesse cenário, técnicas como algoritmos genéticos vinham sendo utilizadas para tentar lidar com o problema de gerar moléculas com propriedades específicas. Mais recentemente, os Modelos Generativos Profundos, que são modelos em Aprendizagem de Máquina capazes de aprender através de dados e gerar novos dados que se assemelham ao conjunto em que foi treinado, vêm sendo propostos como uma alternativa ao problema. Com este trabalho apresentamos um estudo detalhado do complexo problema de gerar moléculas utilizando Aprendizado Profundo.

Palavras-chave: Aprendizagem de Máquina, Aprendizagem Profunda, Modelos Generativos Profundos, Geração molecular.

Abstract

Most of the effort spent today in material science and chemistry comes from the need to create molecules with specific properties, for instance, in medicine, the generation of molecules with desired properties plays a very important role in the discovery of new medicines. However, despite existing a finite number of molecules, this number is huge, making the exploration of this molecular space a hard problem of combinatorial optimization. Therefore, it is necessary to find a robust and efficient way to explore this molecular space. In this scenario, approaches like genetic algorithms had been used to try to handle the problem of generating molecules with specific properties computationally. More recently, Deep Generative Models, that are Machine Learning models capable of learning from data to generate new data that resemble the data they were trained on, are being used as an alternative to tackle the problem. With this work we present a detailed study of the complex problem of generating molecules using Deep Learning.

Keywords: Machine Learning, Deep Learning, Deep Generative Models, Molecular Generation.

Agradecimentos

Primeiramente, agradeço aos meus pais por todo apoio, sacrifícios e amor durante minha jornada acadêmica.

Aos meus irmãos por todo apoio e todo amor.

Ao professor Saulo pela orientação, pelos ensinamentos, paciência e todo suporte durante o meu projeto de pesquisa e a escrita do TCC.

Aos professores Magno e Wilhelm do Departamento de Matemática, que foram cruciais para o meu enriquecimento acadêmico, profissional e pessoal, por todos os ensinamentos, e a todos os outros professores que durante esses anos, contribuíram de algum modo para a minha formação.

"All models are wrong, but some are useful".

George Box

Contents

List of Figures

List of Tables

List of Abbreviations

CFG	Context-Free Grammar
CNN	Convolutional Neural Network
DL	Deep Learning
GAN	Generative Adversarial Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
ML	Machine Learning
RNN	Recurrent Neural Network
SMILES	Simplified Molecular Input Line Entry Specification
VAE	Variational Autoencoder

1 Introduction

1.1 Molecular Design

During World War I (1914-1918), many wounded soldiers would die as a result of bacterial infection. At that time, there were no effective treatments for infections such as pneumonia and gonorrhea. Hence, when a patient had such infections, there was not much that Doctors could do to save them. In 1928, Alexander Fleming, a Scottish physician, made the discovery by accident of Penicillin (Figure 1.1), a small molecule that could kill bacteria (GAYNES, 2017). That accident would change the world forever and would give him the 1945 Nobel Prize in Physiology or Medicine. The discovery of Penicillin was responsible for saving the life of thousands of allied soldiers during World War II, and more broadly, the life of millions of people around the world, since 1928. The discovery of Penicillin, which was the first antibiotic, is considered as one of the greatest advances in therapeutic medicine to date.

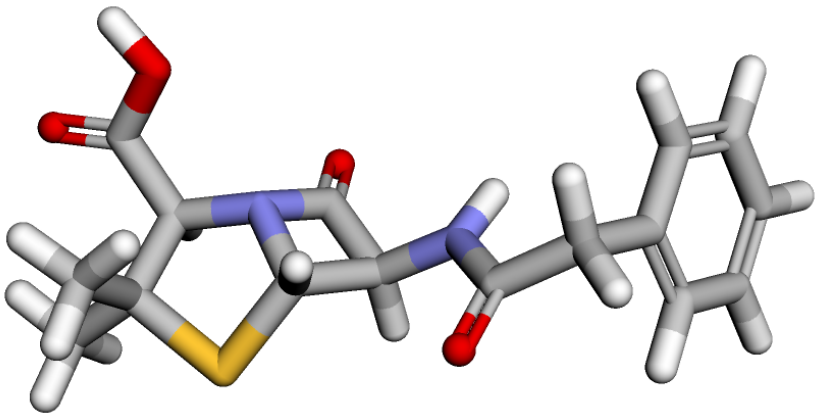


Figure 1.1: Core structure of Penicillin. Generated with: 3Dmol.js Rego and Koes (2015).

The history of Penicillin gives a very good idea of the importance of the creation of medicines, which are basically molecules. But different from Penicillin, this design process is much more challenging. In general, the creation of new medicines is a long, rigorous and costly process to the pharmaceutical industry. It has been estimated that it can take more than 10 years to put a medicine into the market for the cost of billions of dollars

(LOIKE; MILLER, 2017), where only a few medicines are approved each year (Figure 1.2). These kinds of projects, which aim to produce new medicines, are surrounded by uncertainty and have high chances of failure. One of the most important phases of the drug discovery pipeline is the generation of molecules with desired chemical properties. The space of potential drug-like compounds, i.e., compounds that can be used in the design of medicines, has between 10^{23} and 10^{60} elements (POLISHCHUK; MADZHIDOV; VARNEK, 2013). Meanwhile only 10^{8} compounds have been synthesized by humans (KIM et al., 2016). The general process of generating molecules relies on expert knowledge and lab experiments. Recently, a new paradigm has emerged based on the use of Artificial Intelligence (AI).

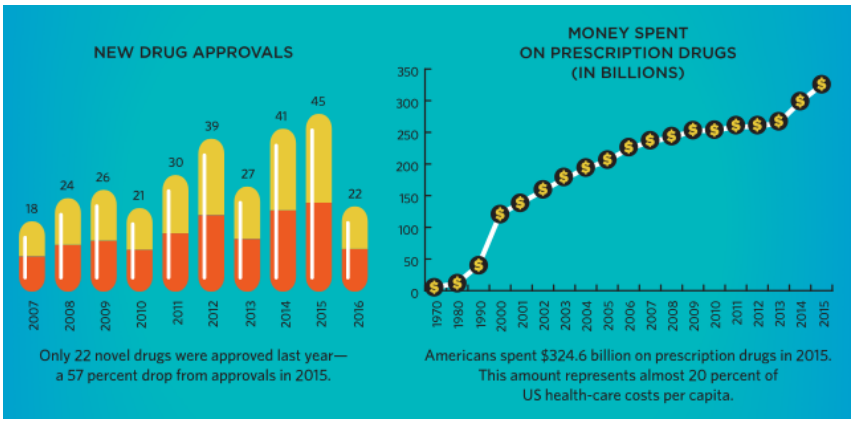


Figure 1.2: Drug discovery costs. Source: Loike and Miller (2017).

1.2 Artificial Intelligence

AI, and more specifically, Machine Learning (ML), has already been used to support a reasonable number of tasks in biochemistry, drug design and material sciences. For instance, by using ML scientists were able to make predictions about quantum properties of organic molecules much faster when compared to classic techniques such as Density Functional Theory (DFT), that are not only time consuming (Figure 1.3), but also expensive, and with ML it is possible to select a pool of candidates that can seem to be promising, and run more expensive experiments on those candidates. And more recently, a ML model called Alpha Fold 2 (JUMPER et al., 2021) achieved remarkable results for the protein structure prediction problem, regarded as one of the most important problems in biology.

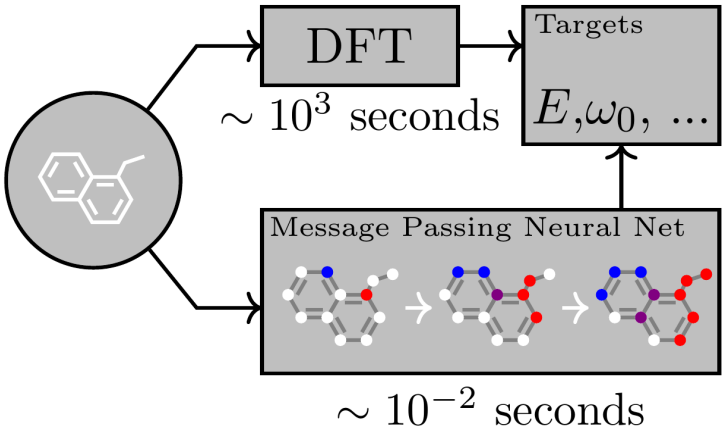


Figure 1.3: Time for the prediction of quantum properties of organic molecules using DFT and ML. Source: Gilmer et al. (2017).

In general, ML techniques have been proven a powerful domain for many tasks. And according to the Stanford AI index "Drugs, Cancer, Molecular, Drug Discovery" received the greatest amount of private AI investment in 2020, with more than USD 13.8 billion, 4.5 times higher than 2019 (ZHANG et al., 2021). Recently, improvements in deep generative modeling show that these methods can achieve remarkable results on molecular generation (ELTON et al., 2019). The generation of molecules using Machine Learning can be tackled using different techniques, one of the reasons behind this, is the fact that it is possible to represent molecules in different ways, and they are objects with rich information that may, or may not, be used in certain representations. These different design choices when dealing with the problem, require a lot of study in how to use such models for properly exploring this huge chemical space of molecules, and generate novel molecules with optimized properties.

ML is a field of Artificial Intelligence which aims to develop intelligent computational models through data. This paradigm differs from traditional approaches, where a study of the problem would be needed and a handcrafted solution would be proposed. With ML instead, a model is built and exposed to data, which serves as experience to the model to learn how to behave. In other words, we do not explicitly program the behaviour of the model.

Machine Learning can be classified according to its downstream task and how the available data is given. Some famous paradigms are Supervised Learning, Unsupervised Learning and Reinforcement Learning. In Supervised Learning we have a dataset with

inputs and outputs, the goal is to learn a mapping that given an input, maps it to the correct output. Unsupervised Learning is concerned with understanding the structure of the data. And Reinforcement Learning is concerned with how agents can take good decisions sequentially.

Deep Learning (DL) is a field of Machine Learning which uses a particular model, namely, Deep Neural Networks (DNNs), to learn from data. DNNs are models inspired by how the human brain works. They are composed of neurons, which are grouped in layers, the neurons from consecutive layers are connected with each other by an edge, which has an associated weight measuring the importance of that connection. Each neuron calculates an affine transformation, followed by the application of a nonlinear function, called activation function. This process is repeated layer by layer until we get an output for each input. Mathematically we can define a Deep Neural Network as a nonlinear function $f_\theta : \mathbb{R}^n \to \mathbb{R}^d$, parameterized by θ.

1.3 Objectives

The main goal of this work is to provide a study of how Deep Generative Models are being used in the task of generation and optimization of molecules. In order to achieve this goal, we specifically:

- Introduce how molecules can be represented in the computer, and the pros and cons of different representations.
- Give a formal introduction of statistical learning, including techniques and models that are important to achieve our goal.
- Formally introduce Deep Generative Modelling, making connections to the generation of molecules, particularly.
- Review the models that are considered the state-of-the-art for the task today, highlighting the pros and cons regarding different models and techniques.

2 Molecular Data

There are different ways to represent molecules computationally. Here we focused on string-based representations and graph-based representations. For strings, particularly, we will talk about the Simplified Molecular Input Line Entry Specification (SMILES) (WEININGER, 1988), which is not only the most used string-based representation for molecules, but the most famous representation in general. For graphs, there is not much variation, besides to represent the molecular graph, it is also possible to represent the molecular graph based on substructures of molecules, such as rings. Graph-based representations can also differ in how the incorporation of node and bond features are done for representing the molecular graph (Figure 2.1).

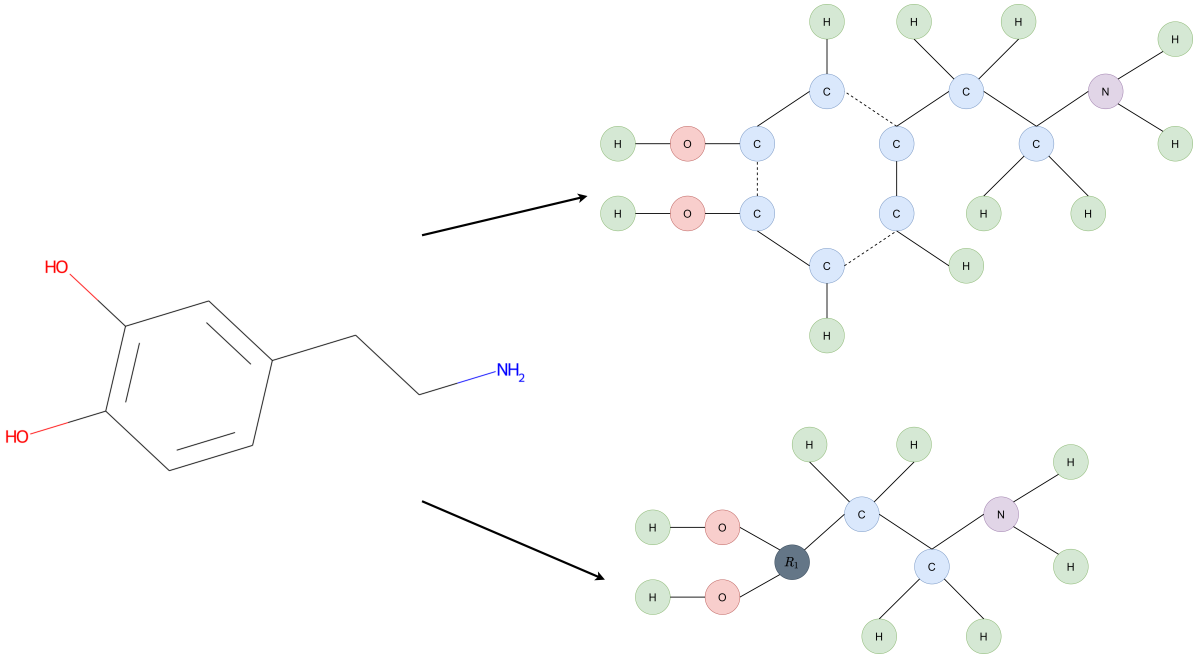


Figure 2.1: The dopamine molecule can be transformed to a regular graph (on the top) or a graph where rings are also nodes (on the bottom).

2.1 Molecular representation

The most used representation for digitally encoding molecules is SMILES. In this encoding, each molecule is simply represented as a sequence of tokens, which encodes the

molecular graph. It uses a line notation for describing the structure of chemical species using short ASCII strings. It is a very simple way to encode molecules, but it does not capture all chemical information present in a molecule. SMILES is a non-unique representation, although there exist algorithms that guarantee a canonical SMILES representation for each molecule, but they are invertible. In other words, one molecule can be represented by more than one SMILES string, but a SMILES string can represent only one molecule.

We call the different, but equivalent, strings that represent a molecule, an enumeration. In some scenarios it is desirable that a molecule has a standard SMILES representation. In this case many algorithms were proposed to provide canonical SMILES representations for molecules. For instance, the Toluene molecule (C_7H_8), has as canonical SMILES the string CC1=CC=CC=C1. But there are other equivalent ways to represent it as a valid SMILES, as shown in Figure 2.2.

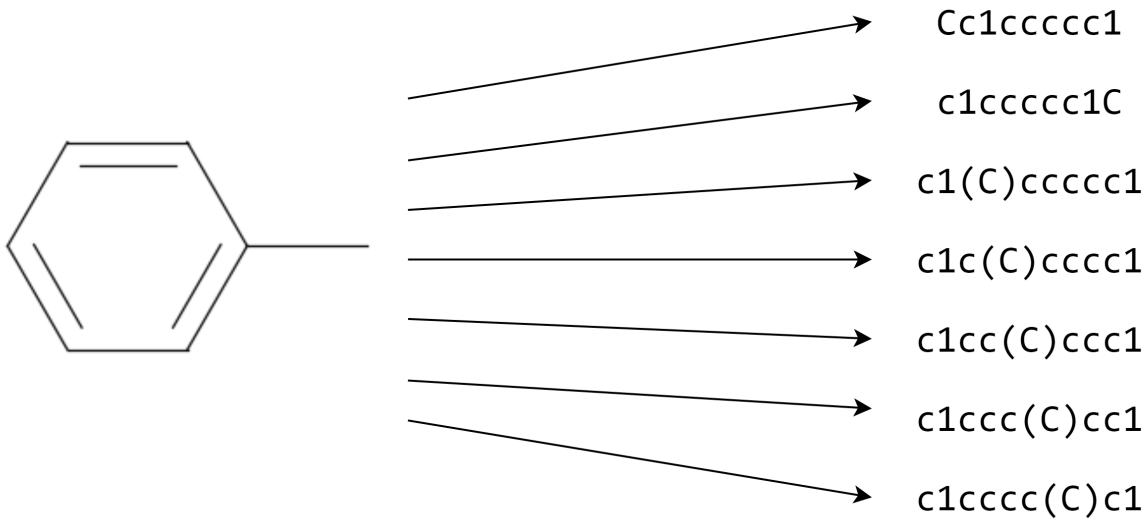


Figure 2.2: Some of the enumerations of the SMILES representing Toluene.

SMILES have also a formal specification of its rules, some of the basics of this notation include:

- Atoms are represented by their atomic symbols.
- Hydrogen atoms are omitted (are implicit).
- Neighboring atoms are represented next to each other.
- Double bonds are represented by =, triple bonds by #.

- Branches are represented by parentheses.
- Rings are represented by allocating digits to the two connecting ring atoms.
- Aromatic rings are indicated by lower-case letters.

Another problem with SMILES strings is that they do not directly capture the similarity between molecules. Two molecules may be very similar, chemically speaking, but still could have very different SMILES strings. As shown in Figure 2.3, we have two almost identical molecules that have very different canonical SMILES representations.

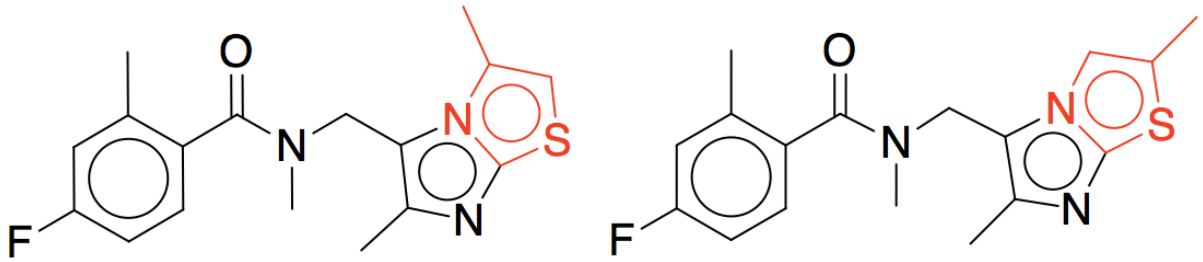


Figure 2.3: Similar molecules with very different canonical SMILES representation. Source: Jin, Barzilay and Jaakkola (2018).

Since SMILES are strings formed by tokens, Neural Language Models can be adapted and used in Machine Learning tasks with SMILES as inputs. When generating SMILES, we can also think about the set of rules that enables a string formed with a set of tokens to represent a valid molecule. In 2007, an open standard called OpenSMILES [1] was created, providing a Context-Free Grammar (CFG) which can be used to parse SMILES strings. A Context-Free Grammar (CFG) G is defined by the 4-tuple $G = (V, \Sigma, R, S)$ (SIPSER, 1996) where:

1. V is a finite set, where each element $\alpha \in V$ is called a variable,
2. Σ is a finite set, disjoint from V, where each element $\beta \in \Sigma$ is called a terminal,
3. R is a finite relation from V to $(V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the rules or productions of the grammar,

[1] http://opensmiles.org/spec/open-smiles-2-grammar.html

4. S is the start variable.

Some of the rules defined by OpenSMILES are shown in Figure 2.4, which was extracted from ⟨http://opensmiles.org/spec/open-smiles-2-grammar.html⟩.

Section	Formal Grammar
	ATOMS
3.1	atom ::= bracket_atom | aliphatic_organic | aromatic_organic | '*'
	ORGANIC SUBSET ATOMS
3.1.5	aliphatic_organic ::= 'B' | 'C' | 'N' | 'O' | 'S' | 'P' | 'F' | 'Cl' | 'Br' | 'I'
3.5	aromatic_organic ::= 'b' | 'c' | 'n' | 'o' | 's' | 'p'
	BRACKET ATOMS
3.1.1	bracket_atom ::= '[' isotope? symbol chiral? hcount? charge? class? ']'
3.1.1	symbol := element_symbols | aromatic_symbols | '*'
3.1.4	isotope ::= NUMBER
3.1.1	element_symbols ::=
	'H'| 'He'
	|'Li'|'Be'| 'B' |'C' |'N' |'O' |'F' |'Ne'
	|'Na'|'Mg'| 'Al'|'Si'|'P' |'S' |'Cl'|'Ar'
	|'K' |'Ca'|'Sc'|'Ti'|'V' |'Cr'|'Mn'|'Fe'|'Co'|'Ni'|'Cu'|'Zn'|'Ga'|'Ge'|'As'|'Se'|'Br'|'Kr'
	|'Rb'|'Sr'|'Y' |'Zr'|'Nb'|'Mo'|'Tc'|'Ru'|'Rh'|'Pd'|'Ag'|'Cd'|'In'|'Sn'|'Sb'|'Te'|'I' |'Xe'
	|'Cs'|'Ba'| 'Hf'|'Ta'|'W' |'Re'|'Os'|'Ir'|'Pt'|'Au'|'Hg'|'Tl'|'Pb'|'Bi'|'Po'|'At'|'Rn'
	|'Fr'|'Ra'| 'Rf'|'Db'|'Sg'|'Bh'|'Hs'|'Mt'|'Ds'|'Rg'
	|'La'|'Ce'|'Pr'|'Nd'|'Pm'|'Sm'|'Eu'|'Gd'|'Tb'|'Dy'|'Ho'|'Er'|'Tm'|'Yb'|'Lu'
	|'Ac'|'Th'|'Pa'|'U' |'Np'|'Pu'|'Am'|'Cm'|'Bk'|'Cf'|'Es'|'Fm'|'Md'|'No'|'Lr'

Figure 2.4: SMILES Grammar from OpenSMILES.

Therefore, we can also represent a SMILES string using the set of rules used to generate it. In Figure 2.5, we have the corresponding parse tree for the MIC molecule, using the OpenSMILES grammar.

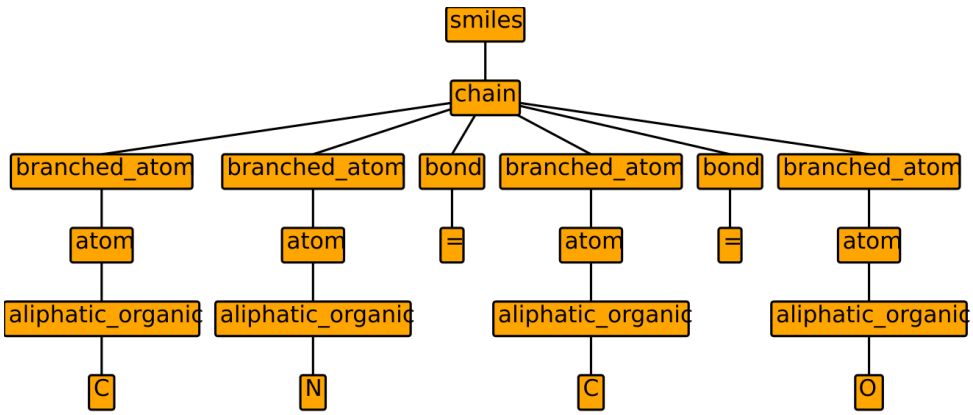


Figure 2.5: Parse tree of the MIC molecule.

2.2 Representation Learning

A very important concept in ML is representation learning, which refers to the process of learning good representations of the data in an automatic fashion. This central idea plays a very important role in the generation of molecules, and many other tasks in general. In this section, we briefly present what representation learning is and how we can learn representations using ML methods.

One very common class of problems are the so called classification problems. Where given a dataset $\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)} \right) \mid i = 1, \ldots, m \right\}$ with $x^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \{1, \ldots, C\}$. We call $x^{(i)}$ a feature vector of class $y^{(i)}$. The goal in this setting is to learn a function $f_\theta : \mathbb{R}^n \to \{1, \ldots, C\}$ parameterized by θ such that it minimizes a loss function.

Suppose we have the following dataset $\mathcal{D}$, with two different classes, namely, red class and blue class as shown in Figure 2.6. $\mathcal{D}$ is clearly non linearly separable, since we cannot find a line (linear model) that would correctly classify the data points, dividing the plane in two regions. In other words, the two regions defined by any line are not capable of separating the regions in a manner that most of the points belonging to the red class are in a different region than the points from blue class.

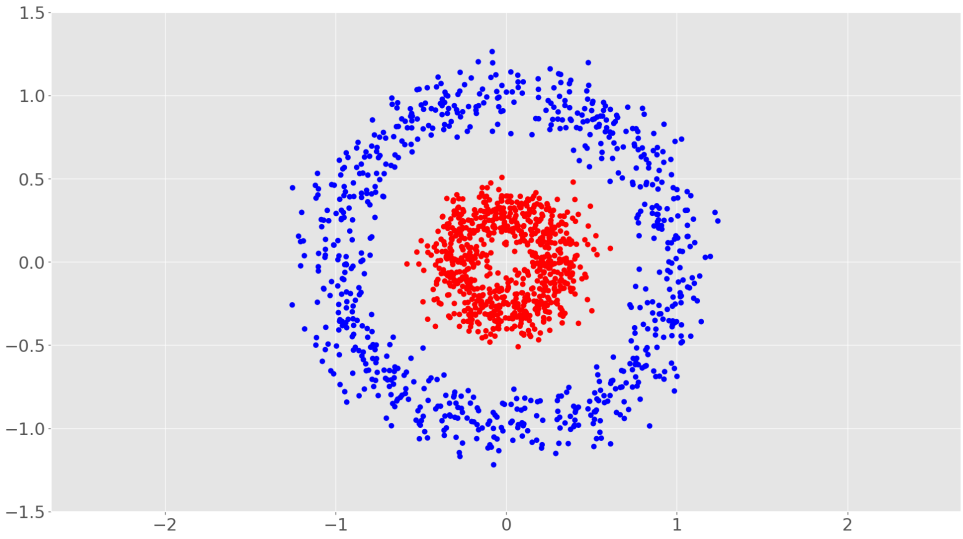


Figure 2.6: Non linearly separable dataset.

One solution to the problem would be to find a transformation $T : \mathbb{R}^2 \to \mathbb{R}^d$ such that the data is linearly separable in this space. We then obtain a new representation of the data $\hat{\mathcal{D}} = \{T(x) \mid x \in \mathcal{D}\}$, where we can use a hyperplane to classify the data. Consider

the transformation $T_1 : \mathbb{R}^2 \to \mathbb{R}^3$ such that $(x, y) \mapsto \left(x, y, \sqrt{x^2 + y^2}\right)$. After applying this transformation we end up with a representation in $\mathbb{R}^3$. This new representation can be easily separated by a plane (Figure 2.7).

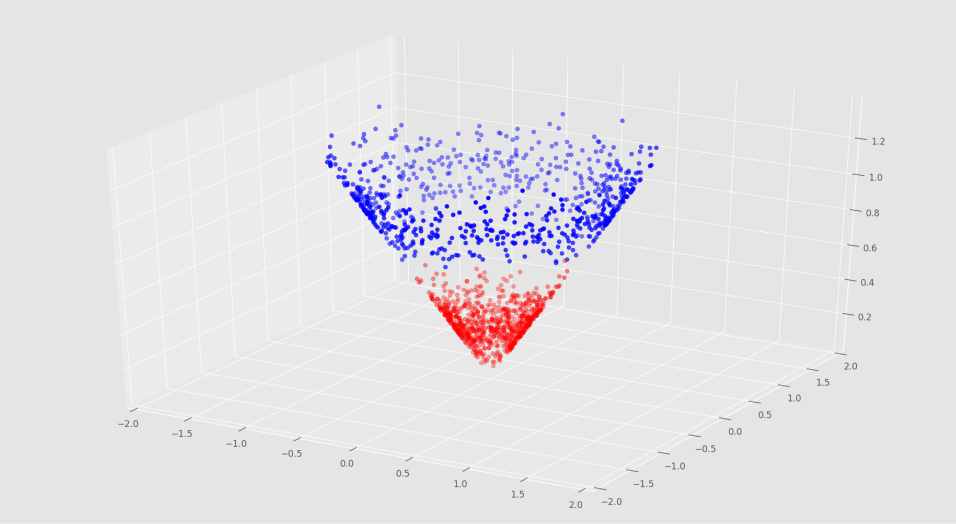


Figure 2.7: Representation obtained with the application of T_1.

But there are several transformations which would lead to useful representations of $\mathcal{D}$. It is not necessary to go to a three-dimensional space, indeed, using polar coordinates is enough to transform the data classification into a linear problem. So, we can consider the transformation $T_2 : \mathbb{R}^2 \to \mathbb{R}^2$ such that $(x, y) \mapsto (\theta, r)$, where $\theta = \arctan 2(y, x)$ and $r = \sqrt{x^2 + y^2}$. This transformation gives the representation seen in Figure 2.8.

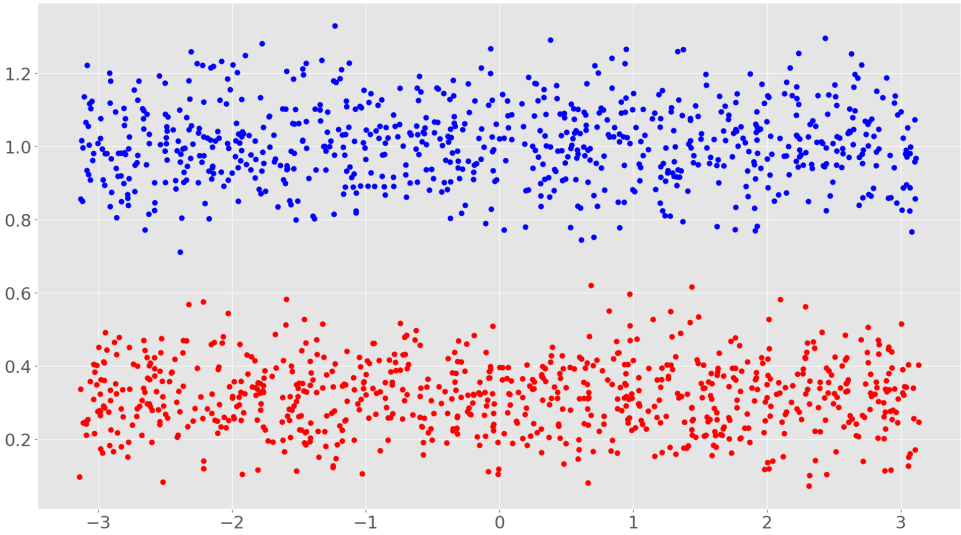


Figure 2.8: Representation obtained with the application of T_2.

But, we can also use the transformation $T_3 : \mathbb{R}^2 \rightarrow \mathbb{R}^1$ such that $(x, y) \mapsto \sqrt{x^2 + y^2}$. Hence, simply mapping the data points to a real number also gives a linear problem. In this new representation we just have to check if the result of transforming a point in a real number is greater than a threshold, as shown in Figure 2.9.

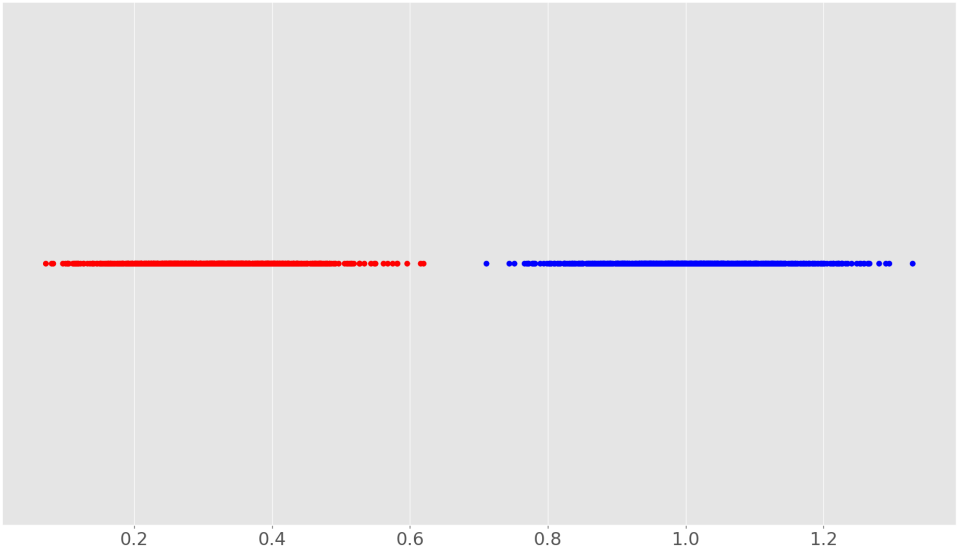


Figure 2.9: Representation obtained with the application of T_3.

In a glance, Deep Learning may be regarded as learning good representations of the data. These representations may be easier to work with, more interpretable and meaningful. Instead of handcrafted transformations, which require understanding of the problem and may be too application specific, we look to build models with learnable parameters and enough capacity to learn complex patterns from data.

Creating good representations will play a very important role in the specific problem of creating molecules with optimized properties. We will show how to obtain good representations for atoms, chemical bonds and for entire molecules using Representation Learning techniques.

3 Theoretical Foundation

In this chapter we provide the required foundations to understand the Machine Learning models that will be presented in this work. These models rely heavily on mathematical formulations that require careful construction. We start by talking a bit about one of the most important concepts in Statistical Learning, we then move to the section about Deep Learning, and we finish the chapter with Deep Generative Modelling (DGM), where we introduce Deep Learning models that can learn to generate new data, focusing on the generation of molecules.

3.1 Empirical risk minimization

In statistical learning, a very important principle is called empirical risk minimization (ERM) (VAPNIK, 1992). According to this principle, if we have a family of learning algorithms, we cannot know beforehand how they will perform exactly in practice, because we usually do not have access to the true data distribution. However, if we have access to a dataset, with samples coming from this unknown data distribution, we can measure the performance of the algorithm in this dataset. Hence, having an empirical measure of the true performance. For example, in a supervised learning setting, we have a data set:

$$\mathcal{D} = \left\{ \left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right) \mid \left(x^{(i)}, y^{(i)}\right) \in \mathcal{X} \times \mathcal{Y} \right\}, \tag{3.1}$$

with m training examples, where $x^{(i)} \in \mathbb{R}^n$ is the feature vector and $y^{(i)}$ is the desired output. $y^{(i)} \in \{1, \ldots, C\}$ in a classification problem and $y^{(i)} \in \mathbb{R}^k$ in a regression problem.

We have a function $f : \mathcal{X} \to \mathcal{Y}$, parameterized by θ, where θ is a vector of learnable parameters adjusted during training. A loss function for pairs ℓ that measures the error for a single data point $\left(x^{(i)}, y^{(i)}\right)$ given a hypothesis function f:

$$\ell\left(f\left(x^{(i)}; \theta\right), y^{(i)}\right). \tag{3.2}$$

A loss function $\mathcal{L}$ that gives an approximation of the overall error made by the model, and is dependent of the choice of parameters:

$$\mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell \left(f(x^{(i)}; \theta), y^{(i)} \right). \tag{3.3}$$

The ultimate goal of learning is the generalization, in other words, the ability to work well under unseen data. In a Supervised Learning setting, since defined a loss function that measures the difference between the model output and the true output, we can define the expected loss as:

$$R(f) = \mathbb{E}[\ell(f(x), y)] = \int \ell(f(x), y) dP(x, y). \tag{3.4}$$

Mathematically, we want to minimize this expected loss. In general this expression cannot be computed because the distribution $P(x, y)$ is unknown. Instead we can compute an approximation that averages the training set loss, the following expression is called empirical risk:

$$R_{\text{emp}}(f) = \frac{1}{m} \sum_{i=1}^{m} \ell \left(f\left(x^{(i)}; \theta\right), y^{(i)} \right). \tag{3.5}$$

But since the ultimate goal is to generalize, minimizing the model's error on examples that were shown to it is not a good strategy. A much more reliable approach is to use two separate datasets. $\mathcal{D}_{\text{train}}$ is used to train the model, i.e., adjust its parameters. $\mathcal{D}_{\text{test}}$ is used to test the model. In this manner, we may avoid that our model just memorizes the training set data without generalizing to unseen data.

This Supervised Learning setting is different from the ones that we will explore here, but it shows a common interface to other settings. In general we will be choosing from a set of models the one which outperforms the others in a given task.

3.2 Deep Learning

3.2.1 Multilayer Perceptron

Multilayer Perceptrons (MLPs) or Feedforward Neural Networks are one of the first and most basic Deep Learning models. They provide universal function approximators that can be learned from data. Although these models are slightly inspired by how the brain works, their goal is not to perfectly model the brain (GOODFELLOW; BENGIO; COURVILLE, 2016). Here we will assume that the input for a MLP is $x \in \mathbb{R}^n$.

MLPs are composed by a sequence of L layers, where each layer l has $n^{(l)}$ neurons, and neurons of consecutive layers are connected by a weight which measures the importance of the connection as shown in Figure 3.1.

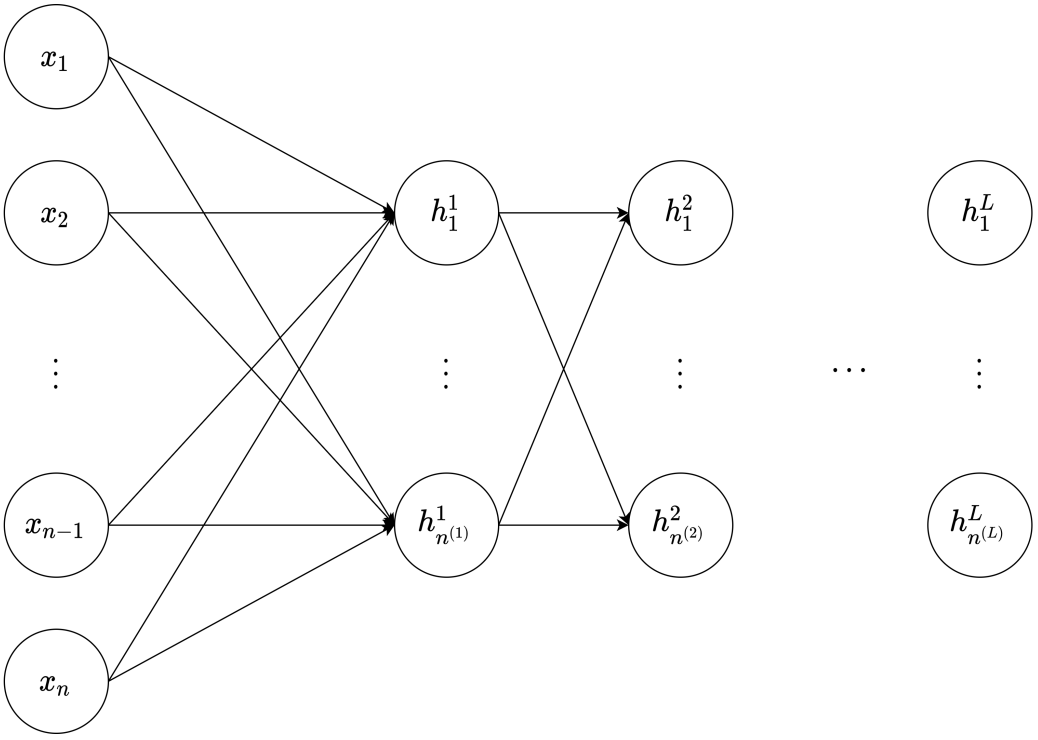


Figure 3.1: A general MLP architecture.

The term Deep Neural Network is usually used when a Feedforward Neural Network has more than 3 layers. A neuron in a layer l is a basic unit which is responsible for collecting information from all the neurons in layer $l-1$ and flowing it through the network. This is usually done by taking a dot product between the values computed by the neurons and the weights of the connection. A neuron computation is defined

mathematically by:

$$h_i^l = \sigma\left(w_i^{(l)} \cdot h^{l-1} + b_i^{(l)}\right), \tag{3.6}$$

where $h^{l-1} \in \mathbb{R}^{n^{(l-1)}}$ is a vector containing all the values computed by the neurons from layer $l-1$, $w_i^{(l)} \in \mathbb{R}^{n^{(l-1)}}$, $i = \left\{1, \ldots, n^{(l)}\right\}$ is the vector of weights related to neuron i of layer l, and $b_i^{(l)} \in \mathbb{R}$ is called the bias term. σ is a nonlinear function, often called an activation function. Without a nonlinearity the model would be limited to learn linear functions, because the composition of linear functions is also linear. Hence, the nonlinearity enables the model to learn complex functions. We call the set of parameters:

$$\left\{w_1^{(1)}, b_1^{(1)}, \ldots, w_n^{(1)}, b_n^{(1)}, \ldots, w_1^{(L)}, b_1^{(L)}, \ldots, w_{n^{(L)}}^{L}, b_{n^{(L)}}^{L}\right\}, \tag{3.7}$$

as the learnable parameters of the model, where we denote by θ the vector which contains all the parameters. θ is usually learned using an algorithm such as gradient descent or some more powerful variation, minimizing a loss function, such as the one defined in the previous section. This compositional nature of how neural networks are defined allows an efficient computation of the gradient using the backpropagation algorithm (RUMELHART; HINTON; WILLIAMS, 1986) which takes advantage of the chain rule.

3.2.2 Recurrent Neural Networks

MLPs are unable to directly deal with sequence-data, such as text and speech. One of the first Neural Networks designed to handle sequences do not differ significantly from the architectures seen previously. In fact, Recurrent Neural Networks (RNNs) (RUMELHART; HINTON; WILLIAMS, 1985), are a simple adaptation from MLPs which can work with sequences.

We can make a parallel about RNNs and dynamical systems. Dynamical systems are described by a state variable $s^{(t)}$, where $s^{(t)}$ is a vector and t denotes the time step. In other words, $s^{(t)}$ refers to the state of the system at the time step t. We control the dynamics of the system, using a function f. At each time step we apply the function f to the current state of the system obtaining a new state, as seen in Figure 3.2. In other words $s^{(t+1)} = f(s^{(t)})$.

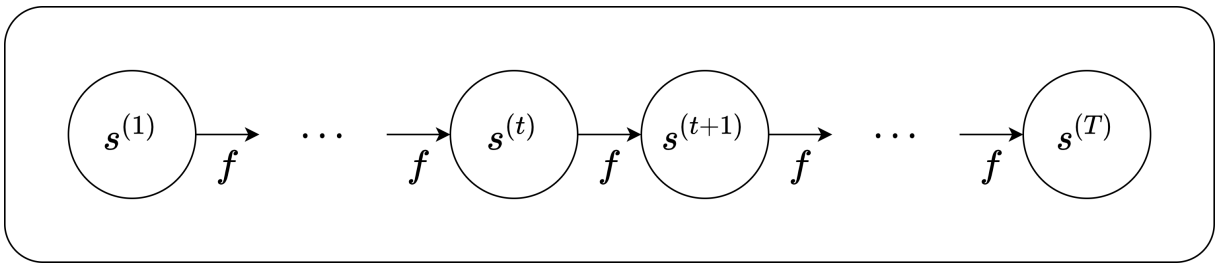


Figure 3.2: Example of a dynamical system.

When we train MLPs, we have a function f_θ with learnable parameters θ, which maps an input $x \in \mathbb{R}^n$ to an output $y = f_\theta(x) \in \mathbb{R}^k$. Adapting this very common architecture to work as a dynamical system is pretty straightforward. Suppose that now the input x is a finite sequence $x = \left(x^{(1)}, \ldots, x^{(T)}\right)$ and the output is also a sequence of same size $y = \left(y^{(1)}, \ldots, y^{(T)}\right)$, where T is the length of the sequence and each element of the sequence is a vector. We can apply an adapted version of a MLP, which we will call a RNN cell, in this sequence, similarly as the function f is applied in a dynamical system. A RNN cell receives the previous state of the system and an element of the sequence, and produces an output and a new state of the system. This procedure is repeated until all the elements of the sequence are processed.

To keep track of information seen in previous time steps, RNNs use hidden states. The hidden states describe the state of the system in the current step, with the hidden representations $h^{(t)}$, where the model learns how to store some of the information from the previous states in a single vector. Hidden states incorporate the correlation between states at different time steps, but RNNs do not explicitly model the relationship between all the time steps, instead, only modeling directly the relationship between time steps t and $t-1$.

We can imagine that we are applying the same MLP at each time step. As we will see later, the input and output do not need to have the same size, which would require adapting the architecture described before. For example, if we consider the natural language translation problem, we have pair of sentences, one in the source language and one in the target language, that we can denote by $x = \left(x^{(1)}, \ldots, x^{(T_x)}\right)$ and $y = \left(y^{(1)}, \ldots, x^{(T_y)}\right)$, respectively. It is easy to see that the same sentence in different languages may not have the same size, therefore T_x and T_y are not necessarily equal. We also have the sentiment

analysis problem, where given for example, a movie review, we could classify it as being positive or negative. Again, the input x is a sequence of words $x = \left(x^{(1)}, \ldots, x^{(T_x)}\right)$, but now we have a unique output $y \in \{0, 1\}$, indicating if the movie review was positive or negative.

A general framework of a RNN, where the input sequence and output sequence have the same length, is shown in Figure 3.3.

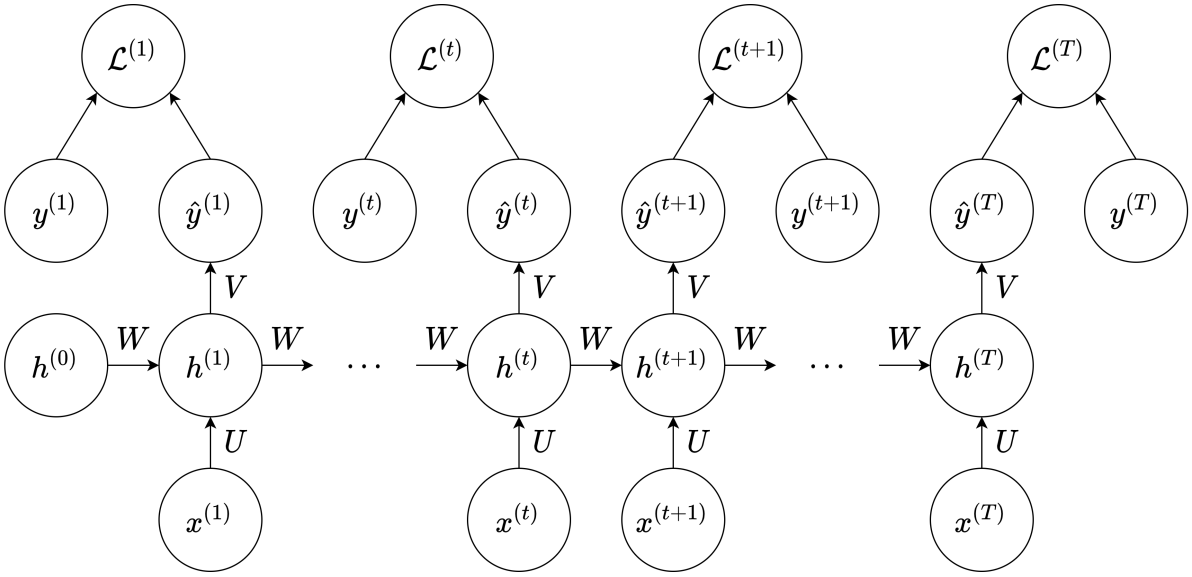


Figure 3.3: The general framework of a RNN with inputs and outputs of same dimension.

The following equations describe the process of flowing information through the network for this case:

$$\begin{aligned} a^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh\left(a^{(t)}\right) \\ o^{(t)} &= Vh^{(t)} + c \\ \hat{y}^{(t)} &= \text{softmax}\left(o^{(t)}\right) \end{aligned} \tag{3.8}$$

In RNNs we also explore the concept of weight sharing. The same weight matrix U is used to transform the input $x^{(t)}$ for all time steps. Analogously, we have the weight matrix W which will transform all the hidden states $h^{(t)}$ to propagate the message to the next hidden state and the matrix V which will also transform the hidden states $h^{(t)}$, but in order to obtain a final prediction $\hat{y}^{(t)}$, which will be compared to the ground truth $y^{(t)}$ using a loss function $\mathcal{L}$. The training of RNNs is done with Backpropagation Through Time

(WERBOS, 1990), which is basically the original Backpropagation algorithm adapted to deal with time dependencies between different time steps.

For generating sequences we could train an RNN in an auto-regressive way. For instance, given a sequence $y = \left(y^{(1)}, \ldots, y^{(T)}\right)$, we would train the model to predict one element of the sequence at time. We start with a special start of the sequence (SOS) token as $x^{(1)}$. The model then produces its first output, $\hat{y}^{(1)}$, which is a probability distribution over all the tokens. We then sample a token from $\hat{y}^{(1)}$, which is passed as input to the model in the next time step, in other words $x^{(2)} \sim \hat{y}^{(1)}$. This process of sampling $x^{(i+1)} \sim \hat{y}^{(i)}$ is repeated until a maximum size of sequence is reached or the end of sequence (EOS) token is sampled.

This approach has several problems that could make the training challenging. For instance, since the model's input at time step $i+1$ is the output in time step i, if the model predicts the wrong token, the errors will rapidly be propagated during training, making this process very unstable. To overcome such problems Williams and Zipser (1989) proposed a method called teacher forcing. With this technique, instead of giving the model's output as input in the next time step during training, we give the model the correct token, even if the model predicted the wrong one. A simple example of training a RNN to generate SMILES strings is shown in Figure 3.4. Other techniques such as schedule sampling (BENGIO et al., 2015) and professor forcing (LAMB et al., 2016) were also proposed, but teacher forcing seems to be the predominant one.

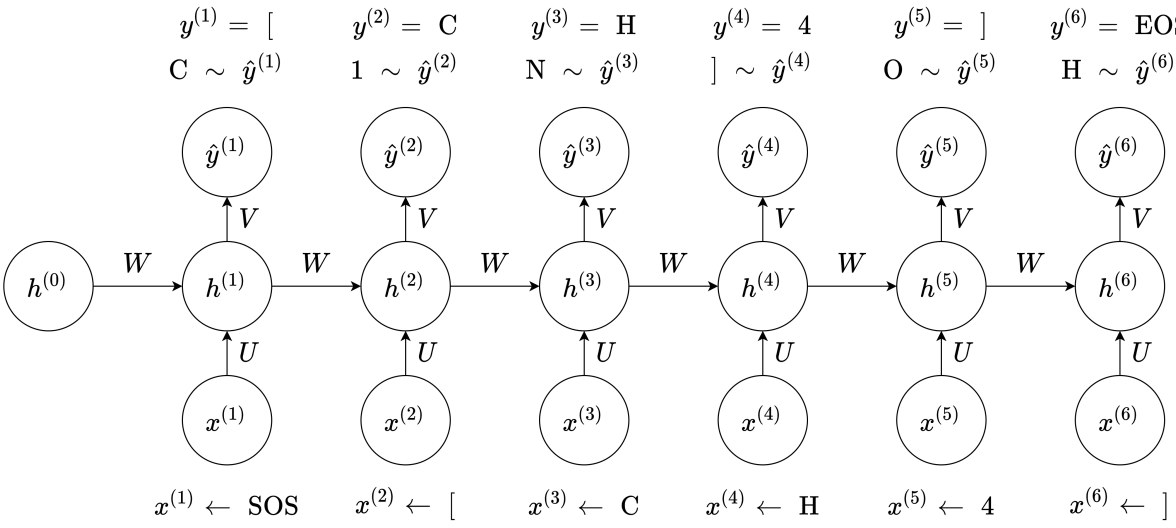


Figure 3.4: Teacher forcing during training for the methane SMILES.

One issue with RNNs is related to their compositional nature, where if we have a sequence of size T it is necessary to perform matrix operations with the same weight matrix, T times. This can lead to a famous problem of RNNs, namely, the gradient exploding and the gradient vanishing (HOCHREITER, 1998). The former is related to gradients becoming too big, and therefore the updates using the gradients affect the weights of the network too much. With the latter, the gradients become vanishingly small, almost not affecting the weights of the network during the updates.

To tackle this problem, different modifications in the original RNN architecture were proposed. Long short-term memory (LSTM) (HOCHREITER; SCHMIDHUBER, 1997) and Gated recurrent unit (GRU) (CHO et al., 2014) are newer architectures that are less prone to suffer from the problems of gradient exploding and vanishing.

3.2.3 Graph Neural Networks

Graphs are a very famous representation for the interaction between objects, with application ranging from social sciences to the natural sciences. Formally, the most basic way to define a Graph is as a tuple, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which is composed by a set vertices (or nodes) $\mathcal{V}$, and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. More flexible notions of graphs could include, for example, node and edge features. This will be the definition of graphs that we will use here, since atom and bond features are vital to represent a molecule with fidelity. We call $\mathcal{N}(u) = \{(u, v) \mid (u, v) \in \mathcal{E}\}$ the neighborhood of u, and $v \in \mathcal{N}(u)$ a neighbor of the node u.

Although DL techniques are not the only way to handle graph tasks, with the growing number of datasets with graph-structured data, it is necessary to adapt DL techniques to work with graphs as well. One problem is that most of the DL pipeline was developed to work with specific types of data, such as images, tabular data and sequences. This pipeline did not include graphs for some time, and the reason is that graphs are more flexible (and discrete) objects, and hence, more challenging to adapt classic techniques to work with. But since many important tasks have graph-structured data, DL techniques have been developed to be applied to graphs.

One particularly interesting task is node classification. Where, given a graph $\mathcal{G}$,

partially labeled, we would like to predict the missing labels of the nodes in the graph. If we consider that each node u has a feature vector $x_u \in \mathbb{R}^n$, we could build, for example, a MLP, which receives an input x_u, and outputs the class probabilities associated with the node features. The problem with this approach is that, differently from tabular data, the nodes in the graph are correlated. Hence, if we do not incorporate the correlations of nodes, such as neighborhood information, the classifier may achieve poor performance.

Initially, we will assume our goal is to learn a function $f_\theta : \mathcal{V} \mapsto \mathbb{R}^d$. The function f_θ learns how to represent nodes in a latent (and low-dimensional) space, summarizing important information of the node and preserving the neighborhood structure present in the graph. In general, we can obtain representations (often referred as embeddings) for edges and graphs as well. The representation obtained $z_u = f_\theta(u)$ should allow a direct computation of metrics such as node similarity between two nodes. For instance, if we want to compare the similarity between two nodes u and v, we could simply take the dot product $z_u \cdot z_v$, and use it as a similarity measure.

As shown in Figure 3.5, we use the function $f_\mathcal{V} : \mathcal{V} \to \mathbb{R}^{d_\mathcal{V}}$ to map each atom $v \in \mathcal{V}$, to their corresponding representation $z_v = f_\mathcal{V}(v)$ in an atoms embedding space $\mathbb{R}^{d_\mathcal{V}}$ shown in purple. Similarly, we apply the function $f_\mathcal{E} : \mathcal{E} \to \mathbb{R}^{d_\mathcal{E}}$ to map each chemical bond $e \in \mathcal{E}$ to an bonds embedding space shown in green.

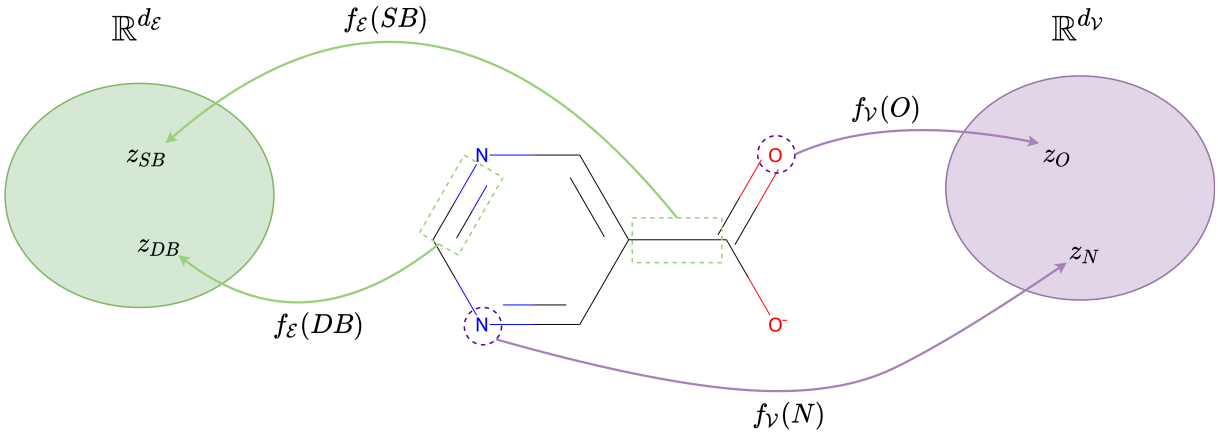


Figure 3.5: Mapping atoms and bonds to low-dimensional representations.

A simple approach to capture neighborhood similarly for a node u is to compute statistics about the nodes that are topologically close to u. We could do this for example, by walking over the graph until a maximum number of steps and starting from u. Deep

Walk (PEROZZI; AL-RFOU; SKIENA, 2014) is a representation learning method for nodes in graphs, which uses Random Walks to capture neighborhood similarity and community membership. Given a node u_i, we perform a Random Walk of size n, i.e., we visit a sequence of nodes $\{u_{i_1}, \ldots, u_{i_n}\}$, starting from node u_i (possible repeating nodes). This multiset of nodes is then used as a similarity measure. Nodes that frequently co-occur with u_i in a Random Walk of size n are in some sense more similar to it. We use this assumption to define the learning goal in this framework, we maximize the probability:

$$P\left(u_i | u_{i_1}, \ldots, u_{i_n}\right). \tag{3.9}$$

After obtaining, for each node u, the multiset $\mathcal{N}_R(u)$, containing all the nodes visited during the Random Walk, we can maximize, with respect the embeddings, the probability of co-occurrence of u and the nodes in $\mathcal{N}_R(u)$. In other words, we directly learn the representations z_u for each node u in the graph. This corresponds to:

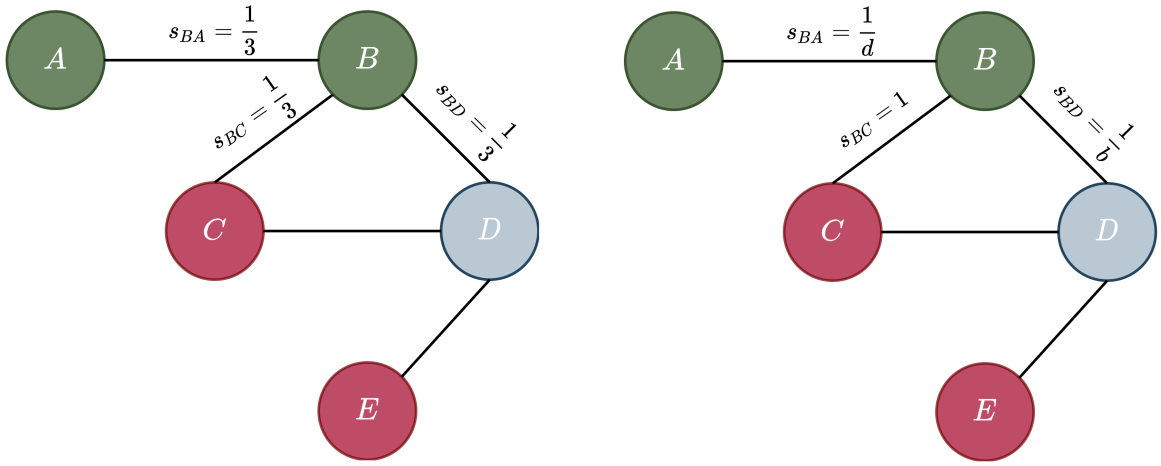
$$\max \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{N}_R(u)} P(v \,|\, u), \tag{3.10}$$

where:

$$P(v \,|\, u) = \frac{\exp(z_u \cdot z_v)}{\sum_{w \in \mathcal{V}} \exp(z_u \cdot z_w)}. \tag{3.11}$$

Another approach, based on Random Walks, which allows learning node embeddings is Node2Vec (GROVER; LESKOVEC, 2016). This method uses biased Random Walks, where each movement from a node u to one of its neighbors has a different effect. For instance, a Random Walk starting at a node u can move to a node v, and stay in the same distance to u, move farther away from u, or get closer to u. In Figure 3.6, for example, going from node B to A has the effect of moving farther away from the root node E. Moving to C has the effect of staying in the same distance. And if we go to D we are getting closer to E.

Although Random Walks based methods have achieved success in the past for some tasks, these methods have several limitations. For instance, neither DeepWalk, nor Node2Vec, consider node feature information in order to obtain node embeddings. These



(a) Deep Walk. (b) Node2Vec.

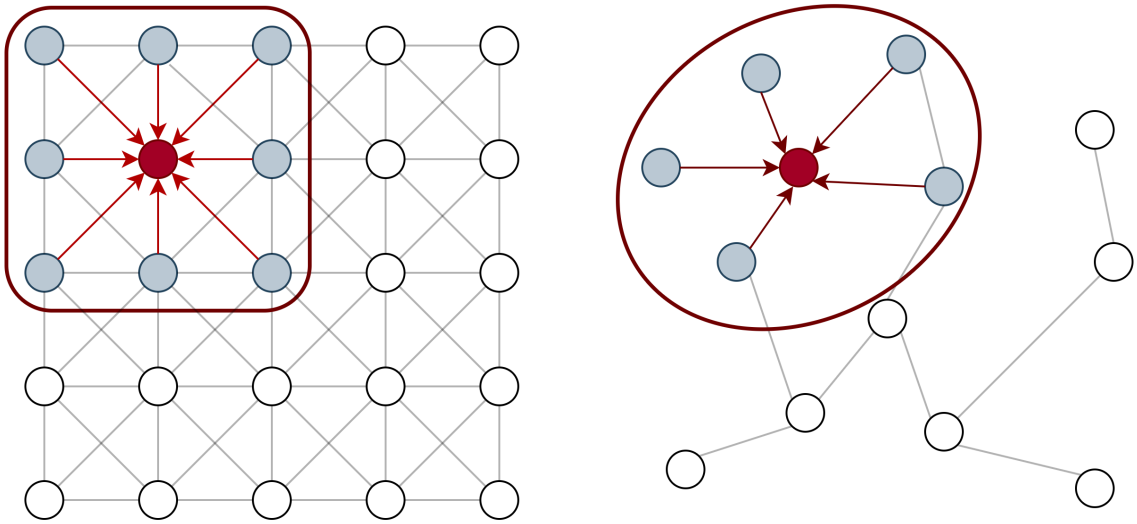
Figure 3.6: Random Walks on Graphs.

methods are also not directly applicable to new nodes, since there is no parameter sharing. In other words, if nodes with unseen labels are added to the graph, there is no function for obtaining the embeddings for these nodes, therefore, it would be needed to retrain the network.

A very famous neural network architecture, which works especially well for natural signals is the Convolutional Neural Network (CNN) (FUKUSHIMA; MIYAKE, 1982). Natural signals present a series of properties, which allow the creation of architectures targeted at them. Three of the main properties present in natural signals Canziani (2020) are stationarity, where certain patterns are prone to repeat throughout the signal, locality, nearby points are more related to each other, if compared to points who are far away from them, and compositionality, a signal can be decomposed in subparts which form the whole signal. CNNs explores these properties, using weight sharing, sparsity and stacked layers. Differently from a MLP, for instance, the same weights are used to calculate representations for different parts of the signal. And instead of fully connecting two consecutive layers in the network, only some nodes are connected by a weight. Moreover, CNNs are usually composed of more than one convolutional layer.

This idea to apply the convolution operation was adapted for graphs as well. In Figure 3.7, we have an interpretation for the convolution operation on an image, where a sliding filter is convolved with different parts of the image. This operation was adapted to graphs in the so-called Graph Convolutional Networks (GCNs) (KIPF; WELLING,

2016a).



(a) Convolution in an image. (b) Convolution in a graph.

Figure 3.7: Convolution Generalization.

Different from images, graphs have no ordering, and arbitrary structure. The idea is that neighbors of a node u influence predictions related to it, hence they should be incorporated in the network propagation function. GCNs use the following update rule for a node u:

$$h_u^{l+1} = \sigma\left(W_0 h_u^l + \sum_{v \in \mathcal{N}(u)} W_1 h_v^l\right). \tag{3.12}$$

GCNs are one of the many types of Graph Neural Networks (GNNs). GNNs offer a general framework for applying DL on graphs (HAMILTON, 2020), using both node features information, and the structure of the graph in the neighborhood of a node, to obtain embeddings. One of the main differences between GNNs and the methods that we have seen before, is that they build a computational graph for each node, which is defined by its k-hop neighborhood, i.e., all the nodes that have distance less or equal than k from the node. In Figure 3.8, we create the computational graph for node A using a 2-hop neighborhood. The functions ϕ_i are used to aggregate information from neighbors of a node, and the functions σ_i are used to combine the information of a node with the information from its neighbors.

Using this computational graph, we obtain a representation for A which is dependent on A features and its neighborhood structure. During training, it is possible to

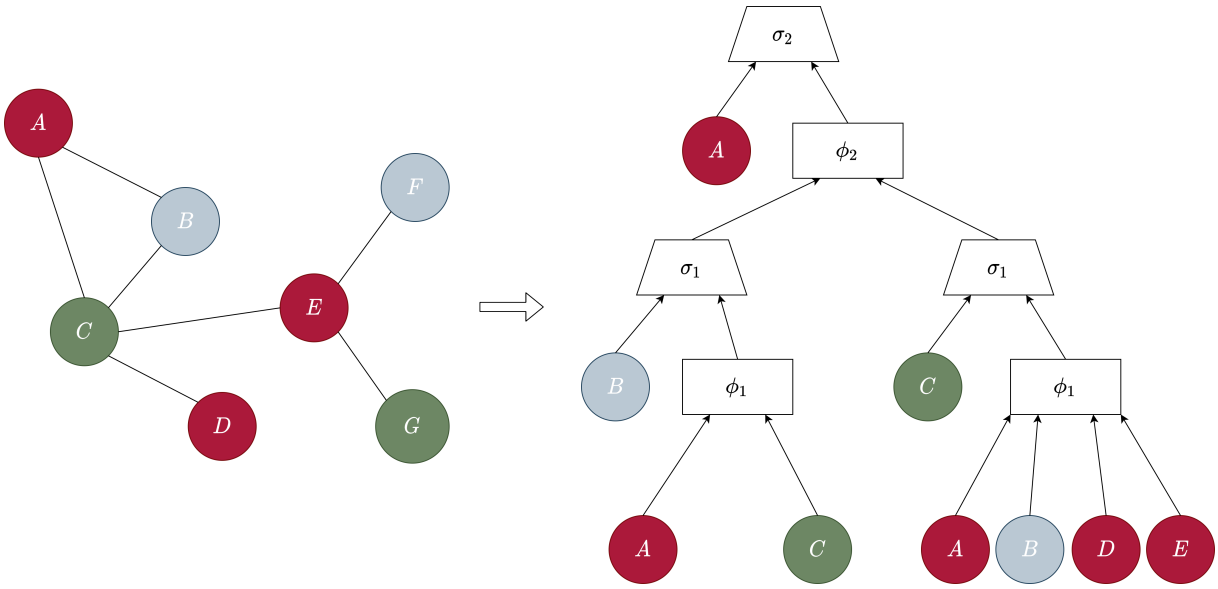


Figure 3.8: Computational Graph.

obtain embeddings optimized to a chosen task, so for instance, one could train a GNN to predict chemical properties of a molecular graph. The effect on the embedding would be creating a correlation between the embeddings and the downstream task. Another advantage of this type of approach is the possibility to generate embeddings to unseen types of nodes. Because now we have a function from both node features and the neighborhood structure.

In GNNs we assume that messages are exchanged between nodes and their neighbors. In this assumption a node's label, for example, depends on its features and the relationship with its neighbors. During each message-passing iteration in a GNN, a hidden embedding $h_u^{(k)}$ corresponding to each node $u \in \mathcal{V}$ is updated according to information aggregated from u's graph neighborhood $\mathcal{N}(u)$. This message-passing update can be expressed as follows:

$$\begin{aligned} h_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(h_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}\right)\right) \\ &= \text{UPDATE}^{(k)}\left(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)}\right). \end{aligned} \tag{3.13}$$

The AGGREGATE function computes a representation (message) $m_{\mathcal{N}(u)}^{(k)}$ using the neighbors of a node u and the UPDATE function combines the representation of the node u with the representation $m_{\mathcal{N}(u)}^{(k)}$ from u' neighbors. If we have K iterations, the last

update will correspond to the node embedding:

$$z_u = h_u^{(K)} \ , \ \forall u \in \mathcal{V}. \tag{3.14}$$

3.2.4 Reinforcement Learning

Learning through interaction is one of the most natural ways that one would think a human can learn. Humans can perform actions in environments and observe the effects of those actions. For some environments we have more control over them, having a good idea of what happens during the interaction. For others, we would need many interactions to simply have a feedback. For some problems it is necessary to carefully plan the actions to be taken in order to achieve a certain goal. This idea of learn through interaction is incorporated in Reinforcement Learning (RL), a learning paradigm which is often put as one of the three basics ML paradigms, besides Unsupervised Learning and Supervised Learning. In this section, we give an introduction to RL and make connections to how RL can be applied to the task of generating molecules. For a broader overview of the field, refer to Sutton and Barto (2018).

Reinforcement Learning is concerned with how an intelligent agent can learn to make good decisions sequentially, and under uncertainty. And the uncertainty comes naturally, since the agent does not know beforehand what the consequences of their actions are. It involves optimization, delayed consequences, exploration and generalization. In this specific scenario, we will talk about a RL agent that will have to produce new molecules and/or optimize them.

When we talk about optimization, we can refer, for instance, to optimize some chemical properties, such as drug-likeness of a molecule. This leads to another important concept, namely delayed consequences. The agent may only be certain about how good the molecule being generated will be, in the last step, when the generation is finished. During the intermediate steps the model can have only a rough idea of its performance so far. The model has to learn how the decisions made in the past affect the outcome, which can be very challenging. The exploration refers to the process of learning where the agent will learn by making decisions and observing the consequences.

This paradigm is similar at some level to how humans learn. To achieve a goal we perform a sequence of actions towards that goal, after each action we may have an idea of how good that action was, but only at the end we will be certain if this sequence of actions lead to the goal or not. In the basic RL setting we have an agent that can take actions in the world and observe the consequences to those actions through a reward signal. The goal of the agent is to maximize the expected future reward. One of the key challenges is to find the balance between getting immediate and long term rewards. In other words, taking the sequence of actions that yields the highest immediate reward may not lead to the highest possible future reward. Hence it is necessary to plan the actions towards both the immediate and long term rewards.

More formally, in the general setting, the agent starts from a state S_0. A state describes the current setting of the environment. The agent then can take an action A_0, receiving a reward R_1, and moving to a new state S_1 afterwards. In episodic tasks, the agent will continue to take actions which will lead to consequences until a maximum time step is reached. In other words, one episode is defined as a sequence:

$$(S_0, A_0, R_1), (S_1, A_1, R_2), \ldots, (S_{T-1}, A_{T-1}, R_T). \tag{3.15}$$

For the molecular generation task S_0 could be either an empty molecule or a starting molecule which the model wants to optimize, and S_t, with $t > 1$ the molecule generated so far. The actions A_t could be for instance the addition and removal of atoms and bonds, and the reward R_t could be defined in different ways, such as the chemical properties of a generated molecule or even if the molecule generated after taking action A_t is valid. The goal of a RL agent is to maximize the expected discounted cumulative reward $\mathbb{E}[G_t]$, where:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.16}$$

and $\gamma \in [0, 1]$. In other words, the agent wants to get as many positive signals as possible. But in a time step t, the agent will usually be more concerned with rewards of interactions closer in time, rather than interactions in a far future. With this in mind we can create approaches so that the agent can learn how to behave in an environment. In order to do

this, we need to be able to measure the quality of decisions that an agent may take, or what an agent should expect when it is at state S_t.

We model the process used by the agent to select actions in a state as a mapping π, called policy. A deterministic policy is a mapping $\pi : \mathcal{S} \to \mathcal{A}$. Where for each state $s \in \mathcal{S}$, it yields the action $a \in \mathcal{A}$ that the agent will choose while in state s. A stochastic policy is a mapping $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. Where for each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, it yields the probability $\pi(a|s)$ that the agent will choose action a while in state s.

The goal of a RL agent is to learn how to behave in an environment, which can be formally defined as learning the optimal policy π, which maximizes the expected discounted cumulative reward $\mathbb{E}[G_t]$. In order to do this, we need a utility function which can be used to measure how good a certain action, or a sequence of actions are, in a given state.

The state-value function for a policy π is denoted by v_π. For each state $s \in \mathcal{S}$, it yields the expected return if the agent starts in state s and then uses the policy to choose its actions for all time steps. That is:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]. \tag{3.17}$$

We refer to $v_\pi(s)$ as the value of state s under the policy π. All optimal policies have the same state-value function v_*, called the optimal state-value function.

The action-value function for a policy π is denoted by q_π. For each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, it yields the expected return if the agent starts in state s, takes action a, and then follows the policy ϕ for all future time steps. That is:

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]. \tag{3.18}$$

We refer to $q_\pi(s, a)$ as the value of taking action a in state s under policy π (or alternatively as the value of the state-action pair (s, a)). All optimal policies have the same action-value function q_*, called the optimal action-value function.

Evaluating the state-value function or the action-value function is important because it can be used to find an optimal policy. Once the agent determines the optimal

action-value function q_*, it can quickly obtain an optimal policy π_* by setting:

$$\pi = \underset{a \in \mathcal{A}(s)}{\text{argmax}} q(s, a). \tag{3.19}$$

Many problems can be modeled as a finite Markov Decision Process (MDP), which is defined by a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where:

1. $\mathcal{S}$ is a finite set of states,
2. $\mathcal{A}(s)$ is a finite set of actions available at state s,
3. $\mathcal{R}$ is a set of rewards,
4. $p(s', r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$ is the one-step dynamics. The one-step dynamics of the environment determine how the environment decides the state and reward at every time step.

MDPs satisfy the so-called Markov Property, in other words, the one-step dynamics of the environment only uses information about the current time step t to decide what happens next, at time step $t + 1$, but none of the information from time steps $1, \ldots, t - 1$ are used to make decisions.

3.3 Deep Generative Modelling

Given a dataset $\mathcal{X} = \left\{x^{(1)}, \ldots, x^{(m)}\right\}$, where $x^{(i)} \in \mathbb{R}^n$, with (unknown) probability distribution p_{data}, which explains the data generating process. A generative model is an, implicit or explicit, approximation of p_{data}. There are several ways to learn a generative model from data, the most successful ones are the parametrics. With parametric models we approximate p_{data} by p_θ, where θ is a finite-dimensional vector of learnable parameters.

A good approximation of a complex distribution, such as distributions over images and videos, may allow the generation of new samples which are similar to the ones contained in the dataset $\mathcal{X}$. In general, the goal of approximating p_{data} consists in an optimization problem. We need to find parameters θ that minimize a form of distance

(such metric will be defined later) between two probability distributions, in other words:

$$\theta^* = \min_{\theta} d(p_{\text{data}}, p_\theta). \tag{3.20}$$

The most used metrics to measure how dissimilar two distributions are, is the divergence. This is a more general concept than distances. For instance, the divergence between two distributions may be not symmetric and it does not have to satisfy the triangle inequality. Formally, if S is the space of the probability distributions with common support. Then the divergence in S is a function $D(\cdot||\cdot) : S \times S \to \mathbb{R}$ which satisfies:

1. $D(p||q) \geq 0 \ \forall \ p, q \in S$,
2. $D(p||q) = 0 \iff p = q$.

Several divergence functions have been proposed to model problems. One of the most used ones is known as Kullback-Leibler divergence, or KL divergence for short. Given the distributions q and p, the KL divergence, denoted by $D_{KL}(p||q)$, measures the relative amount of information lost when we use q to approximate p, and is given by the following formulas:

1. For discrete probability distributions, the KL divergence from q to p, defined on $\mathcal{X}$ is:
$$D_{KL}(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{p(x)}{q(x)} \right). \tag{3.21}$$
2. For continuous distributions we have:
$$D_{KL}(p||q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx. \tag{3.22}$$

In the real world, there are several phenomena that are not directly measurable or observable, but instead, inferred through other observable phenomena, using a mathematical model. For instance, when the picture of a person is taken, using a certain camera, we use pixels as measuring units. In this analogy, the pixels are the observed variables. But there is information which the two-dimensional array of pixels does not directly tell, but can be inferred from these numbers. For example, there is information

about the gender or age of the person in the picture. Although this information is not present in the picture, the pixels can be used to infer them.

To these variables which are not directly observed, but inferred we give the name of latent variables. They are extremely important in different fields, including in social sciences, economics and physics. The central idea is that behind a perceived complex data generation process, there are simpler hidden explanations, which can be used for modeling purposes. These variables may correspond to abstract concepts or be unfeasible to directly measure.

As seen in Figure 3.9, the two images contain basically the same semantic meaning, although the first one has higher resolution, and therefore is more clear than the second one. Figure 3.9a is a representation of size $1024 \times 1024 \times 3 \approx 3 \times 10^6$ and Figure 3.9b is a representation of size $128 \times 128 \times 3 \approx 5 \times 10^4$. They both represent, basically, the same information, but they use a very discrepant amount of information. The assumption here is that there is a low-dimensional representation which can capture the essence of the process.

(a) 1024×1024. (b) 128×128.

Figure 3.9: Two images containing the same semantic meaning. Source: Karras, Laine and Aila (2019).

We can assume that our dataset $\mathcal{X}$ has a hidden structure. We have the random (latent) vector z related to this hidden structure, and the random vector x, related to the observed variables. We then have a generative process, where we first obtain an explanation $z \sim p(z)$ and then we generate an observation $x \sim p(x|z)$.

3.3.1 Variational Autoencoders

One of the first successful deep generative models was the Variational Autoencoder (VAE) (KINGMA; WELLING, 2013). The basic idea of VAEs is to map an input data point x to a distribution, where is possible to sample a latent vector z, which can be decoded in x again. This is similar to the vanilla Autoencoders (RUMELHART; HINTON; WILLIAMS, 1986), where the main difference is that instead of mapping x deterministically, because we use a stochastic process. Intuitively, this helps to fill the latent space, since a same input x may lead to a different sample z, since we have a stochastic process. Therefore, latent vectors, which are close to each other, will be decoded in x which are also similar. This framework is described by a prior $p_\theta(z)$, a posterior $p_\theta(z|x)$ and the likelihood $p_\theta(x|z)$, where θ is a set of learnable parameters, and we want to use a dataset $\mathcal{X} = \left\{x^{(1)}, \ldots, x^{(m)}\right\}$ to learn them. The posterior works as an encoder, since it is used to map inputs x to latent vectors z, and the likelihood as a decoder since it maps latent vectors z back to an input x. We can model this problem as Maximum Likelihood Estimation (MLE). Therefore, we have to compute the posterior:

$$p_\theta(z|x) = \frac{p_\theta(x,z)}{p_\theta(x)} = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}, \tag{3.23}$$

where:

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz. \tag{3.24}$$

The problem with this integral is that it would require to check all possible values that z can assume in order calculate it, what makes the calculation of $p_\theta(z|x)$ intractable, since it is dependent on this integral. To solve this problem, an approximation of the posterior $q_\phi(z|x)$ is introduced, where ϕ is a set of learnable parameters. Now it is also necessary to guarantee that the approximation $q_\phi(z|x)$ is close to the real posterior $p_\theta(z|x)$. In order to do this, we can use the KL divergence to measure how far apart the two distributions are:

$$
\begin{aligned}
D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z|x)) &= \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)} dz \\
&= \int q_\phi(z|x) \log \frac{q_\phi(z|x) p_\theta(x)}{p_\theta(z,x)} dz \\
&= \int q_\phi(z|x) \left(\log p_\theta(x) + \log \frac{q_\phi(z|x)}{p_\theta(z,x)} \right) dz \\
&= \int q_\phi(z|x) \log p_\theta(x) dz + \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z,x)} dz \\
&= \log p_\theta(x) + \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(x|z) p_\theta(z)} dz \\
&= \log p_\theta(x) + \int q_\phi(z|x) \left(\log \frac{q_\phi(z|x)}{p_\theta(z)} - \log p_\theta(x|z) \right) dz \\
&= \log p_\theta(x) + \mathbb{E}_{z \sim q_\phi(z|x)} \left[\log \frac{q_\phi(z|x)}{p_\theta(z)} - \log p_\theta(x|z) \right] \\
&= \log p_\theta(x) + D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z)) - \mathbb{E}_{z \sim q_\phi(z|x)} \log p_\theta(x|z).
\end{aligned}
\tag{3.25}
$$

Therefore, we have:

$$
D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z|x)) = \log p_\theta(x) + D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z)) - \mathbb{E}_{z \sim q_\phi(z|x)} \log p_\theta(x|z), \tag{3.26}
$$

we can now arrange the equation:

$$
\log p_\theta(x) - D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z|x)) = \mathbb{E}_{z \sim q_\phi(z|x)} \log p_\theta(x|z) - D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z)). \tag{3.27}
$$

In the left hand side of the equation we have the log-likelihood of the real data x, $\log p_\theta(x)$ that we want to maximize, and the negative of the KL divergence of estimated posterior $q_\phi(z|x)$ with respect to the real posterior $p_\theta(z|x)$, that we also want to maximize (we want to minimize the divergence, what corresponds to maximize the negative of the divergence). The expression $\log p_\theta(x) - D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z|x))$ is known as Evidence Lower Bound (ELBO) in Variational Bayesian methods. The negative of the ELBO define the loss function for the VAE:

$$\begin{aligned}\mathcal{L}_{VAE}(x;\theta,\phi) &= -\text{ELBO} \\ &= -\log p_\theta(x) + D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z|x)) \\ &= -\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z) + D_{KL}(q_\phi(z|x) \,\|\, p_\theta(z)).\end{aligned} \tag{3.28}$$

Our goal is to find parameters θ and ϕ such that:

$$\theta^*, \phi^* = \underset{\theta,\phi}{\text{argmin}}\ \mathcal{L}_{VAE}(x;\theta,\phi). \tag{3.29}$$

A common way to map an input x to a distribution is making the encoder predict parameters of a distribution, which can be used to sample a latent vector (Figure 3.10).

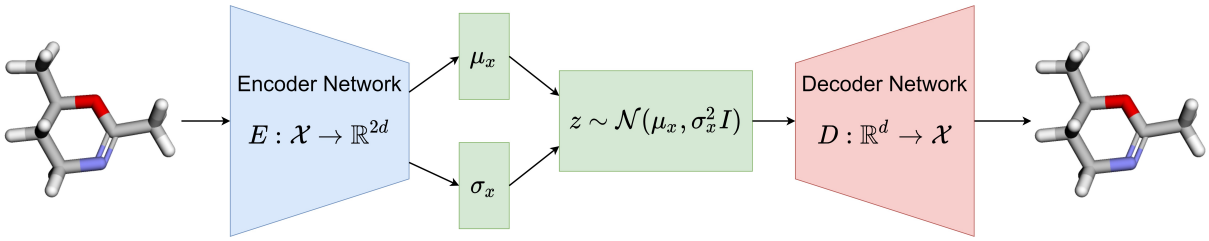


Figure 3.10: VAE framework.

We could think about using an algorithm such as gradient descent to find the parameters θ and ϕ, but there is still a problem with the current framework. We have a sampling operation in the loss function. In order to calculate $-\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z)$ we have to sample $z \sim q_\phi(z|x)$, which is not a problem during the forward pass, but during the backward pass, propagating gradients through a stochastic node is difficult. To solve this issue, the authors of VAE proposed the reparameterization trick (Figure 3.11).

Instead of directly sampling z, they use a deterministic function, which one of the parameters is a random variable ϵ, in other words:

$$z = g\left(\phi, x, \epsilon\right). \tag{3.30}$$

A common choice of distribution for the posterior $q_\phi(z|x)$ is $\mathcal{N}(\mu, \sigma^2 I)$. Now it is possible to train this network without having problems with a stochastic node stopping the gradients from flowing.

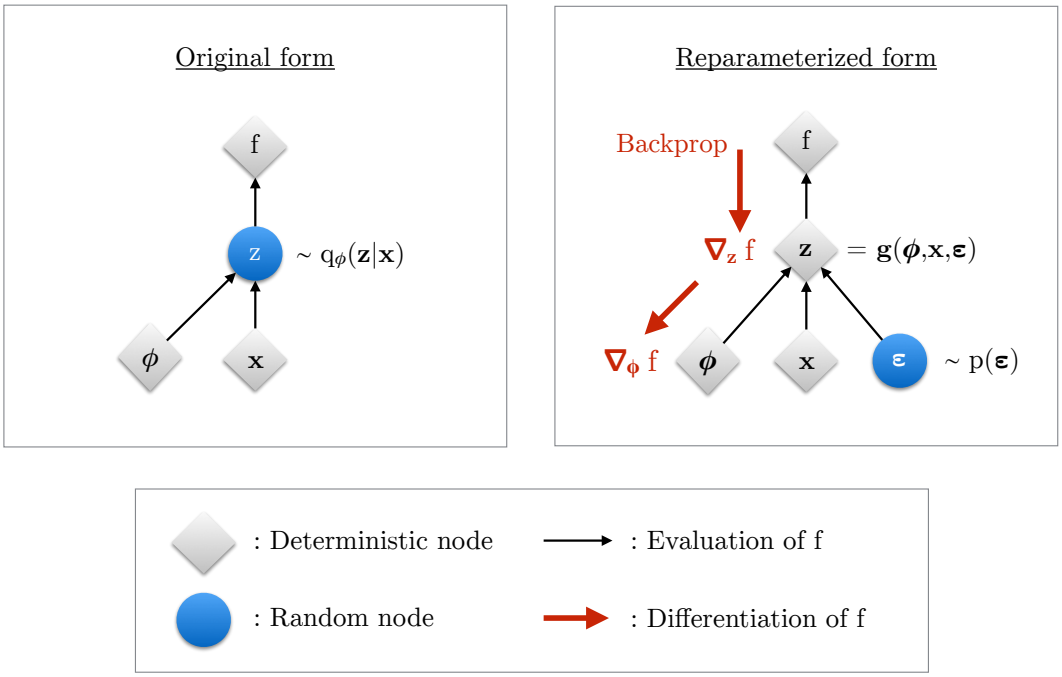


Figure 3.11: Reparameterization trick. Source: Kingma and Welling (2019).

3.3.2 Normalizing Flows

In general, learning the true data generating distribution p_{data}, is a very difficult problem. As we have seen previously, Variational Autoencoders, for example, do not explicitly model the probability distribution of the data p_{data}, since it would require the calculation of the integral $p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz$, and checking all the values that z can assume is infeasible. Instead, we define a distribution which is easier to sample from (usually a normal distribution), and apply a series of nonlinear transformations in order to obtain a new data point.

With Normalizing Flows we can directly learn the probability distribution of the data. But in order to do that, we impose a limitation in the dimension of the latent space. Even with the limitations, using normalizing flows provides a very interesting way to construct generative models with nice properties. Normalizing flows also provides a useful latent representation which we can invert.

It is possible to construct a generative model with these properties because of a famous result in probability theory called the change of variables theorem. This theorem states that if $x \sim p_\theta(x)$ and $z \sim p_z(z)$, where $z = f(x)$ and $f : \mathbb{R}^n \to \mathbb{R}^n$ is a differentiable

and invertible function, we have:

$$p_\theta(x) = p_z(z) \left| \det Jf(x) \right| \tag{3.31}$$

where J is the Jacobian of f. The change of variables theorem provides a connection between the density function of a random variable x with a random variable z, under some transformation f, where $z = f(x)$. The determinant of the Jacobian of f is a volume correction which guarantees that the new probability distribution integrates to one. The function $f(x)$ is called a flow, which is simply a parametric function $f : \mathbb{R}^n \to \mathbb{R}^n$ which is both differentiable and invertible. Instead of thinking about f as a single object, we can design it as a composition of flows, in other words:

$$f(x) = f_K \circ f_{K-1} \circ \ldots \circ f_2 \circ f_1(x). \tag{3.32}$$

The key idea of normalizing flows is to learn to map x, which has a complex distribution, to z which has a simple distribution p_z, where is simple to sample from, using a flow f (Figure 3.12).

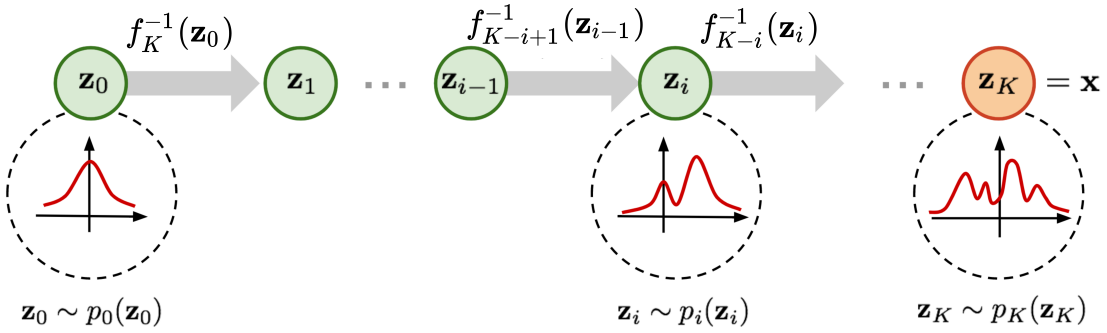


Figure 3.12: Basic framework of normalizing flows. Adapted from: Weng (2018).

We can also take advantage of f invertibility, and obtain x by simply calculating:

$$x = f^{-1}(z) = f_1^{-1} \circ f_2^{-1} \circ \ldots \circ f_{K-1}^{-1} \circ f_K^{-1}(z) \tag{3.33}$$

Sampling is straightforward now, because first we have to sample z from $p_z(z)$, which is easy by design, since we chose $p_z(z)$, and then we only have to compute $x = f^{-1}(z)$, obtaining a sample from $p_\theta(x)$. In other words, to generate a new data point x,

we sample z from a simple distribution p_z, and through the composition of invertible and differentiable functions, we can go from the simple distribution to a complex one, such as the ones of natural images or molecules. Recall that with VAEs we can only approximate the lower bound of the log-likelihood (the ELBO), with normalizing flows, we have an exact formula for the log-likelihood:

$$\begin{aligned} \log p_\theta(x) &= \log p_z(z) + \log |\det Jf(x)| \\ &= \log p_z(z) + \sum_{k=1}^{K} \log |\det Jf_k| \,. \end{aligned} \tag{3.34}$$

Therefore, we can compute the parameters θ of the function f, using Maximum Likelihood Estimation over the dataset $\mathcal{X} = \left\{x^{(1)}, \ldots, x^{(m)}\right\}$:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log p_z \left(f\left(x^{(i)} \,|\, \theta\right)\right) + \sum_{k=1}^{K} \log \left|\det Jf_k \left(x^{(i)} \,|\, \theta\right)\right| . \tag{3.35}$$

Computing the determinant of a sequence of high-dimensional functions can be quite expensive, therefore, the sequence of functions has to be carefully chosen. We could, for example, choose transformations such that the Jacobian matrix is triangular, and hence, the determinant can be calculated by simply multiplying the elements of the diagonal of the Jacobian matrix. Another possible bottleneck in this process is the calculation of the inverse of f. If $f = Ax+b$, for example, where A is an invertible matrix, we still would have problems finding A^{-1}, since it has complexity $O(n^3)$. Therefore, restricting the matrix representing the flows f_i is necessary in order to build models that are feasible.

Coupling flow (Figure 3.13) is a very common and general approach to make the computation of the Jacobian and the inverse efficiently by dividing the input x, in two disjoint subsets, where $x = (x_{1:p}, x_{p+1:n})$. First we apply a permutation $\mathcal{P}$ in the input x in order to make the model learn from all parts of the input, and taking advantage that the inverse of a permutation operation is simply the transposed of the permutation matrix $\mathcal{P}$. We then compute a transformation of x which keeps the first part of the vector, $x_{1:p}$, fixed and apply an invertible transformation to the second part of the vector, $x_{p+1:n}$, in

which the parameters are dependent on $x_{1:p}$. In other words, we have:

$$f(x) = \left(x_{1:p}, \hat{f}\left(x_{p+1:d} \,|\, \theta\left(x_{1:p}\right)\right)\right), \tag{3.36}$$

where $\hat{f}$ is also a flow, but its parameters depend only on $x_{1:p}$. We call θ a coupling network, which receives as input $x_{1:p}$, and outputs the parameters that will be fed to the function $\hat{f}$, which is called coupling transform, where $\hat{f}$ only modifies $x_{p+1:n}$, getting:

$$z = (z_{1:p}, z_{p+1:n}) = \left(x_{1:p}, \hat{f}\left(x_{p+1:n} \,|\, \theta\left(x_{1:p}\right)\right)\right). \tag{3.37}$$

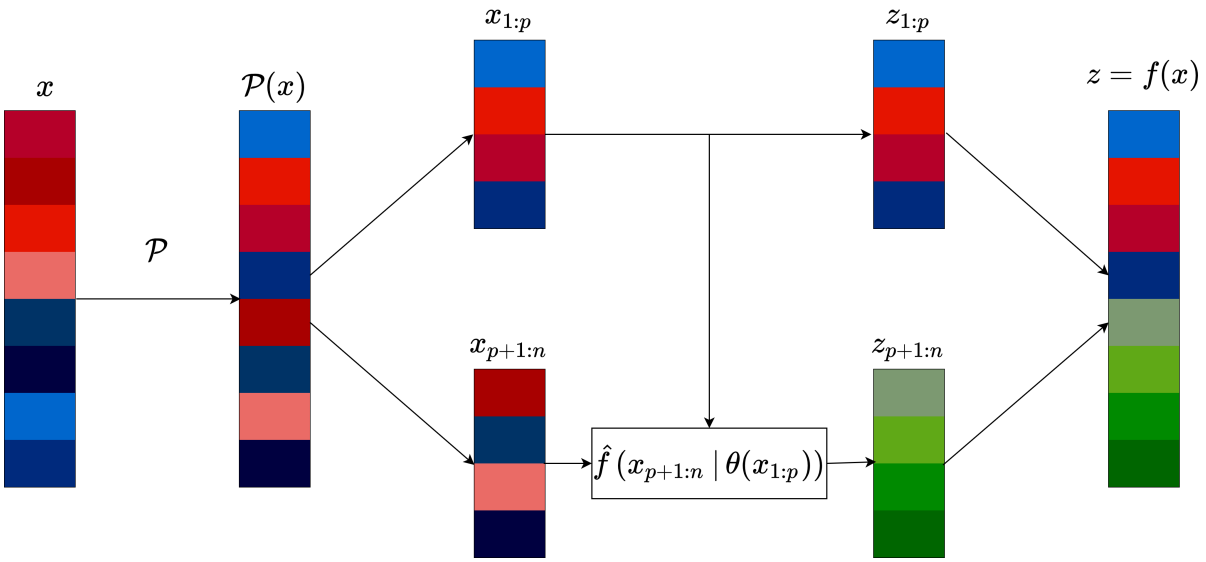


Figure 3.13: Basic interface of a Coupling Flow.

In order to invert z we have just to copy $x_{1:p}$ and apply the inverse transformation on $\hat{f}\left(x_{p+1:n} \,|\, \theta\left(x_{1:p}\right)\right)$, hence:

$$x = f^{-1}(z) = \left(z_{1:p}, \hat{f}^{-1}\left(z_{p+1:n} \,|\, \phi\left(z_{1:p}\right)\right)\right). \tag{3.38}$$

This construction gives a very interesting form to the Jacobian matrix:

$$Jf = \begin{bmatrix} I & 0 \\ \dfrac{\partial \hat{f}\left(x_{p+1:n} \,|\, \phi\left(x_{1:p}\right)\right)}{\partial x_{1:p}} & J\hat{f}\left(x_{p+1:n} \,|\, \phi\left(x_{1:p}\right)\right). \end{bmatrix} \tag{3.39}$$

Since the Jacobian matrix is triangular in this case and the first block is the

identity matrix, in order to compute the determinant, we only have to compute:

$$\det J\hat{f}\left(x_{p+1:n} \mid \phi\left(x_{1:p}\right)\right). \tag{3.40}$$

Many different choices of coupling transformations were explored. In NICE (DINH; KRUEGER; BENGIO, 2014), a simple additive operation is used:

$$\hat{f}\left(x_{p+1:n} \mid \theta\left(x_{1:p}\right)\right) = x_{p+1:n} + \theta\left(x_{1:p}\right). \tag{3.41}$$

Since the calculation of the inverse function $\hat{f}$ does not require inverting $\theta(x_{1:p})$, we can model it as a neural network. In RealNVP (DINH; SOHL-DICKSTEIN; BENGIO, 2016), a more sophisticated (and widely used) affine transformation was proposed:

$$\hat{f}\left(x_{p+1:n} \mid s\left(x_{1:p}\right), t\left(x_{1:p}\right)\right) = x_{p+1:n} \odot \exp\left(s\left(x_{1:p}\right)\right) + t\left(x_{1:p}\right), \tag{3.42}$$

where s and t are the scale and translation functions respectively, and $\odot$ is the Hadamard Product. Again, since there is no need to calculate their inverses, both functions can be modeled as deep neural networks.

3.3.3 Generative Adversarial Networks

One year after the Variational Autoencoder paper was published, another breakthrough in the deep generative modeling field came out, the Generative Adversarial Networks (GANs) (GOODFELLOW et al., 2014). They brought a level of quality never seen before for the image generation task and have been considered one of the most important developments in Deep Learning in the last decade. Instead of trying to model the density function, GANs focus on producing high quality samples that resemble the data generation distribution.

For GANs we assume there are two data distributions, p_{data}, as we have seen before, is the data generating distribution. And now we also have p_g, which is the generative model distribution. We call x real if $x \sim p_{\text{data}}$ and false if $x \sim p_g$. In this setting we want to approximate p_{data} by p_g. In order to approximate this distribution, they propose the use of two players in a adversarial setting:

- The discriminator $D : \mathbb{R}^n \to [0, 1]$, receives as input an observation x that can be real or fake, and gives as output the probability, according to the model, of that observation being real. The goal of D is to tell apart when observations are real or fake, in other words, when $x \sim p_{\text{data}}$, the model wants $D(x) \approxeq 1$ and $D(x) \approxeq 0$ if $x \sim p_g$.

- The generator $G : \mathbb{R}^d \to \mathbb{R}^n$, receives a latent vector $z \sim p_z$ as input, and outputs a valid input for D, $\hat{x} = G(z)$. The goal of the generator is to fool the discriminator, i.e., make $\hat{x} \sim p_g$ so similar to real samples that $D(\hat{x}) \approxeq 1$, in other words D gives high probability of a generated sample from G, being real.

This setting summarizes the framework (Figure 3.14). If the discriminator is good at telling apart real and fake samples, but it still has a hard time to differentiate when $x \sim p_g$ is fake or not, it may imply that p_g recovered, or is a good approximation, of p_{data}. In other words, it would be impossible to recognize when an observation was generated by G or is coming from the dataset $\mathcal{X}$ because they have the same distribution.

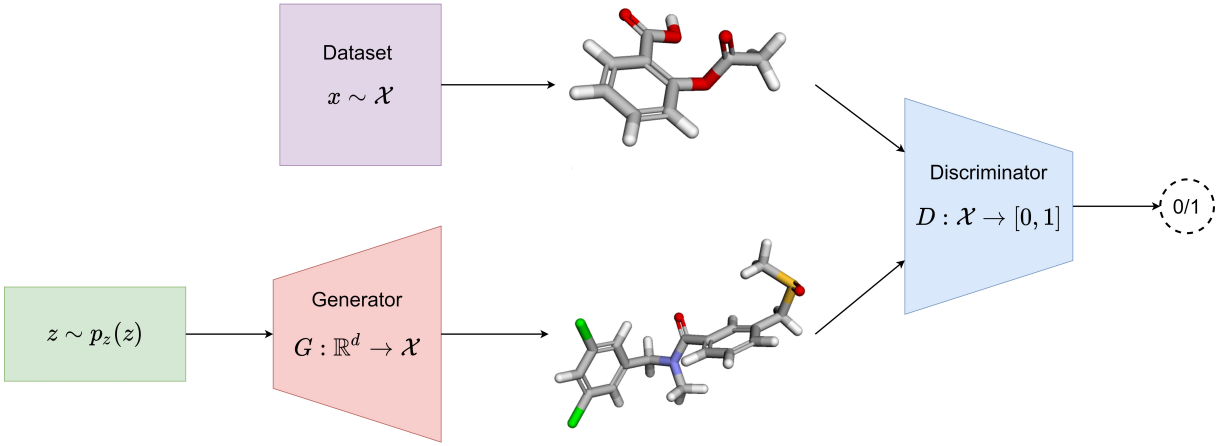


Figure 3.14: GAN framework.

This two-player game is defined by the following function:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}} \log D(x) + \mathbb{E}_z \log (1 - D(G(z))), \tag{3.43}$$

where the discriminator D maximizes the log-probability of correctly classifying

real and fake samples:

$$J^{(D)} = \mathbb{E}_{x \sim p_{\text{data}}} \log D(x) + \mathbb{E}_z \log (1 - D(G(z))), \tag{3.44}$$

and the generator minimizes the log-probability of the discriminator being right when classifying fake samples:

$$J^{(G)} = \mathbb{E}_z \log (1 - D(G(z))), \tag{3.45}$$

if both models have enough capacity, the Nash equilibrium is achieved when $G(z)$ has the same probability distribution as the training set and $D(x) = \frac{1}{2}$ for all x.

GANs have some known bottlenecks (ARJOVSKY; BOTTOU, 2017). In general, GANs are difficult to optimize, in many times, not converging during the training. Another problem occurs when the discriminator is too good at identifying samples, in this scenario, it does not provide meaningful information to the generator, which is then unable to learn, since the gradients are vanishingly small. Another common issue with GANs is called mode collapse, where the generator focuses on a specific part of the latent space, or specific samples, generating realistic, but very similar samples every time, instead of trying to diversify. The generator basically exploits the discriminator, since it knows that the current generated samples are realistic, it does not need to try to generate different samples, at the risk of them being recognized as fake by the discriminator. To remedy these and other problems, many variations in the training procedure and/or the loss functions were proposed (LUCIC et al., 2017). Some of these variations are shown in Table 3.1.

GAN	Discriminator Loss	Generator Loss
mm gan	$-\mathbb{E}_{x \sim p_{\text{data}}}[\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$	$\mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$
ns gan	$-\mathbb{E}_{x \sim p_{\text{data}}}[\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$	$-\mathbb{E}_{\hat{x} \sim p_g}[\log(D(\hat{x}))]$
wgan	$-\mathbb{E}_{x \sim p_{\text{data}}}[D(x)] + \mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$	$-\mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$
wgan gp	$\mathcal{L}_D^{\text{wgan}} + \lambda \mathbb{E}_{\hat{x} \sim p_g}[(\|\|\nabla D(\alpha x + (1 - \alpha \hat{x})\|\|_2 - 1)^2]$	$-\mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$
ls gan	$-\mathbb{E}_{x \sim p_{\text{data}}}[(D(x) - 1)^2] + \mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})^2]$	$-\mathbb{E}_{\hat{x} \sim p_g}[(D(\hat{x} - 1))^2]$
dragan	$\mathcal{L}_D^{\text{gan}} + \lambda \mathbb{E}_{\hat{x} \sim p\text{data} + \mathcal{N}(0,c)}[(\|\|\nabla D(\hat{x})\|\|_2 - 1)^2]$	$\mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$
began	$\mathbb{E}_{x \sim p_{\text{data}}}[\|\|x - \text{AE}(x)\|\|_1] - k_t \mathbb{E}_{\hat{x} \sim p_g}[\|\|\hat{x} - \text{AE}(\hat{x})\|\|_1]$	$\mathbb{E}_{\hat{x} \sim p_g}[\|\|\hat{x} - \text{AE}(\hat{x})\|\|_1]$

Table 3.1: Different GANs architectures. Adapted from: Lucic et al. (2017).

4 Molecular Generation

Many different models were proposed to generate molecules, using a combination of architectures. In this chapter, we first understand how we can evaluate a generative model for molecules and then we will go through some of the state-of-the-art deep generative models for the task of molecular generation. The landscape of models is enormous, so we divided it into a taxonomy that helps to categorize each of them. We try to diversify, regardless of novelty in the ideas, and at the same time, we bring the most successful models so far. For graph generation, which is a nascent field, we also present some works that were not directly used in molecular generation, but are very relevant, and influenced the field in general.

4.1 Evaluating Generative Models for Molecules

There are several ways to determine how good a generative model is for generating molecules. Usually, a set of metrics is used at the same time to evaluate a model. And some metrics are specific to some types of models. For VAE-based methods, we can verify the reconstruction accuracy in a test set. In other words, we measure if after obtaining the latent vector of a given molecule the model is capable of correctly reconstructing the original molecule without errors. We also have more general metrics for evaluating the generation of molecules. For example, it is generally of interest of chemists to know if a molecule is drug-like or not, or to measure predictors (specific chemical properties of a molecule). Some particularly important predictors are the Quantitative Estimate of Drug-likeness (QED) (BICKERTON et al., 2012) and the Octanol-water partition coefficient, $\log P$.

In the most general case, we can measure properties that are not specific to the molecular generation task. For example, the validity metric measures the percentage of generated molecules which are valid. When generating SMILES strings it is usually difficult to guarantee that the generated strings correspond to a molecule. Similarly, for

graphs, the generated graph may not correspond to the molecular graph of any molecule. We could have for instance valence constraints being violated. The uniqueness measures if the model is generating the same molecules repeatedly by simply counting the percentage of the generated molecules that are unique. Diversity captures if the model can explore the chemical space, verifying if the model is not simply generating the same molecules present in the training set. This could mean that the model is suffering from problems like overfitting, where the model is too sensitive to the training data, or mode collapse, common in GANs as we have seen before.

4.2 String based-methods

The first generative models for molecules worked directly with SMILES strings. Taking advantage of advances in DL for Natural Language Processing. The models that generate SMILES strings are usually simpler, which leads to faster training and less complexity for the implementation part. In this section, we present some models that were created to generate SMILES.

4.2.1 MolVAE

The first deep generative model for molecules was the MolVAE (GÓMEZ-BOMBARELLI et al., 2018), which was posted on Arxiv in 2016. MolVAE generates SMILES strings, character by character, using a VAE. The model maps a SMILES string in a continuous representation using an encoder, which is mapped back to a (possibly valid) SMILES string by the decoder. Both the encoder and the decoder are modeled as RNNs. Generating SMILES character by character can be problematic regarding the validity of the strings generated, since the model needs to keep track of the SMILES syntax. In some sense, the model does not only have to learn to generate molecules, but also the SMILES grammar as well. If a parenthesis was opened, for example, it must be closed, otherwise, the generated string will not represent a valid SMILES string. And since the generation of their model is unrestricted, it is prone to generate invalid SMILES.

Additionally, MolVAE uses a predictor on top of the latent space, in order to

optimize molecular properties by guiding the generation (Figure 4.1). As mentioned before, their model maps a molecule to a latent representation z, and they jointly train a MLP $f(z)$ to predict chemical properties of molecules from the latent representation of the molecule. Therefore, starting from a vector z, it is possible to optimize z in the latent space, in order to find a vector with optimized chemical properties. The decoder can then be used to decode the new optimized latent representation to a (possibly valid) SMILES string.

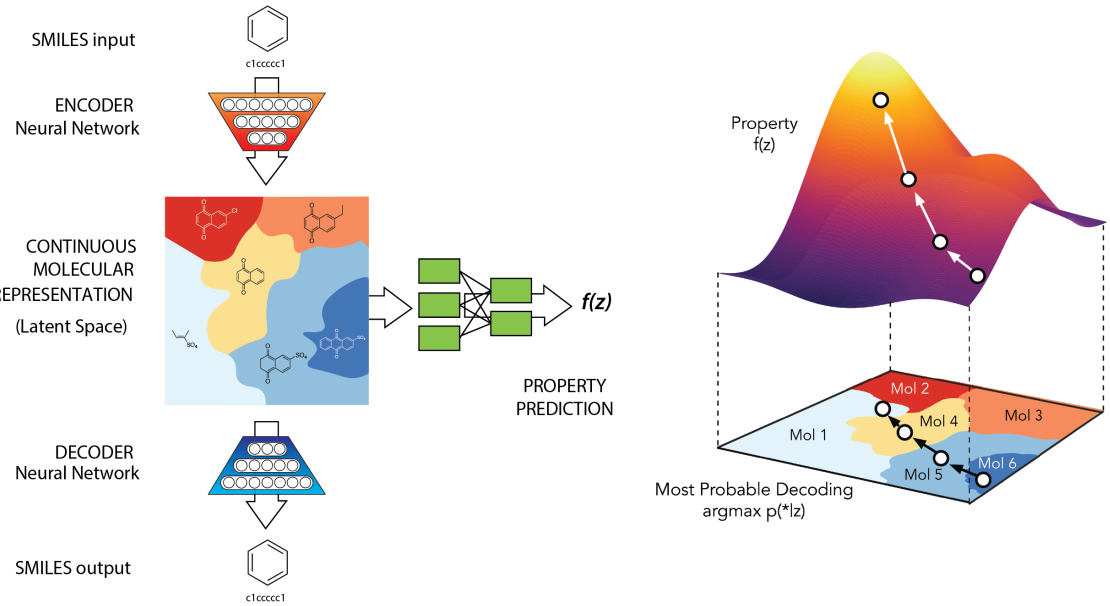


Figure 4.1: MolVAE framework. Source: Gómez-Bombarelli et al. (2018).

4.2.2 CharRNN

A simpler idea is to directly use RNNs to encode SMILES strings into continuous representations using a CharRNN (SEGLER et al., 2018). CharRNN uses LSTMs cells to generate a sequence of characters. At each time step a character is sampled from the learned distribution of the model, and fed to the model again so it can sample the next character as shown in Figure 4.2. They use one-hot encoding for representing each character of the SMILES strings. First they trained their model on a large set of molecules with the goal of learning how to generate valid SMILES strings. After this training they chose datasets with active molecules and re-trained the model in order to produce more interesting molecules. With this strategy they were able to generate molecules with a

good rate of validity, and at the same time, generate molecules that were interesting for a specific downstream task.

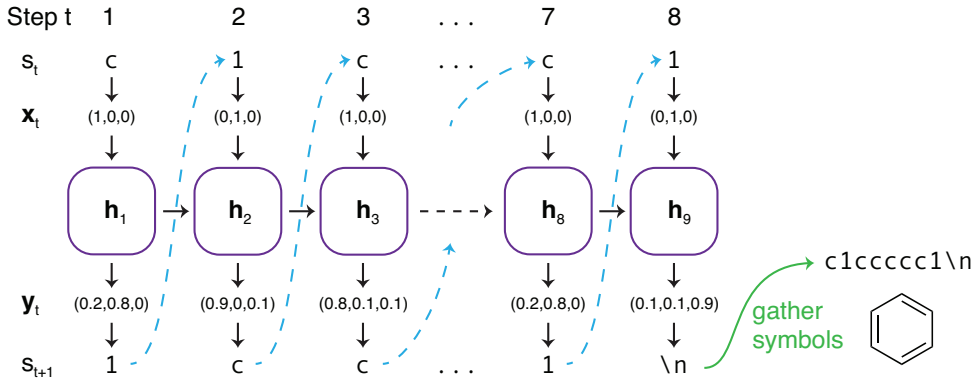
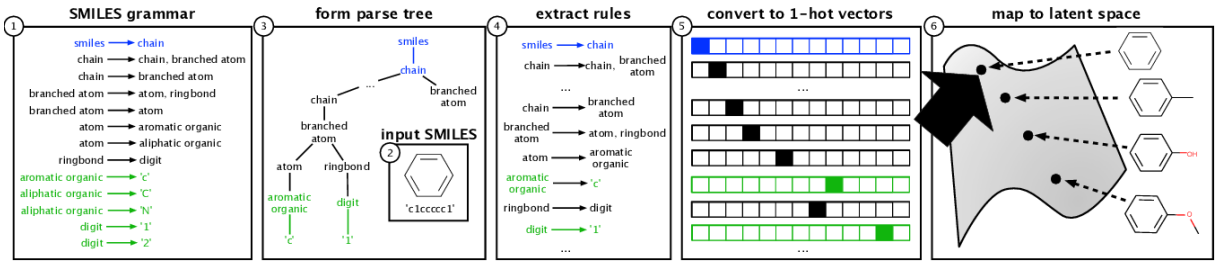


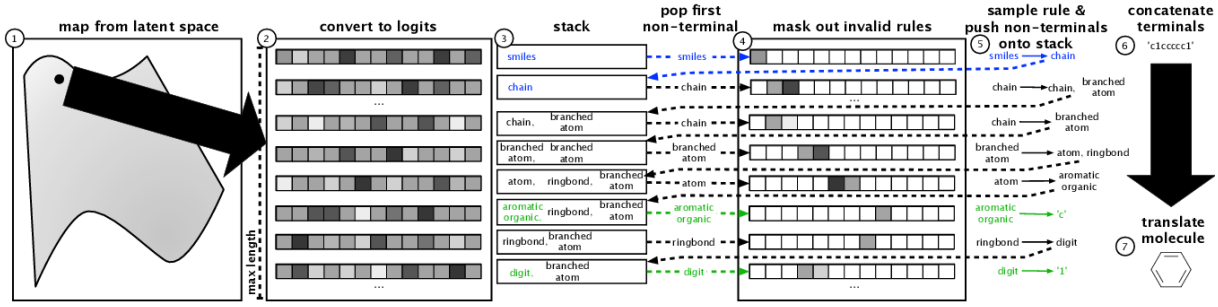
Figure 4.2: CharRNN architecture. Source: Segler et al. (2018).

4.2.3 Grammar Variational Autoencoder

In Chapter 2, we presented a context-free grammar from the SMILES specification and showed how SMILES strings can be represented by a parse tree using the OpenSMILES grammar. The methods that we have seen so far encode each character of the SMILES string as a one-hot vector, which is fed to the neural network for generating new molecules. This idea can be directly applied to a context-free grammar as well. In GrammarVAE (KUSNER; PAIGE; HERNÁNDEZ-LOBATO, 2017), instead of encoding single characters, the authors encode the production rules. Therefore, during the generation, production rules are sampled in a sequential way. The encoder maps a sequence of production rules into the latent space, and the decoder tries to recover the sequence from the latent vector, as shown in Figure 4.3. An important observation is the fact that not every production rule is valid at a given time step, for this reason, a mask is used in the probability vector used to sample the rules, guaranteeing that only valid rules (according to the grammar) can be sampled. But this restriction does not guarantee the generation of only valid SMILES strings, since there are syntactic constraints that are not captured by the OpenSMILES grammar. Similarly to MolVAE, they use a VAE, where after obtaining a latent representation z, they propose the use of Bayesian optimization in the latent space to obtain optimized molecules.



(a) Encoder



(b) Decoder

Figure 4.3: GrammarVAE architecture. Source: Kusner, Paige and Hernández-Lobato (2017).

4.2.4 ORGAN

ORGAN (GUIMARAES et al., 2017) is a generative model for sequences, in general, which combines adversarial training with RL to generate sequences, with optimized properties through guided generation. The proposed model was tested in two tasks, molecule generation as SMILES strings and musical notes. The adversarial training was done using SeqGAN (YU et al., 2017), an adaptation of the original GAN that works with sequences. In the original formulation, it is not possible to train a GAN, when p_{data} is a discrete distribution, since we would have a discrete sampling operation, which is non-differentiable.

With SeqGAN, the generator G is modeled as a stochastic policy in a RL setting where the discriminator's output, which measures the quality of the generated samples, works as a reward for the generator. G is trained using the REINFORCE algorithm (WILLIAMS, 1992). Additionally, they also incorporate domain-specific goals, in other words, it is possible to guide the generation with other reward functions, besides the discriminator signal, and enforce diversity in the generated samples, by penalizing sequences that are non-unique and less diverse (Figure 4.4). A tunable parameter is introduced to weigh the importance of each type of reward.

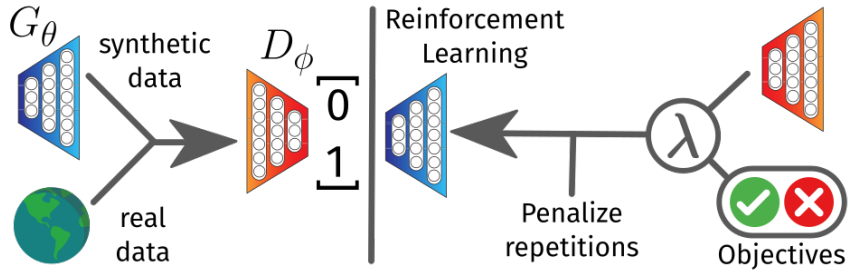


Figure 4.4: ORGAN framework. Source: Guimaraes et al. (2017).

The reward function is defined by:

$$R(y_{1:T}) = \lambda D_\phi(y_{1:T}) + (1 - \lambda)O_i(y_{1:T}), \tag{4.1}$$

where O_i, represents a specific property that we want to optimize. And the parameter λ controls the importance of each type of reward. If $\lambda = 1$, we have the original SeqGAN model, which is based only on the discriminator signal. If $\lambda = 0$, we have a RL algorithm, which is not enforced to generate realistic samples, only to optimize a property. Additionally, in order to prevent mode collapse, the model is penalized if it generates repeated sequences. Where they simply divide the reward of a repeated generated sequence, by the number of times the sequence was generated.

4.3 Graph-based methods

The generation of graphs is much more complex in general, when compared to the generation of SMILES strings. In order to not limit a generative model for molecular graphs, regardless of the chemical information that is present in molecules, it is not sufficient to represent it with the adjacency matrix and node and bond types. Information, such as chirality, may be incorporated as well to create more general models.

4.3.1 Variational Graph Auto-Encoders

One of the first works using Variational Autoencoders for graphs was proposed by Kipf and Welling (2016b). In this work they use a GCN as encoder, mapping an undirected graph to a latent space, and using a simple process of taking the dot product of the latent

vector as decoder. For the encoder, both the Adjacency matrix and the feature vector are taken as input to generate the latent vector. For the decoder, it is only necessary to take the dot product between different latent vectors (representing the nodes in the graph) to uncover the entries of the adjacency matrix. In other words, $z_i \cdot z_j$ gives the score between nodes i and j. After that the logistic function is applied, normalizing the score to a number between 0 and 1. This number can be interpreted as an edge probability between the two nodes. Even though a generative model is used, they evaluated the model only in the task of link prediction.

They were able to observe that the latent vectors learned were richer and improved the accuracy on a link prediction task (Figure 4.5), when compared to a vanilla Autoencoder, for example. But their method was not evaluated or constructed to be applied in generative tasks. Therefore, it has several limitations for generating graphs. For instance, it is not possible to generate graphs of variable sizes. Even with this kind of limitation, this work showed the possibility of applying VAEs to graph-structured data.

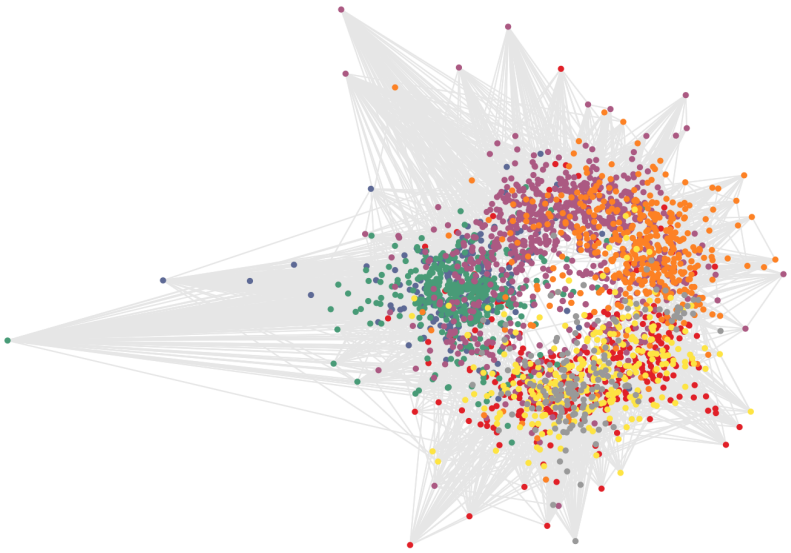


Figure 4.5: Learned latent space of a VGAE trained on a citations dataset. Source: Kipf and Welling (2016b).

4.3.2 GraphVAE

While Kipf and Welling (2016b) proposed a VAE for Graphs with the goal of learning node embeddings, Simonovsky and Komodakis (2018) proposed a framework (Figure 4.6)

which was used to generate graphs, and was evaluated in the molecular generation task. Most of the generative models for graphs are based on Encoder-Decoder approaches. That means we first map a graph, a discrete object to a continuous vector z. From this vector we try to reconstruct the original discrete object. The first part (encoding) is generally easy, the challenging part is to decode a continuous representation to a discrete object. To solve this problem, their model's output is a probabilistic fully-connected graph at once, where they use a parameter k to define the maximum number of nodes. The decoder of the model is deterministic, they use a MLP with three outputs in the output layer. The presence of nodes and edges in the generated graph are modeled as Bernoulli variables. After decoding, they use a graph matching algorithm to align the generated graph to the ground truth. The graph matching algorithms were used to find a correspondence between the nodes of the original molecular graph $\mathcal{G}$ and the generated graph $\hat{\mathcal{G}}$, then measuring the quality of the reconstruction.

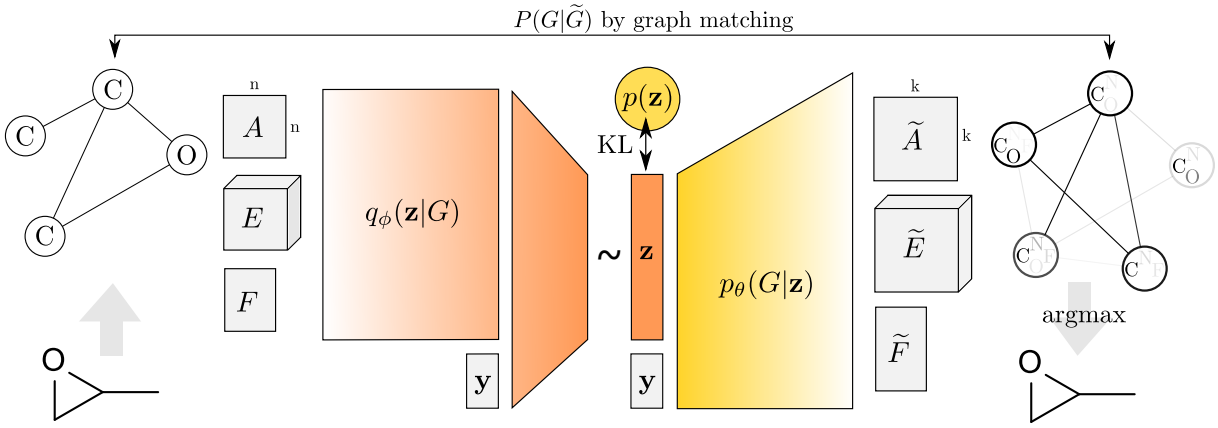


Figure 4.6: GraphVAE architecture. Source: Simonovsky and Komodakis (2018).

4.3.3 MolGAN

As seen before, the VAE loss, in the original formulation, is formed by a regularization term, and a reconstruction term. Although it is easy to compare some types of data, such as images, which have fixed size, for graphs, more sophisticated approaches are necessary. In the last section, we talked about GraphVAE, which uses a graph matching algorithm in order to calculate the similarity between the original and reconstructed graph.

Now we talk about MolGAN (CAO; KIPF, 2018), a generative one-shot model for small molecular graphs (molecules with up to 9 heavy atoms) which uses GANs. Since

GANs map directly a latent vector z to a graph $\mathcal{G}$, without comparing it to another graph in the loss function, MolGAN does not need to perform an expensive graph matching procedure, as GraphVAE does, which makes the training faster.

However, as we also have seen, one interesting approach for generative models based on VAEs is to perform optimization directly on the latent code z, which is continuous. What is not possible using GANs. One alternative to generate optimized molecules is the use of RL, where we can compute properties of the generated molecular graph and use it as rewards to a RL agent. In MolGAN, RL is used to guide the generation of molecules with desired properties during training.

The generator G is modeled as a policy. G takes as input a vector z sampled from a standard normal distribution and outputs a graph (which may or may not be a valid molecular graph). A reward network, which is used as an approximation of the reward function, takes as input a graph generated by G and computes the immediate reward. The general framework of MolGAN is shown in Figure 4.7.

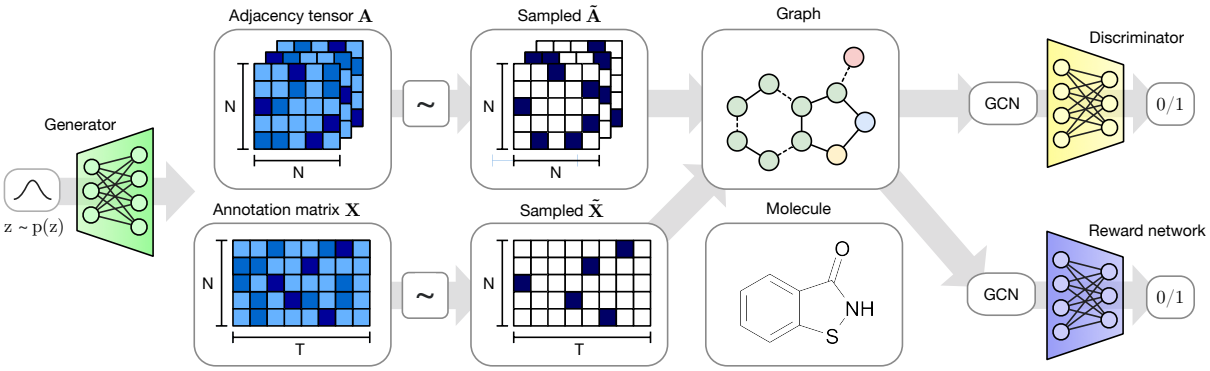


Figure 4.7: MolGAN architecture. Source: Cao and Kipf (2018).

4.3.4 Graph Convolutional Decoder

The two one-shot methods presented before, namely, GraphVAE and MolGAN have to generate in one step the number of atoms, the atoms and the bonds between atoms, which can be difficult to do simultaneously. In order to tackle this problem, Bresson and Laurent (2019) proposed a framework where it is possible to generate molecular graphs in two steps. Using a latent code, first it generates the atoms and then the bonds. In order to generate the atoms, they train a MLP to predict the molecular formula of a

given latent vector. With this information, they are able to understand the distribution of atoms in the molecule as a bag of atoms, and then they use this bag of atoms to build a histogram. At this point all the atoms that are part of the molecule are known, but not the way they are bond together to form the molecule. The second part is responsible for assembling the bag of atoms. To generate the bond structure, both the latent code and the molecular formula generated before are used to predict how the atoms are bonded. With this approach they avoid working with a variable size structure, since both the latent code, which is a d-dimensional vector, and the histogram of each molecule, will have the same size. This one-shot approach can easily generate invalid molecules (that for example, violates the valency of atoms). To avoid generating invalid molecules the authors perform beam search. The general framework of the model is shown in Figure 4.8.

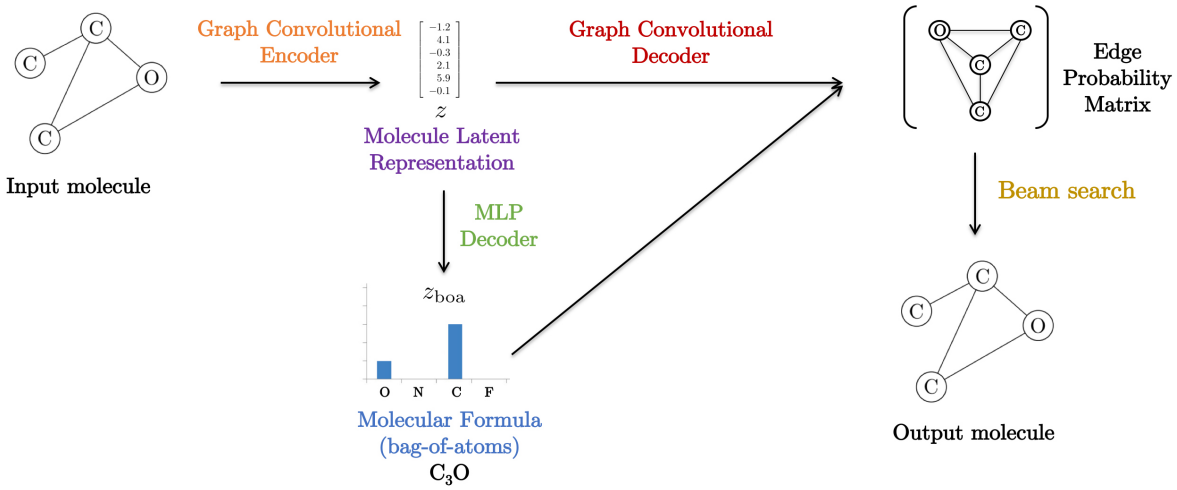


Figure 4.8: Graph Convolutional Decoder framework. Source: Bresson and Laurent (2019).

4.3.5 MolDQN

Zhou et al. (2019) presented a framework called Molecule Deep Q-Networks (MolDQN), for both molecular generation and optimization. Their model uses RL and domain chemistry knowledge in the generation process. The RL agent is used to work on the molecular graph and with the domain knowledge of chemistry, the agent only performs valid actions on the molecular graph, i.e., actions that ensure chemical validity (guarantee that

valence constraints are being followed). In order to do so, they model the modification of a molecule as a MDP. A Deep Q-Network (DQN) (MNIH et al., 2013) is used to solve the MDP. In their formulation, at each time step the agent has to take an action of adding/removing atoms and bonds. This process can take place until a maximum number of time steps T and at each time step, invalid actions are never available to the model. In Figure 4.9, the model is generating a molecule and has to pick the next action, we can see the Q values for all the valid actions in this time step.

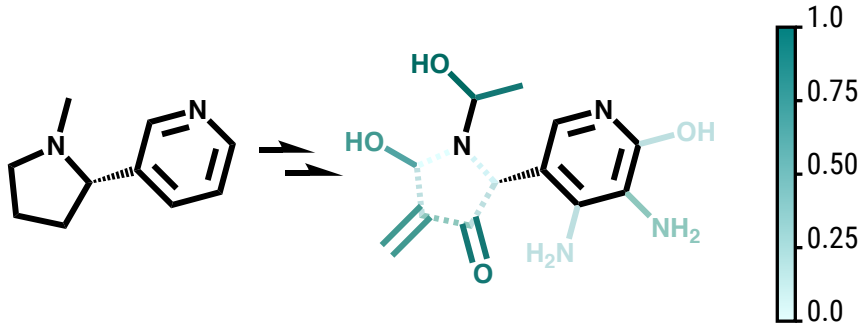


Figure 4.9: Q values of MolDQN for a given molecule. Source: Zhou et al. (2019).

Another important component of their model is the use of heuristics in order to generate more interesting molecules. For example, they allow the formation of rings in a generated molecule up to a maximum number of atoms, which is predefined. This type of decision is based on expert knowledge. Their reward signal is a vector of different properties weighted by a vector w which is defined by the user. In this manner, the user can choose before starting the training which properties the model should give more importance. This vector of properties is then scalarized and weighted by the vector w using a simple dot product. Moreover, a higher reward is given to steps closer to the limit of time steps T, this is due to the fact that the final molecule is more important than intermediate ones, since it will be the output of the model.

4.3.6 GraphRNN

You et al. (2018b) proposed GraphRNN, which generates graphs using a sequential process of adding nodes and edges, modeling transition probabilities. They use a RNN to condition the graph generated at a certain time step, so they can add new nodes and edges. The generation process works as follows, they first add the first node, and then

the second. After that they decide if the two nodes need to be connected by an edge or not. And they keep adding nodes and using a classifier to predict if the new added node will be connected to the nodes already present in the graph or not. One key observation is that a graph $\mathcal{G}$ with a node ordering π can be uniquely mapped into a sequence of nodes edge additions $s^{\pi} = (s_1^{\pi}, s_2^{\pi}, \ldots, s_T^{\pi})$. The sequence s^{π} has two levels, one for adding nodes, and other for adding the edges (Figure 4.10). After a node i is inserted in the graph, it is necessary to add the edges of that node. This is done with edge level sequence $s_i^{\pi} = \left(s_{i,1}^{\pi}, \ldots, s_{i,i-2}^{\pi}, s_{i,i-1}^{\pi}\right)$ where $s_{i,j}^{\pi} = 1$ if the model decides to add an edge between nodes i and j and 0 otherwise. To model this two level-process they use nested RNNs with teacher forcing as strategy. In other words, during training is given to the model the real input, instead of the predicted by the model. And during test time, the output of the model is passed as input to the next time step.

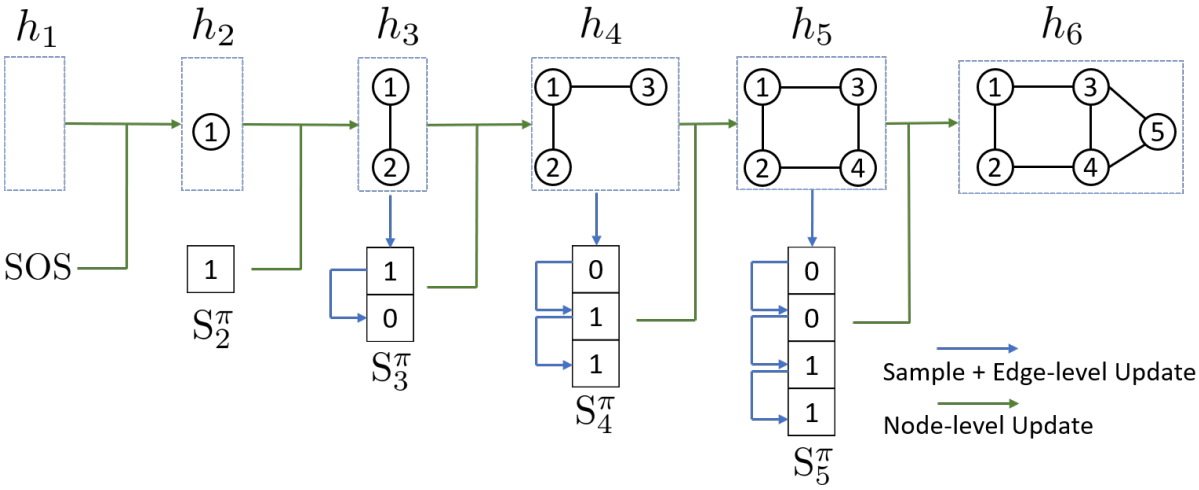


Figure 4.10: GraphRNN architecture. Source: You et al. (2018b).

One problem with learning to generate graphs with an arbitrary node ordering is that when the i^{th} node is added, the model has to decide if the edges will be added with each of the $i-1$ nodes added previously in the graph. To avoid this expensive operation, the authors propose a breadth-first-search (BFS) node ordering.

4.3.7 Graph Convolutional Policy Network

You et al. (2018a) proposed a Graph Convolutional Policy Network (GCPN). They combine Graph Representation Learning, Reinforcement Learning and Adversarial Training in one framework which allows the generation of molecules with specific desired proper-

ties. Their model is able to generate molecular graphs, using a GCN to capture structural graph information and RL for directing the generating process of molecular graphs. The three aspects they want their model to have is the ability to optimize chemical properties, such as drug-likeness, obey chemical rules and mimic the dataset distribution.

They use two types of rewards for the RL agent, the instantaneous reward is a small positive reward given at each time step if the model took a valid action, and the long-term reward is given at the end of the generation, and the reward signal is dependent on the chemical properties of the molecule. The GCN is used to predict the next generation action by obtaining an embedding, which is fed into a predictor to decide the links of the added node. In a high level overview of this framework (Figure 4.11), they first add a node to the graph, and then use a GCN to predict which nodes it will be connected to, after this action is taken, the chemical validity of the graph is checked and then a reward computed, after the graph generation is completed, a final reward is computed based on the chemical properties of it. The training is divided into two parts, first they train the model using a policy to imitate the actions taken by real molecular graphs. In the second step they train a policy to optimize the rewards.

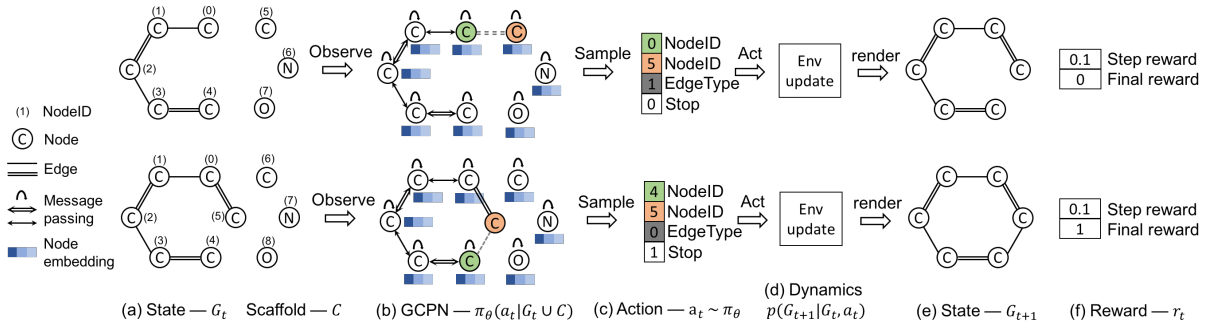


Figure 4.11: GCPN architecture. Source: You et al. (2018a).

4.3.8 Junction Tree Variational Autoencoder

So far we have seen generative models for graphs which either generate the whole graph at once or sequentially add nodes and edges to the graph. But not every graph is a chemically valid molecule. When we are generating atoms and bonds we may achieve an invalid state, but it may be difficult to evaluate if a state is valid or not. Moreover, this sequential generation may suffer from another problem, it could be difficult to train, because the

model has to remember actions taken during the process, this can be problematic for generating big graphs. With this idea in mind, Jin, Barzilay and Jaakkola (2018) proposed the Junction Tree Variational Autoencoder (JT-VAE), where molecules are represented by their function group (Figure 4.12) rather than only their atoms, trying to tackle the aforementioned problems.

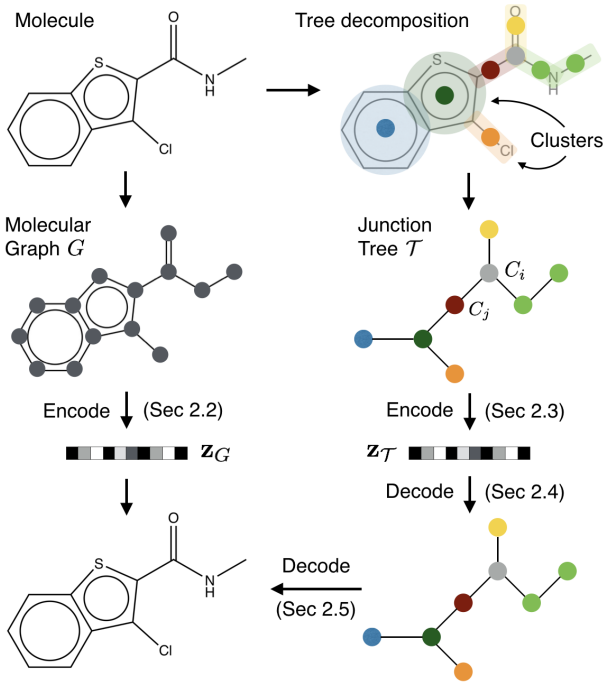


Figure 4.12: JT-VAE architecture. Source: Jin, Barzilay and Jaakkola (2018).

One motivation to propose such a framework was the observation that many invalids states were achieved during the generation of an aromatic ring. Hence, in their work, they generate molecules group by group. This creates a generation sequence which has a shorter number of time steps and is easier to check the chemical validity of intermediate steps. The key idea of the work is to use tree decomposition to decompose the molecules in functional groups. They encode both the molecular graph and the tree structure into latent spaces using Neural Message Passing Networks. To decode from the latent vector z_T they use two types of predictions. The topological prediction is used to decide whether they want to expand a child node or backtrack. And the label prediction is used to determine the label of a node. Additionally, they use Bayesian optimization to generate optimized molecules. Achieving this by predicting molecular properties in the latent space. Starting from the latent vector of a known molecule, it is possible to apply gradient ascent in the latent space to find a latent vector which can be decoded to

a molecule with desired properties, this will be potentially a modification of the original one.

4.3.9 Hierarchical VAE

Previous works presented rely mostly on atoms or small structures to generate molecules. For instance, with auto-regressive models, during generation, at each time step, atoms or small structures will be added and connected to the current generated graph. This can be problematic when generating larger molecular structures, since these models will have to use many generation steps, therefore, they may suffer from gradient vanishing problem and error accumulation. Jin, Barzilay and Jaakkola (2020) tested the reconstruction accuracy of such models, and pointed out their poor performance to reconstruct the molecular graph of large molecules. With this in mind, they proposed Hierarchical VAE, a model that uses motifs, which are larger and more flexible building blocks, during the generation of molecules, as shown in Figure 4.13.

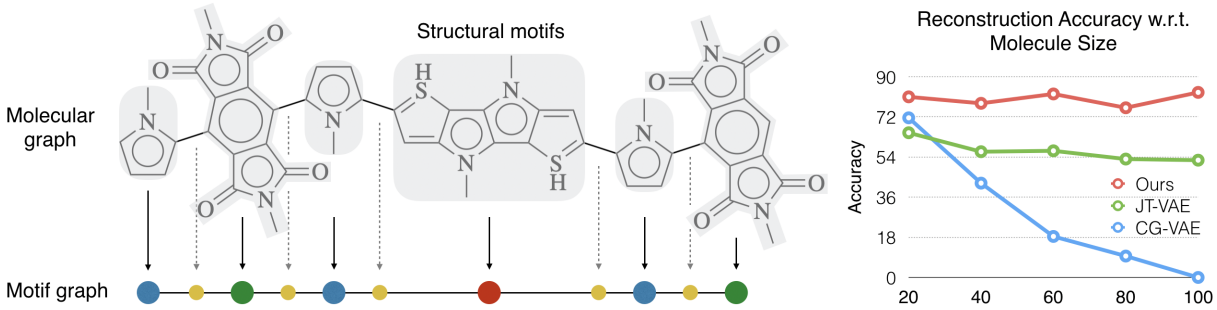
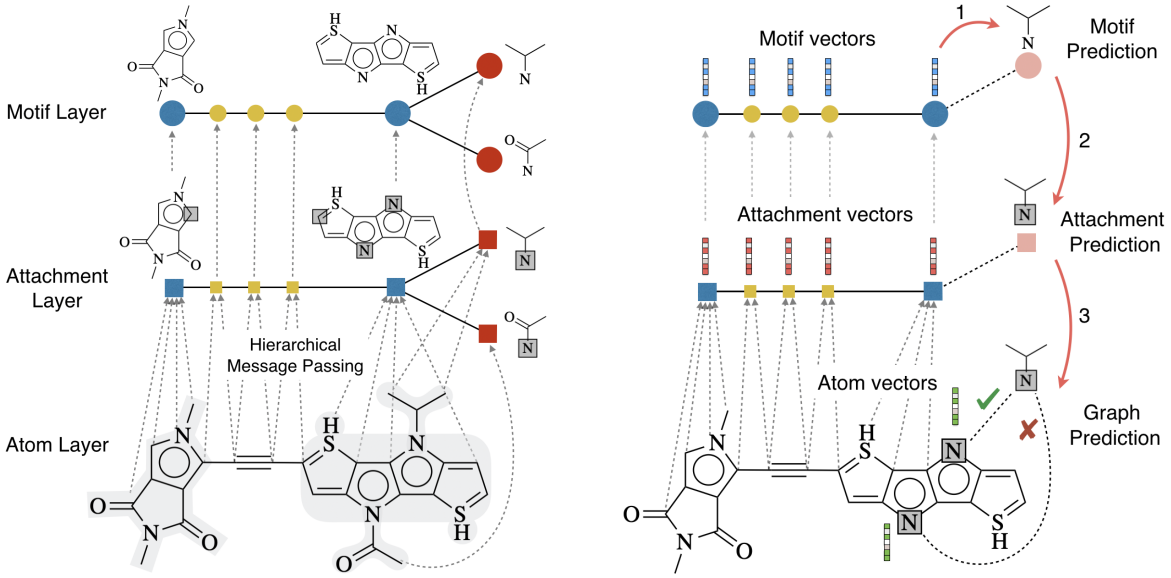


Figure 4.13: Hierarchical VAE framework. Source: Jin, Barzilay and Jaakkola (2020).

JT-VAE for example, is restricted only to generate substructures, which are atoms or rings. Hierarchical VAE is an extension of JT-VAE, which cannot directly adapt the structure from the previous work. One issue is that for a JT-VAE, after predicting the substructures present in the molecule, it has to assemble the substructures in each neighborhood. This task would be computationally infeasible in this new architecture, because now the number of possible combinations could explode, since the combinatorial enumeration depends on the size of the motif (which are much larger now than the substructures). The structure of this new architecture is the following. First, it is necessary to extract the motifs from the molecules. For this, they apply simple heuristics and it can be easily

adapted to other types of extraction strategies. For instance, it is also possible to learn the motifs from the data. The Encoder encodes both the molecular graph at atomic level and motif level (Figure 4.14).



(a) Hierarchical VAE Encoder (b) Hierarchical VAE Decoder

Figure 4.14: Hierarchical VAE Encoding and Decoding Processes. Source: Jin, Barzilay and Jaakkola (2020).

During the generation process there are three steps. Motif prediction: At each time step, a new motif is added in an auto-regressive fashion. Attachment prediction: It predicts how the new added motif will attach to the current generated graph. In other words, it selects which of the possible attachment points will be attached. Graph prediction: Predicts how two different motifs will be connected by their attachment points.

4.3.10 GraphAF

GraphAF (SHI et al., 2020) is an autoregressive model for molecular generation based on normalizing flows. As some of the models that we have seen before, the generation of a molecule is defined as a sequential process of adding atoms and bonds, and checking the validity of the generated structure (Figure 4.15). With normalizing flows they were able to learn an invertible mapping from the complex molecule data distribution and a Gaussian distribution, what allows the computation of the exact likelihood.

They adapted the R-GCN architerure (SCHLICHTKRULL et al., 2018) to learn the node embeddings. And they used two MLPs to predict the type of the nodes and

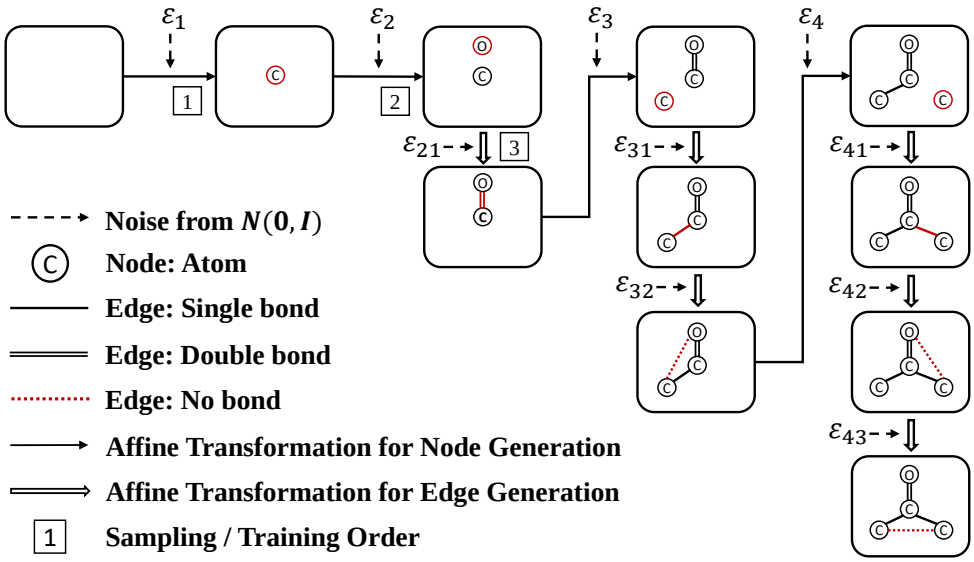


Figure 4.15: GraphAF framework. Source: Schlichtkrull et al. (2018).

edges of the graph being generated at each time step. Since atom and bond types are discrete, they use a dequantization technique, to obtain a continuous representation from this discrete representation by adding a noise to the tensors representing these structures. To be able to guide the generation towards desired properties they use Proximal Policy Optimization (PPO) (SCHULMAN et al., 2017). They use both an intermediate reward, which penalizes structures that violate valency constraints, and a final reward to guide the generation towards desirable properties.

5 Conclusion

In this work we provided an overview and analysis of Deep Generative Models for molecular design and optimization, a nascent field, which deals with a very important and challenging problem, which can bring a lot of benefits for society. We formalize the main concepts involving Deep Generative Models and make a connection about how these methods have been applied to molecular design.

With this review, it was possible to identify a growing interest in methods that work directly with graphs, as opposed to SMILES strings that were a preferred representation for the early generative models for molecules. Another important observation is the combination of Reinforcement Learning with other methods in order to generate optimized molecules. RL provides a powerful manner to not only generate valid molecules, but to guide the generation towards specific goals.

As a future work, with all the knowledge acquired during this work, we intend to propose a deep generative model for molecules combining ideas from the current state-of-the-art methods. Molecular generation is a challenging task, and the use of Deep Generative Models seems to be a promising direction that can lead to many innovations in the future.

Bibliography

ARJOVSKY, M.; BOTTOU, L. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

BENGIO, S. et al. Scheduled sampling for sequence prediction with recurrent neural networks. *arXiv preprint arXiv:1506.03099*, 2015.

BICKERTON, G. R. et al. Quantifying the chemical beauty of drugs. *Nature chemistry*, Nature Publishing Group, v. 4, n. 2, p. 90–98, 2012.

BRESSON, X.; LAURENT, T. A two-step graph convolutional decoder for molecule generation. *arXiv preprint arXiv:1906.03412*, 2019.

CANZIANI, A. *Properties of natural signals*. 2020. ⟨https://atcold.github.io/pytorch-Deep-Learning/en/week03/03-3/⟩. Accessed: 2021-05-05.

CAO, N. D.; KIPF, T. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

CHO, K. et al. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

DINH, L.; KRUEGER, D.; BENGIO, Y. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

DINH, L.; SOHL-DICKSTEIN, J.; BENGIO, S. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

ELTON, D. C. et al. Deep learning for molecular generation and optimization-a review of the state of the art. *arXiv preprint arXiv:1903.04388*, 2019.

FUKUSHIMA, K.; MIYAKE, S. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In: *Competition and cooperation in neural nets*. [S.l.]: Springer, 1982. p. 267–285.

GAYNES, R. The discovery of penicillin—new insights after more than 75 years of clinical use. *Emerging infectious diseases*, Centers for Disease Control and Prevention, v. 23, n. 5, p. 849, 2017.

GILMER, J. et al. Neural message passing for quantum chemistry. In: PMLR. *International conference on machine learning*. [S.l.], 2017. p. 1263–1272.

GÓMEZ-BOMBARELLI, R. et al. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, ACS Publications, v. 4, n. 2, p. 268–276, 2018.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep learning*. [S.l.]: MIT press, 2016.

GOODFELLOW, I. et al. Generative adversarial nets. *Advances in neural information processing systems*, v. 27, 2014.

GROVER, A.; LESKOVEC, J. node2vec: Scalable feature learning for networks. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.: s.n.], 2016. p. 855–864.

GUIMARAES, G. L. et al. Objective-reinforced generative adversarial networks (organ) for sequence generation models. *arXiv preprint arXiv:1705.10843*, 2017.

HAMILTON, W. L. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, Morgan & Claypool Publishers, v. 14, n. 3, p. 1–159, 2020.

HOCHREITER, S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, World Scientific, v. 6, n. 02, p. 107–116, 1998.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

JIN, W.; BARZILAY, R.; JAAKKOLA, T. Junction tree variational autoencoder for molecular graph generation. In: PMLR. *International conference on machine learning*. [S.l.], 2018. p. 2323–2332.

JIN, W.; BARZILAY, R.; JAAKKOLA, T. Hierarchical generation of molecular graphs using structural motifs. In: PMLR. *International Conference on Machine Learning*. [S.l.], 2020. p. 4839–4848.

JUMPER, J. et al. Highly accurate protein structure prediction with alphafold. *Nature*, Nature Publishing Group, p. 1–11, 2021.

KARRAS, T.; LAINE, S.; AILA, T. A style-based generator architecture for generative adversarial networks. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. [S.l.: s.n.], 2019. p. 4401–4410.

KIM, S. et al. Pubchem substance and compound databases. *Nucleic acids research*, Oxford University Press, v. 44, n. D1, p. D1202–D1213, 2016.

KINGMA, D. P.; WELLING, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

KINGMA, D. P.; WELLING, M. An introduction to variational autoencoders. *arXiv preprint arXiv:1906.02691*, 2019.

KIPF, T. N.; WELLING, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

KIPF, T. N.; WELLING, M. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

KUSNER, M. J.; PAIGE, B.; HERNÁNDEZ-LOBATO, J. M. Grammar variational autoencoder. In: PMLR. *International Conference on Machine Learning*. [S.l.], 2017. p. 1945–1954.

LAMB, A. M. et al. Professor forcing: A new algorithm for training recurrent networks. In: *Advances in neural information processing systems*. [S.l.: s.n.], 2016. p. 4601–4609.

LOIKE, J.; MILLER, J. Opinion: Improving fda evaluations without jeopardizing safety and efficacy. v. 31, 03 2017.

LUCIC, M. et al. Are gans created equal? a large-scale study. *arXiv preprint arXiv:1711.10337*, 2017.

MNIH, V. et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

PEROZZI, B.; AL-RFOU, R.; SKIENA, S. Deepwalk: Online learning of social representations. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.: s.n.], 2014. p. 701–710.

POLISHCHUK, P. G.; MADZHIDOV, T. I.; VARNEK, A. Estimation of the size of drug-like chemical space based on gdb-17 data. *Journal of computer-aided molecular design*, Springer, v. 27, n. 8, p. 675–679, 2013.

REGO, N.; KOES, D. 3dmol. js: molecular visualization with webgl. *Bioinformatics*, Oxford University Press, v. 31, n. 8, p. 1322–1324, 2015.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. *Learning internal representations by error propagation*. [S.l.], 1985.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. *nature*, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986.

SCHLICHTKRULL, M. et al. Modeling relational data with graph convolutional networks. In: SPRINGER. *European semantic web conference*. [S.l.], 2018. p. 593–607.

SCHULMAN, J. et al. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

SEGLER, M. H. et al. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, ACS Publications, v. 4, n. 1, p. 120–131, 2018.

SHI, C. et al. Graphaf: a flow-based autoregressive model for molecular graph generation. *arXiv preprint arXiv:2001.09382*, 2020.

SIMONOVSKY, M.; KOMODAKIS, N. Graphvae: Towards generation of small graphs using variational autoencoders. In: SPRINGER. *International conference on artificial neural networks*. [S.l.], 2018. p. 412–422.

SIPSER, M. Introduction to the theory of computation. *ACM Sigact News*, ACM New York, NY, USA, v. 27, n. 1, p. 27–29, 1996.

SUTTON, R. S.; BARTO, A. G. *Reinforcement learning: An introduction*. [S.l.]: MIT press, 2018.

VAPNIK, V. Principles of risk minimization for learning theory. In: *Advances in neural information processing systems*. [S.l.: s.n.], 1992. p. 831–838.

WEININGER, D. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, ACS Publications, v. 28, n. 1, p. 31–36, 1988.

WENG, L. Flow-based deep generative models. *lilianweng.github.io/lil-log*, 2018. Disponível em: ⟨http://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html⟩.

WERBOS, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, IEEE, v. 78, n. 10, p. 1550–1560, 1990.

WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, Springer, v. 8, n. 3, p. 229–256, 1992.

WILLIAMS, R. J.; ZIPSER, D. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , v. 1, n. 2, p. 270–280, 1989.

YOU, J. et al. Graph convolutional policy network for goal-directed molecular graph generation. *arXiv preprint arXiv:1806.02473*, 2018.

YOU, J. et al. Graphrnn: Generating realistic graphs with deep auto-regressive models. In: PMLR. *International conference on machine learning*. [S.l.], 2018. p. 5708–5717.

YU, L. et al. Seqgan: Sequence generative adversarial nets with policy gradient. In: *Proceedings of the AAAI conference on artificial intelligence*. [S.l.: s.n.], 2017. v. 31, n. 1.

ZHANG, D. et al. The ai index 2021 annual report. *arXiv preprint arXiv:2103.06312*, 2021.

ZHOU, Z. et al. Optimization of molecules via deep reinforcement learning. *Scientific reports*, Nature Publishing Group, v. 9, n. 1, p. 1–10, 2019.