Ludwig-Maximilians-Universität München
Faculty for Mathematics, Computer Science and Statistics

Master's thesis

De novo drug design in continuous space

Author:

Tuan Le

Supervisor:

Prof. Dr. Ulrich Mansmann [LMU]

Dr. Roman Hornung [LMU]

Dr. Djork-Arné Clevert [Bayer AG]

M.Sc. Robin Winter [Bayer AG]

11th November 2019

Abstract

Finding novel compounds with favorable properties is an essential step in the drug discovery process. *In-silico de-novo* drug design seeks to generate novel chemical compounds, tailored to very specific healthcare needs using computational methods.

Recently, much work has been done to utilize generative models to generate and enrich molecular libraries with compounds that satisfy specified biochemical and physicochemical properties. Most state-of-the-art generative models in drug discovery utilize the capabilities of deep neural networks and many work with string-based representations of compounds. In contrast to most recent state-of-the-art generative models, we use a continuous vector representation of compounds that was learned by unsupervised pre-training.

The main goal of this thesis is to develop and benchmark generative adversarial networks (GANs) that learn the continuous data distribution of ChEMBL, a large chemical database of already synthesized compounds.

First, we show that our GAN is able to learn the distribution of compounds in ChEMBL while generating novel and diverse compounds, and that it is competitive against other state-of-the-art methods when compared in the GuacaMol benchmark, which is a standardized evaluation framework for de novo generative models.

Next, we address the main goal in *de novo drug design* to generate chemical libraries with compounds that satisfy specific physicochemical properties. We optimize our GAN to generate compounds that are very drug-like by maximizing a single metric called the `QED` (Quantitative Estimate of Druglikeness). Our final GAN model is able to generate novel and diverse molecules with high `QED` values.

Contents

1 Introduction

With the rise of big data and deep neural networks, new techniques for supervised machine learning, especially in computer vision and natural language processing, have shown to be very powerful and effective in their performance [Schmidhuber (2014); LeCun et al. (2015)]. Apart from supervised learning, the task of unsupervised learning such as generation of data following a given distribution, e.g. images of dogs or cats, is a lively area of machine learning research [Guzel Turhan & Bilge (2018)].
Especially in the chemical and pharmaceutical field, generating novel compounds with desired properties to cure diseases is a challenging task. *De novo drug design* is complex due to the large chemical space. The space of drug-like molecules is estimated to be on the order 10^{23} to 10^{60} [Polishchuk et al. (2013)].
Focused drug discovery is often described as finding a needle in a haystack [Olivecrona et al. (2017)]. Finding that needle often entails satisfying constraints that drug-like compounds should fulfill. For instance, the compounds should be active against a biological target and/or have melting temperature within a defined range. Being active against a biological target means for example, that a compound binds to a protein, which causes an effect in the living organism, or inhibits replication of bacteria [H. S. Segler et al. (2017)]. There exists a plethora of biological as well as physicochemical properties that bias the generation process.
In general, the lifecycle of drug discovery can take many years that last at least a decade (10-20 years) [Brown (2009); Sanchez et al. (2017)]. The regular procedure of drug discovery follows a set of common stages, shown in Figure 1.

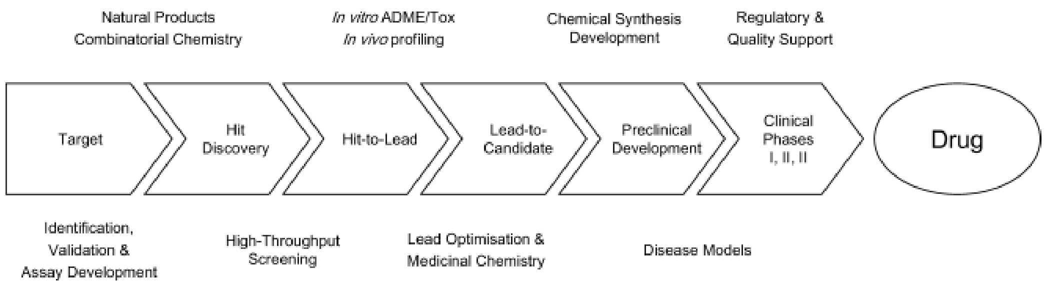


Figure 1: An illustration of typical workflow of a drug discovery endeavor. Source: Brown (2009)

First, a biological target, for example a protein that is part of a disease pathway, is selected and screened against a large *chemical library* of compounds in a hit discovery experiment to identify *hits*. Hits are compounds with an adequate (but usually weak) activity on the selected biological target. Hit discovery is usually conducted in High-Throughput-Screening (HTS), a method where thousands of experiments are conducted in parallel *in vitro*, on actual physical plates with many wells [Brown (2009)]. Each of these wells contain a compound and some biological matter of experimental interest, such as protein, cells or an animal embryo. If a desired response is observed, then the compounds that were tested are referred to as hits. In the following hit-to-lead step, a number of *leads* from the hits are discovered with various profiling analyses to determine whether any of these compounds are suitable for the biological target of interest. The leads can then be converted to *candidates* by optimizing on the biological activity and other objectives of interest, such as molecular weight or solubility. Once suitable candidates have been designed, the candidates enter the next step of preclinical development.

Generative models for focused library design aim to automatically generate *large chemical libraries* that contain a high number of hits and leads. By achieving the aforementioned, the upcoming steps of drug discovery are accelerated and navigating in (drug-like) chemical space to identify synthesizable compounds can be performed more efficiently.

It is estimated that an average drug discovery process costs between one [Brown (2009)] and three billion dollars [Schneider (2019)]. Hence, accerelating the drug discovery process with powerful generative models to enrich chemical libraries of compounds is also highly motivated to reduce costs, e.g. less in vitro HTS experiments conducted.

1.1 Generative Models in Drug Discovery

A generative model is a powerful tool for learning any kind of data distribution using unsupervised learning methods. All variants of generative models aim at learning the true data distribution of a training set, in order to sample new data points from this learned distribution. With the preceding rise of deep learning, many new methods for generative models in the field of image-, text- or music generation have emerged [Kingma & Welling (2013);Goodfellow et al. (2014)],[Graves (2013);Fedus et al. (2018)], [Mogren (2016);Yu et al. (2017)]. Those methods rely either on convolutional neural networks (CNNs), when dealing with images, or recurrent neural networks (RNNs) for sequential data such as text or music.

Due to active research in generative modeling, especially generative adversarial networks (see Section 2.4), new methods have also emerged in the field of computational chemistry and de novo molecular generation.

As in any machine learning setting, the representation of data is crucial. Since we deal with chemical data in terms of molecules, the SMILES representation, a string-based representation derived from molecular graphs, is often used as representation for drug discovery generative models.

Within SMILES (described in Section 2.1.2), data lies in form of a sequence of characters and symbols corresponding to atoms and its bindings as well as special characters denoting opening and closure of rings and branches.

H. S. Segler et al. (2017) trained a recurrent neural network (RNN) on the large chemical database ChEMBL [Mendez et al. (2018)] as language model, with the objective to predict the next character conditioned on the previous seen characters using maximum likelihood estimation to generate drug-like molecules. One character can be defined as an atom, except for atom types that comprise two characters such as '`Cl`' or '`Br`'. In addition, character symbols for bonding, branches and ring-openings/closings as well as disconnected structures are included {`-,+,=,#,:,(,),[,], d+`} in the SMILES vobaculary, where `d+` means digits between 0 and 9.
By applying transfer learning [Weiss et al. (2016)], novel compounds satisfying a biological target, such as being active towards 5-HT2A-receptor, could be generated. The method introduced by H. S. Segler et al. (2017) is described in Section 2.2.6.2 in detail.

Olivecrona et al. (2017) also used a RNN as sequence-based generative model to first learn the training set of ChEMBL and then fine-tune another RNN to bias the network to generate compounds with specified desirable properties using the policy gradient algorithm from reinforcement learning [Sutton & Barton (1998)].
In reinforcement learning (RL) a problem is defined as Markov Decision Process (MDP). The MDP consists of (discrete) states and (discrete) actions that can be conducted given a current state. Following an action in a given state returns a reward for choosing that action. The final goal in RL is to maximize the expected reward. For the SMILES RNN language model, possible actions are defined to be symbols of the SMILES vocabulary and the state can be defined as the current SMILES sequence obtained. Since a RNN language model outputs a probability distribution over possible characters conditioned on the previous seen characters, Olivecrona et al. (2017) fine-tuned the pretrained RNN model to maximize the expected reward by updating its policy, which is a probability distribution over actions given a state from RL theory, to generate compounds satisfying certain properties. The properties, which the generated SMILES should satisfy, were the absence of sulfur atoms (`S`), activity towards dopamine receptor type 2, as well as high similarity to the drug *celecoxib*. However, those three properties were fine-tuned as single-optimization tasks in three single steps resulting to three RNN models.

Gómez-Bombarelli et al. (2016) proposed a variational autoencoder [Kingma & Welling (2013)] to encode discrete SMILES representation of molecules to a multidimensional continuous (latent) representation that comprises an *information bottleneck* and from this latent representation decode it back to the SMILES representation. Generating new molecules is done via variational inference by sampling from the distribution in the latent space. The main idea of autoencoders will be discussed in Section 2.3.

As generative adversarial networks (see Section 2.4 for a detailed explanation of GANs) have been mostly proposed for learning data with continuous and dense representations, Yu et al. (2017) introduced *seqGAN*, a new methodology to train a GAN with sequential data, e.g. SMILES representations, using reinforcement learning. In GAN, a generator network is guided through a discriminator network by learning from the discriminator‘s feedback. Because the sampling process of the next character or sequence for the generator is discrete by using the multinomial distribution, the sampling process is not differentiable.
Hence, in GAN it is impossible to pass gradient updates from the discriminator network to the generator network. Therefore classical gradient-based methods cannot be applied (see Section 2.2.4 and 2.2.5 for an overview on optimization). The direct application to molecular generation with the *seqGAN* algorithm using SMILES notation was executed in *ORGAN* and *ORGANIC* models [Guimaraes et al. (2017); Sanchez et al. (2017)].

Instead of using SMILES representation, Cao & Kipf (2018) directly used the two-dimensional molecular graph as data input to train a GAN, called *MolGAN*. Their proposed method is trained in combination with reinforcement learning to encourage the generation of molecules with specific desired properties. The generative model of *MolGAN* predicts discrete graph structure at once, i.e. non sequentially.

Zhou et al. (2019) introduced *Molecule Deep Q-Networks (MolDQN)* for molecule optimization by combining domain knowledge of chemistry and state-of-the-art reinforcement learning algorithms. The data representation they work with is the SMILES notation. By defining the generation of a molecule as a Markov Decision Process (MDP) with possible three valid actions: (1) atom addition, (2) bond addition and (3) bond removal, the molecule generated is only dependent on the molecule being changed and modifications to be made.
The authors claim to directly operate on the molecular generation without validating the SMILES grammar by defining a set of valid actions given a current state.
Additionally, the authors claim that their framework has the benefit that no pre-training of the generative model is needed in contrast to Olivecrona et al. (2017). The goal of multi-objective optimization of properties simultaneously is also included in their framework.

This work combines several unsupervised learning techniques utilizing the capability of deep neural networks to learn a continuous chemical space of molecules/-compounds. If we think of compounds as discrete string representations, following a certain grammar and vocabulary of characters, such as the SMILES grammar

[Weininger (1988)], to my best knowledge, most generative models work in a discrete space. The generating process of those models can be summarized by a probabilistic model that samples the next character conditioned on the previous sequence.
If we imagine the chemical space of compounds as a compact continuous space $\mathcal{C} \subset \mathbb{R}^k$ that comprises a probability density, the goal is to learn this probability density in order to sample new observations from this respective probability density. Therefore, this study aims to tackle following subsequent unsupervised learning goals:

1. Description of a method to learn a continuous space $\mathcal{C}$ for compounds using unsupervised learning techniques [Winter et al. (2018)].

2. Once a training set of compounds, encoded in this continuous space $\mathcal{C}$ is given, the goal is to learn a probability distribution over this training set.
   The main algorithm will be a GAN that can model the true training data distribution within its respective domain space $\mathcal{C}$.

3. Fine-tuning of the learned GAN such that it is able to synthesize new compounds that satisfy certain physico- and/or biochemical properties.

Since this work is mainly utilizing deep neural networks to learn a chemical space, fundamental network classes, namely feedforward neural networks and recurrent neural networks are explained in Section 2.2.1 and their application in drug discovery displayed in Section 2.2.6.2.
As one objective of this work is to learn and obtain a continuous vector representation of compounds, the rationale and theory of an autoencoder is explained in Section 2.3. Followed by that, the idea of generative adversarial networks will be presented in Section 2.4, which are powerful methods to model the probability distribution of a given training set.
In the application part in Section 4.3, the training of a generative adversarial network on a large dataset, extracted from the chemical database 'ChEMBL' [Mendez et al. (2018)], will be described. Furthermore, this trained GAN will be compared to state-of-the-art models in drug discovery using the GuacaMol benchmark framework. Section 4.4 of this work describes the fine-tuning of the trained GAN in order to synthesize new compounds that satisfy certain physicochemical properties.

2 Theoretical Framework

Notation In the context of machine learning and probability theory, in this work $\mathcal{X}$ denotes a $p-$dimensional input space. Usually we assume $\mathcal{X} = \mathbb{R}^p$. For the prediction task, we will denote $\mathcal{Y}$ as target space, where $\mathcal{Y} = \mathbb{R}$ or $\mathcal{Y} = \mathbb{R}^k$, stating univariate or k-multivariate regression respectively. Since many machine learning algorithms are formulated as classification tasks, the target can be either $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{1, ..., n_c\}$, stating binary or n_c-class classification, hence $\mathcal{Y} \subset \mathbb{N}_0$.
In case we obtain a dataset of N samples/observations, $x^{(i)} = (x_1^{(i)}, ..., x_p^{(i)})^\mathsf{T} \in \mathcal{X}$ denotes the $i-$th feature representation from the input/domain space and $y^{(i)}$ the $i-$th true target belonging to its corresponding feature.
In conclusion, the entire dataset will be noted as $D = \{(x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)})\}$.
From a probability theoretical view x is a realisation of the random variable X with domain $\mathcal{X}$. Hence, $\mathbb{P}_x$ is the probability distribution on $\mathcal{X}$, concluding $X \sim \mathbb{P}_x$ (sample x which is drawn from X, comes from the distribution $\mathbb{P}_x$).
Similarly $\mathbb{P}_{x,y}$ is the joint probability on the domain space $\mathcal{X} \times \mathcal{Y}$. In this work $p_X(x)$ stands for the probability density function (pdf) of the random variable X for one sample $x \in \mathcal{X}$. This work will not use **bold** representation of vectors and matrices.

2.1 Molecular Representation

Molecules are complicated real-world objects and the molecular representation refers to the computer-interpretable (digital) encoding used for each molecule/compound. 'In cheminformatics, the most popular representation is the two-dimensional (2D) chemical structure (*topology*) with no explicit geometric information' [Brown (2009)]. This representation is the **2D connectivity graph** chemists draw to describe a molecule, from which string-based *line notations* were derived. The 2D connectivity graph, called Lewis structure in chemistry, is a molecular graph, in which atoms are shown as labeled nodes. The edges describe the bonds between atoms, which are labeled with the bond order (e.g. single, double or triple).
Another way of representing molecules is to use geometric information by using **3D geometry coordinates** of molecules. This method though, is not widely used in predictive modeling due to the fact that coordinates are not invariant to molecular translation, rotation and permutation of atomic indexing [Elton et al. (2019)].
This change of coordinates of a molecule is described as *conformer problem* in computational chemistry. Since molecules are three-dimensional objects connecting atoms together, a *conformer* of a molecule is a single geometric arrangement of atoms in a molecule. However, a molecule may adopt infinite conformations because it interacts with the (natural) system in its environment and therefore can change its conformation.
One possible way to generate a conformer or multiple possible conformers is the

minimum-energy conformation, a conformation in which the geometric arrangement of atoms leads to a global minimum in the internal energy of the system [Pearlman (1987)].

Molecular Descriptors

The generation of informative data from molecular structures in a so-called *molecular descriptor* is called featurization [Elton et al. (2019)] and plays an important role in cheminformatics because those descriptors are often the 'precursor to permitting statistical analyses of the molecules' [Brown (2009)] or predictive modeling tasks. Hence, a molecular descriptor is mostly a computer-interpretable vector of numbers capturing the most salient information of the molecule. Chemical information can be characterized by experimental measurements, e.g. physicochemical properties such as molecular weight, hydrogen bond acceptors (HBA) or hydrogen bond donors (HBD) measurements. Those quantities can be calculated easily *in-silico*[1] as a function of the available atoms within the molecule. For example, molecular weight is simply the summation function according to the numbers and types of atoms that are present in the molecule under consideration. The HBA and HBD can be computed by counting the number of nitrogen (N) and oxygen (O) and NH and OH groups, respectively [Brown (2009)]. Combining all those physicochemical descriptors together into one vector leads to the molecular descriptor.
Other vector representations considering the configuration of atoms, based on molecular structure-key fingerprints are also widely used as exemplified in Figure 2.

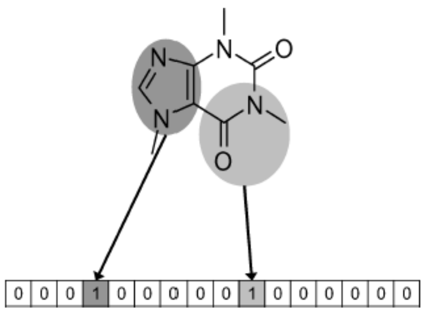


Figure 2: An example of the encoding of a simple molecule as a structure-key fingerprint using a defined substructure dictionary. A defined substructure is assigned a single bit position on the string to which it is mapped or not. Source: Brown (2009)

Molecular fingerprints encode structural or functional features of a molecule in a bit string format and are commonly used for tasks like virtual screening[2], similarity searching and clustering [Willett et al. (1998); Cereto-Massagué et al. (2014)]. The structure-key fingerprint uses a dictionary of defined substructures to generate a binary vector, where each bit in the vector equates to a one-to-one mapping between

[1]*In-silico* means that a procedure has been performed in a computer.

[2]Virtual screening is a computational technique used in drug discovery to search libraries of small molecules in order to identify those structures which are most likely (true/false) to bind to a drug target, typically a protein receptor or enzyme [Gillet (2013)] using a predictive model.

the molecule and a substructure in the dictionary for presence or absence. Since the number of potential substrucures can be large ($\approx 2^{32}$), the resulting sparse set of bits is usually hashed and folded to a much smaller size ($\approx 10^3$) at the expense of hash and bit collisions [Rogers & Hahn (2010)].
A way to obtain an informative continuous vector representation of compounds by utilizing the power of unsupervised learning methods is described in Section 2.3.1. The focus in this Section will lie on the InCHI and SMILES representation of molecules that are both derived from the 2D molecular graph.

2.1.1 InCHI Representation

The InCHI (International Chemical Identifier) [Heller et al. (2015)] notation is a unique string-based representation of ASCII characters divided into layers and sublayers providing different types of information such as the chemical formula, bonds and charges. The InCHI notation allows to describe a molecule in a very compact form but is not intended for readability [Brown (2009)].
An example InCHI representation of *caffeine* is provided in Figure 3.

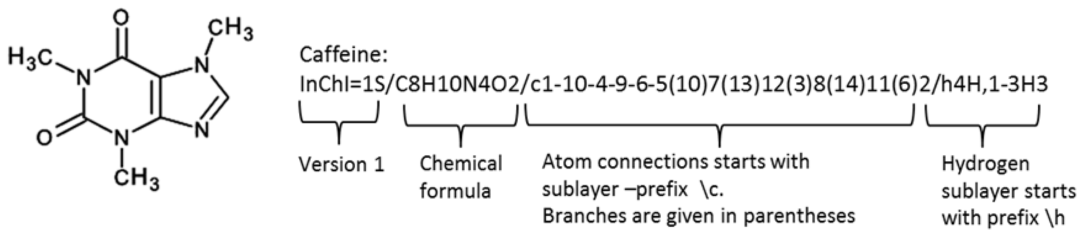

Caffeine:
InChI=1S/C8H10N4O2/c1-10-4-9-6-5(10)7(13)12(3)8(14)11(6)2/h4H,1-3H3
Version 1
Chemical formula
Atom connections starts with sublayer –prefix \c. Branches are given in parentheses
Hydrogen sublayer starts with prefix \h

Figure 3: InCHI representation of caffeine $C_8H_{10}N_4O_2$.

2.1.2 SMILES Representation

The SMILES (Simplified Molecular Input Line Entry System) [Weininger (1988)] notation is a non-unique representation that encodes the molecular graph into a string-based sequence of ASCII characters. In contrast to InCHI, the SMILES notation is not divided into different information layers but encodes the entire molecular structure into one sequence including identifiers for atoms as well as identifiers denoting topological features like bonds, rings, branches and cycles.
SMILES is a chemical notation language specifically designed for computer use by chemists and has become popular because it represents molecular structure by a linear string of symbols, similar to natural language [Weininger (1988); Brown (2009)]. Hydrogen atoms (H) may be ommited (hydrogen-suppressed graphs) or included (hydrogen-complete graphs).
The simplified topological encoding system consists of several rules [Weininger (1988)].

(1) Atoms. Atoms are represented by their atomic symbols. This is the only required use of letters in SMILES. Each non-hydrogen atom is specified independently by its atomic symbol enclosed in square brackets `[,]`. The second letter of a two-character symbol must be entered as lower case, such as for the chlorine (`Cl`) or bromine (`Br`) atom. Note that this states one entity and is therefore one token in terms of language-modeling. Elements in the defined 'organic subset', {`B, C, N, O, P, S, F, Cl, Br, I`} may be written without brackets if the number of attached hydrogens conforms to the lowest normal valence consistent with explicit bonds. Atoms in aromatic rings are specified by lower case letters; e.g., normal carbon is presented by the character `C`, aromatic carbon by `c`. As attached hydrogens are implied in the absence of brackets for the elements of the organic subset, the first four atomic symbols in Table 1 are valid SMILES.

SMILES	**Name of atom or molecule**
`C`	methane (CH_4)
`N`	ammonia (NH_3)
`O`	water (H_2O)
`Cl`	hydrogen chloride (HCl)
`[Cl]`	chlorine atom (Cl)
`[C]`	carbon atom (C)
`[Au]`	element gold (Au)
`[H+]`	proton
`[OH-]`	hydroxil anion
`[NH4+]`	ammonium cation
`[Fe+2]`	iron(II) cation

Table 1: Displayed are SMILES, where the hydrogen number conforms to the lowest normal valence (row one to four), SMILES representing single atoms (row five to seven) and SMILES, where charges had been made (row eight to eleven). Source: Weininger (1988)

Attached hydrogens and formal charges are always specified inside the brackets, where the number of attached hydrogens is shown by the symbol `H` followed by an optional digit. Formal charges on the atom itself without hydrogen attachements, are shown similar by one of the symbols `+` or `-` followed by an optional digit. Examples to display charges are listed in Table 1 rows eight to eleven.
If unspecified, the number of attached hydrogens and charges is assumed to be zero for an atom inside the bracket as shown in the rows five to seven in Table 1.

(2) Bonds. Single, double, triple and aromatic bonds are represented by the symbols {-, =, #, :}, respectively. Single and aromatic bonds are usually omitted.

SMILES	Name of atom or molecule
`CC`	ethane (CH_3CH_3)
`C=C`	ethylene ($CH_2 = CH_2$)
`CCO`	ethanol (CH_3CH_2OH)
`O=C=O`	carbon dioxide (CO_2)
`C#N`	hydrogen cyanide (HCN)
`[H][H]`	molecular hydrogen (H_2)

Table 2: SMILES displayed with single (rows one, three and six), double (rows two and four) and triple bonds (row five). Source: Weininger (1988)

(3) Branches. Branches are encoded by round parentheses (,) surrounding the branching fragment, which may be nested or stacked, as illustrated in Figure 4.

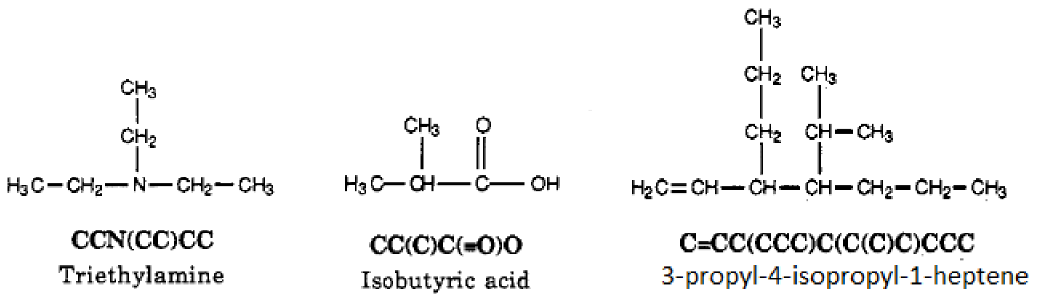


Figure 4: Illustration of branches in SMILES notation. The first two SMILES representations show topologies with branches that are not nested. The third SMILES representation has a branch that is nested. Source: Weininger (1988)

(4) Cyclic Structures. Cyclic structures are represented by breaking one single (or aromatic) bond in each ring. The bonds are numbered in any order, designating ring-opening (or ring-closure) bond by a number immediately following the atomic symbol at each ring closure. The result is a connected noncyclic graph, which is written as a noncycled structure by using the three rules described above. One example for describing a cyclic structure in SMILES representation is displayed in Figure 5.

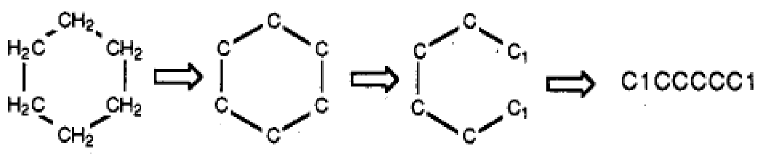


Figure 5: Cyclohexane represented in SMILES notation breaking the ring at a position and closing the ring. The integer number stands for ring-opening and ring-closure. Source: Weininger (1988)

Since some atoms in a cyclic structure might have different ring-closures, different SMILES notation for one cyclic structure can be derived. Therefore, the SMILES representation is non-unique as mentioned in the beginning of this Section and illustrated in Figure 6. When breaking the ring in Figure 6, the rule of branches is applied differently but leading to valid SMILES, depending on which atom lies in the 'main'-chain and which substructure is considered to be a branch (rule 3), embodied in the parentheses.

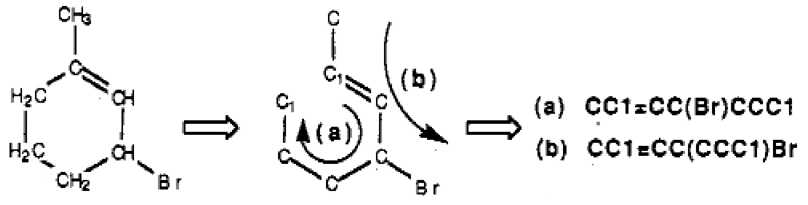


Figure 6: 1-methyl-3-bromo-cyclohexene can have different ring-openings and ring-closures leading to different SMILES representations. Here, the ring-opening and ring-closure is the same for both valid SMILES representation but the way how to 'read' the SMILES and define the branch is differently. Naturally, representation (a) is the simplest. Source: Weininger (1988)

(5) Disconnected Structures. Disconnected molecules are written as individual structures seperated by a period. This is important since single bonds are implicit, and showing the dependency between ions and ligands (molecules) has to be guaranteed. If desired, the SMILES of one ion may be imbedded within another, as shown in the example in Figure 7.

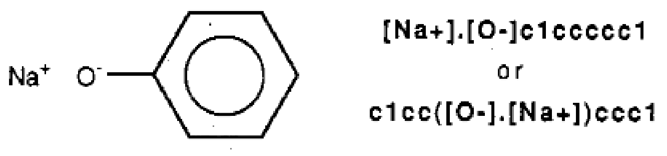
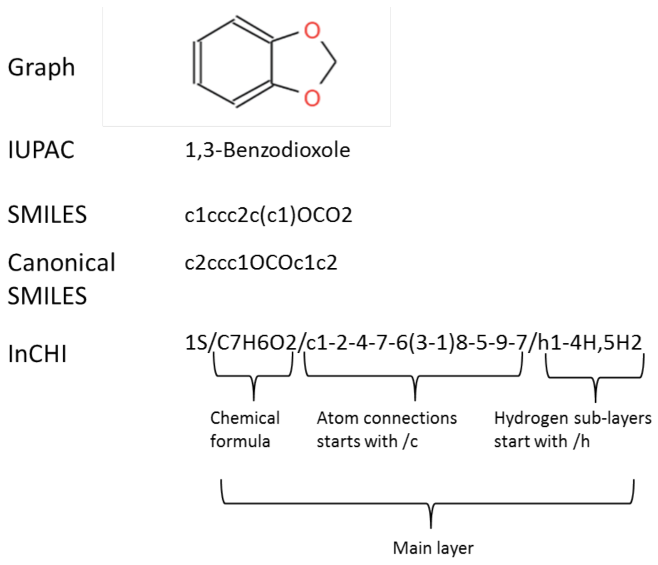


Figure 7: SMILES representation for sodium phenoxide, where one natrium ion is connected to the ligand that contains the benzene ring. Here, the rules (1:atoms) and (5:cyclic structures) are combined. Recall that the carbon atoms are included in the (broken) aromatic ring and therefore written in small letters `c`. Source: Weininger (1988)

As described in the last two examples, one drawback of the SMILES notation is the lack of unique representations. The reason for the non-uniqueness lies in the fact that a molecule with no (aromatic) ring can be encoded from any starting point of the topological graph. For example, the molecule ethanol has following four valid SMILES representations: `CCO`, `OCC`, `C(C)O` and `C(O)C`. When dealing with rings or disconnected structures, non-unique SMILES representation can occur as well, depending on where the opening of the ring is executed (see Figure 6) and how the order of nested connection is set (see Figure 7).

For that reason, several canonicalization algorithms have been developed, such as the Morgan algorithm [Morgan (1965)] to create unique SMILES, which are called *canonical SMILES*.

The upcoming Figure 8 shows different molecular representations of the 1,3 - Benzodioxole molecule.



Graph	
IUPAC	1,3-Benzodioxole
SMILES	c1ccc2c(c1)OCO2
Canonical SMILES	c2ccc1OCOc1c2
InCHI	1S/C7H6O2/c1-2-4-7-6(3-1)8-5-9-7/h1-4H,5H2

O
O
Chemical formula
Atom connections starts with /c
Hydrogen sub-layers start with /h
Main layer

Figure 8: Different sequence-based molecular representations of 1,3-Benzodioxole. Modified Source: Winter et al. (2018)

2.2 Deep Learning

Neural networks (NNs) are considered as a part of artificial intelligence (AI) and designed as an attempt to simulate the human nervous system [Aggarwal (2018)]. In recent years, deep learning has steadily increased in popularity, mainly due to their state-of-the-art performance in image and speech recognition, text mining and other related tasks. Deep neural networks endeavor to automatically learn multi-level representations and features of (large) data and are able to uncover complex underlying data structures.
The general aim of supervised learning, is to approximate a function f that is used to predict an outcome y, using an input x, i.e. $y \approx f(x)$. Nearly all supervised learning algorithms can be described by three components [Domingos (2012)]:

$$\textbf{Learning = Representation + Evaluation + Optimization.}$$

In classical machine learning, one tries to find a mapping from **feature** to **output**, where the performance heavily depends on the representation of the feature data. Hence, traditional machine learning is also called **feature learning**.
To improve the performance of a learning algorithm, instead of discovering the mapping from representation to output, one can also tackle the task of learning the representation itself. This approach is also known as **representation learning**. Learned representations often result in much better performance than can be obtained with hand-designed representations (e.g. feature engineering) [Goodfellow et al. (2016)]. In neural networks, new features are represented as intermediate neurons, called *hidden neurons*. The basic idea is to apply many *simple operations* consecutively to build a **computational graph**. These simple operations are explained in the upcoming Section 2.2.2.
The term *deep learning* was formulated from the idea of building large computational graphs, e.g. applying/stacking many simple operations one after another for the final prediction task.
In general, three major classes of neural networks exist:
feedforward neural networks (see Section 2.2.1), convolutional neural networks, which are mostly used when working with images (not covered in this work) and recurrent neural networks, mostly used when dealing with sequential data (see Section 2.2.6).

2.2.1 Feedforward Neural Network

The quintessential example of a deep learning model is the feedforward neural network, or **multilayer perceptron** (MLP). A multilayer perceptron is simply a mathematical function mapping some input values to output values, making use of the idea of computational graphs. The function is formed by composing many simple functions. We can think of each application of a different mathematical function as providing a new representation of the input [Goodfellow et al. (2016)].

Figure 9 displays an example a feedforward neural network.

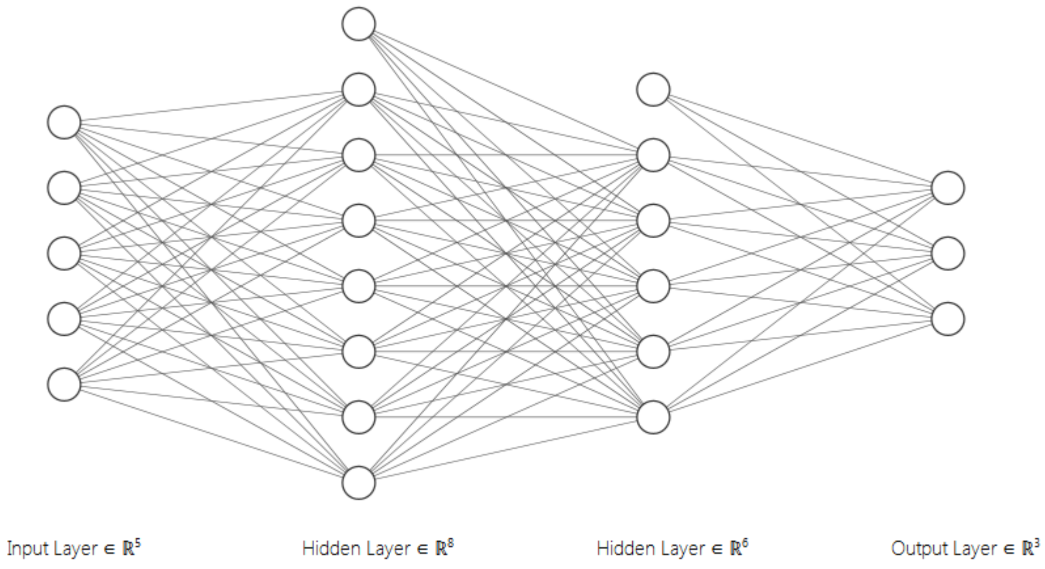


Figure 9: Feedforward neural network with two hidden layers.

MLPs are also called **fully-connected** neural networks because the output of each neuron in one layer is fed into each neuron in the next layer. For MLPs, there are no feedback connections allowing outputs of the model to be fed back to itself. If the output of a model should be inserted as input in the input layer, one model class would be **recurrent neural networks** (RNNs, explained in Section 2.2.6), which deliver state-of-the-art performances in natural language processing (NLP) tasks, e.g. speech recognition, automatic language translation etc. and generative drug discovery as will be explained in Section 2.2.6.2.

In MLP, information flows from the input (forward) from one layer to the following layer, until it reaches the final output layer (feedforward network). As mentioned in the beginning of this section, a MLP is composing n_l mathematical functions $f^{(1)}, f^{(2)}, ..., f^{(n_l-1)}, f^{(n_l)}$ in a chain altogether, where n_l is the total number of layers in the network. The output of the MLP can be expressed by linking these functions in one entire chain of layers,

$$f(x) = f^{(n_l)}(f^{(n_l-1)}(...(f^{(2)}(f^{(1)}(x)))...)), \tag{1}$$

where $f^{(1)}$ is called the *input layer* that takes a feature point x as input. The layers $f^{(2)}, f^{(3)}, ..., f^{(n_l-1)}$ are called *hidden layers* since their outputs are not directly accessible or interpretable in the context of a specific prediction task. The hidden layers are applied in order to model the complex relationships between the input feature x and the target variable y at the last (output) layer. Therefore, the function $f^{(n_l)}$ is called *output layer* that contains the final result for the prediction task.

The more functions (layers) the neural network contains, the deeper it gets, leading to the terminology of *deep neural networks*.

2.2.2 Basics and Building Blocks

Neural networks contain computation units which we will call neurons. The computational units are connected through weights that symbolize the strengths of synaptic connections in biological organisms.
The classical neural network contains three different types of layers: input layer, hidden layers and output layer.
The input layer takes the input x and propagates it to the upcoming first hidden layer. The hidden layers do all the processing for neural networks. Generally speaking, the more hidden layers the network has, the more accurate the network will be on a given training set. However, the problem of overfitting the training data occurs.
Each hidden layer can be thought of a non-linear transformation of in-going data. For this non-linear transformation, every neuron performs a two-step computation (earlier mentioned as *simple operations*) [Bischl (2018*a*)].

1. Compute the weighted sum of inputs (with bias). This operation only includes multiplication and summation. We will call this result *pre-activations* z.

2. Apply an activation function $\phi(\cdot)$ to each element of z. This is used for non-linear transformation of the input. We will call this output *activations*[3], hence $a = \phi(z)$.

2.2.2.1 Weight Matrices and Biases

Weight matrices and bias vectors are learnable parameters that will be adjusted during training of the neural network.
The weight matrices have the purpose to apply linear transformation to the incoming data from the current layer to the upcoming layer by computing a dot product of incoming data and weight matrix. The bias has the purpose to shift the weighted sum in the upcoming layer.
Assume the neural network contains l hidden layers, leading to a total of $(l + 1)$ weight matrices $W^{(l)}$, and bias vectors $b^{(i)}$, for $i = 0, ..., l$.
Concluding to a parametric model we obtain following learnable parameters
$\theta^{(i)} = \{W^{(i)}, b^{(i)}\},\ i = 0, ..., l$ in a neural network.
Note that $\theta^{(0)}$ and $\theta^{(l)}$ are the weight matrices and biases for the neural connections between input and first hidden layer and last hidden layer to output layer.
Let d_i be the dimensionality[4] of the $i-$th hidden layer.

[3]Often the results of the activations are also called *hidden states*.
[4]Dimensionality is in this case the number of neurons in the $i-$th hidden layer.

The dimension for each weight matrix depends on the number of neurons in the current layer and next layer. In general, one can say that $W^{(i)}$ is element of $\mathbb{R}^{d_i \times d_{i+1}}$, where d_i is the number of neurons in the current layer i and d_{i+1} is the number of neurons in the next layer $(i+1)$. Therefore, each *column* of the $(d_i \times d_{i+1})$ matrix corresponds to a single (hidden) neuron. The bias term $b^{(i)}$ is a $d_{i+1}-$dimensional column vector. Assuming we apply the identity function as activation function, $\phi(z) = z$, we can compute the *pre-activations* as follows for all $i = 0, ..., l$:

$$\underbrace{z^{(i+1)}}_{\in\mathbb{R}^{d_{i+1}}} = \underbrace{W^{(i)\intercal}}_{\in\mathbb{R}^{d_{i+1}\times d_i}} \underbrace{a^{(i)}}_{\in\mathbb{R}^{d_i}} + \underbrace{b^{(i)}}_{\in\mathbb{R}^{d_{i+1}}}, \text{ where } a^{(i)} = \phi(z^{(i)}) = z^{(i)}. \tag{2}$$

When explaining the training of the neural network in the upcoming Section 2.2.4, the weights and biases are updated in order to improve the performance of the deep neural network model. For the success of training and optimizing neural networks, it is vital to initialize the weight matrices and biases with useful values. It is common in practice, to randomly draw values for weights from a symmetric distribution that is zero-centered. A normal distribution $\mathcal{N}(\mu = 0, \sigma^2)$ satisfies this condition. The *Xavier Initialization Rule* [Glorot & Bengio (2010)] suggests to draw the elements of the weight matrices $W^{(i)}$ from $\mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{d_i})$, where d_i is the number of neurons in the $i-$th layer. The bias vectors should be initialized with 0 or very small values such as 0.01.

2.2.2.2 Activation Functions

The activation function has the purpose to incorporate non-linearity of incoming data. To amplify this thought, one can think of a simple binary classification problem. In many machine learning algorithms such as logistic regression, the goal is to find a linear hyperplane to discriminate/seperate data points into two classes. Assume the data points lie in $\mathbb{R}^2$ and the two classes are **not** linearly seperable. In this case logistic regression will fail to classify all samples correctly. By transforming the data points to a hidden representation, for example from cartesian to polar coordinates, the transformed data points are linearly seperable as shown in Figure 10.

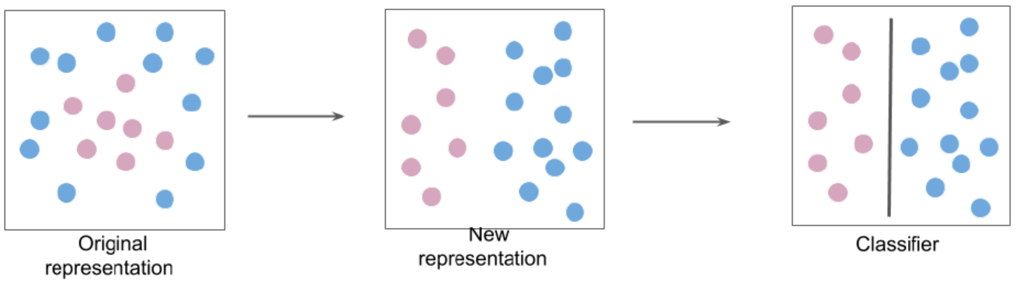


Figure 10: In the original representation, there exists no linear line to perfectly discriminate between the two classes (red and blue). If the original features are transformed into a new representation, what a neural network does in the hidden layers, the data might become perfectly linearly separable for a classifier. Source: Bischl (2018*a*)

The term activation function arises from models of biological neurons in the brain and defines the expected firing rate of the neuron as a function of the incoming signals at synapses [Dayan & Abbott (2005)]. Hence, the main purpose is to convert an input signal (weighted sum + bias) of a node into an (activated) output signal, where the output signal is then used as input for the next layer.

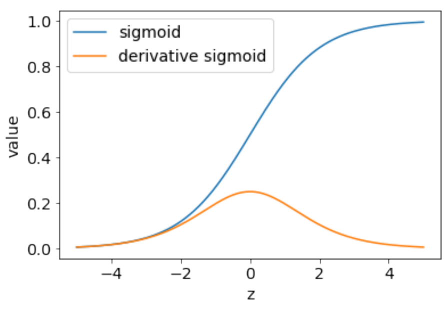
Note that all upcoming activation functions will be applied *element-wise* to each component of a real-valued vector z.

There are many different popular choices of non-linear activation functions (see Figure 11), for example the sigmoid function (also used as activation in logistic regression to compute *positive* class probability)

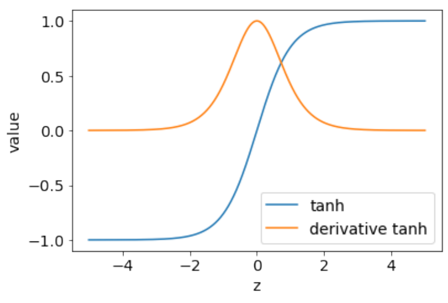
$$\sigma(z) = \frac{1}{1 + \exp(-z)}, \tag{3}$$

or the hyperbolic tangent function

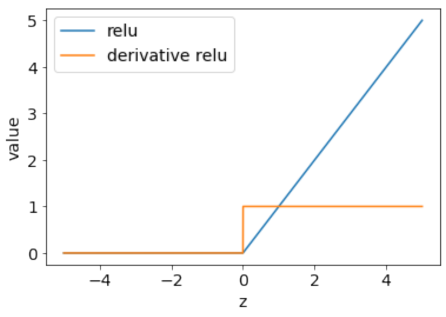
$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}. \tag{4}$$



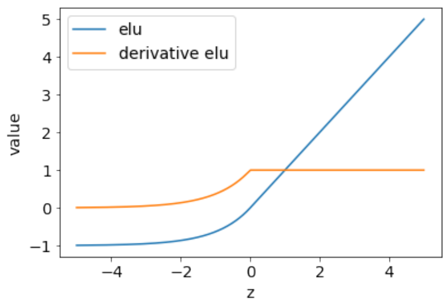
(a) sigmoid function and its derivative.



(b) tanh function and its derivative.



(c) ReLU function and its derivative.



(d) ELU function and its derivative.

Figure 11: Example activation functions often used in neural networks. Each subplot also displays the first derivative of the respective activation function.

Currently the most common activation function for deep neural networks is the rectifier linear unit (ReLU). The ReLU function was first introduced by Nair & Hinton (2010) in neural networks and is formulated as

$$\mathrm{relu}(z) = \max(0, z). \tag{5}$$

Before the usage of ReLU, most hidden layers of deep neural networks were activated using sigmoid or tanh. This has often caused the *vanishing gradient problem*[5] and led to slow convergence and little effect on the weight update when doing backpropagation (see Section 2.2.5.2). ReLU has beneficial properties [Goodfellow et al. (2016)] such a piecewise linearity which preserves many of the properties that make a linear model easy to optimize with gradient-based methods. Another popular activation function is the exponential linear unit (ELU) [Clevert et al. (2015)] that has been successfully applied in convolutional neural networks for image classification.

$$\mathrm{elu}(z) = \max(0, z) + \min(0, \alpha(\exp(z) - 1)), \text{ with default } \alpha = 1. \tag{6}$$

[5]It is one example of unstable behaviour when training deep neural networks. The vanishing gradient problem is caused when the neural network is unable to propagate useful gradient information from the output layer of the model, back to the layers near the input of the model. This is caused by the chain rule when multiplying partial derivatives (note that for example the derivatives of sigmoid and tanh are restricted to $(0, 0.25)$ and $(0, 1)$). If very small numbers $|\delta| > 0$ (partial derivatives in the last layers) are multiplied with each other, the product (in this case a partial derivative in the very first layers) will be very small. Hence, the weight update for any gradient-based method will not make any change.

This activation function does not have the *dying ReLU problem*[6] and is a combination of linear and non-linear function in one term, leading to better generalization.

In MLP, the output layer takes the input from the activations of the last hidden layer to do a prediction task (either regression or classification). Computing the weighted sums and activations from the input layer right up to the output layer leading to the prediction is called **forward pass**.
The MLP in Figure 9 contains two hidden layers and one output layer with three output neurons. Assume that the neural network is a classifier, where the output variable has three possible categories. We conclude that each output neuron o_i states the predicted class probability of belonging to class i for a given sample x.
Hence, $o_i = \mathbb{P}(y = i|x)$, $i \in \{0, 1, 2\}$.
To squash the output neurons into range $(0, 1)$ and guarantee that the sum of all output neurons equals to one, the *softmax* function will be used as an activation function for the output layer when dealing with a multi-class classification problem.

$$\text{softmax}(z_j) = \frac{\exp(z_j)}{\sum_k \exp(z_k)}. \tag{7}$$

2.2.2.3 Loss Functions

In supervised learning, the goodness of prediction $y = f(x|\theta)$ is measured by a *loss function* $L(y, f(x|\theta))$, where $f(x|\theta)$ is the model parameterized with θ.
The aim is to find an optimal θ that performs well on a training set but also generalizes well on an unseen test set. Good performance means to have a minimal *risk*. Hence, we face the folllowing optimization problem:

$$\min_\theta \mathcal{R}(f|\theta) = \min_\theta \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{P}_{x,y}} [L(y, f(x|\theta))] = \min_\theta \int L(y, f(x|\theta)) d\mathbb{P}_{x,y}. \tag{8}$$

The objective in equation (8) is not feasible or practical since the joint probability $\mathbb{P}_{x,y}$ is unknown. Instead, the risk can be approximated with the *empirical risk* based on a dataset D with N samples, which leads to the following optimization problem:

$$\min_\theta \mathcal{R}_{\text{emp}}(f|\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, f(x^{(i)}|\theta)). \tag{9}$$

Loss functions should include some relevant properties such as [Bischl (2019*b*)]:

1. Differentiability.

2. Robustness.

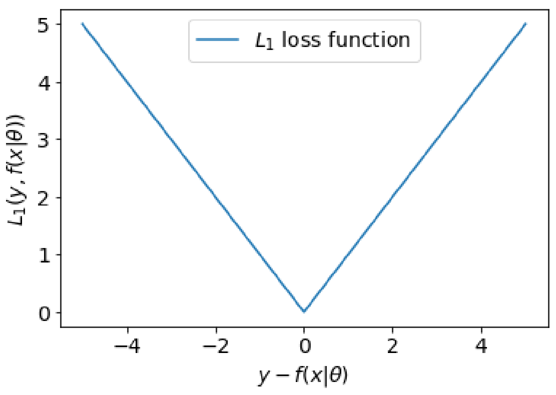
[6]The derivative of ReLu for values less than zero is equal to zero. Hence, no gradient information is propagated back when the input is negative as illustrated in Figure 11c.

3. Convexity.

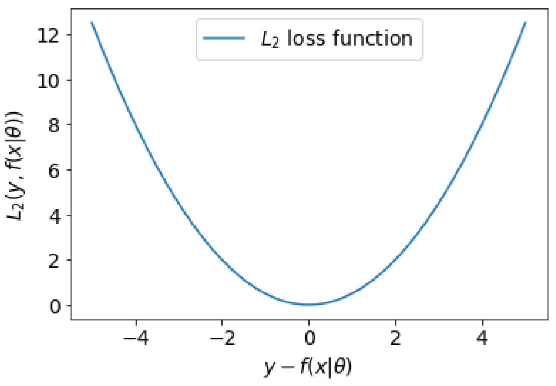
Differentiability is desired in order to optimize. Section 2.2.5 describes *gradient-based* approaches such as **gradient descent** which are used to train deep neural networks. Robustness shows how strong a loss function reacts to deviation of errors, i.e. $\epsilon = y - f(x|\theta)$, and convexity guarantees that a global minimum exists (this will in most cases not hold for deep neural networks as we want to model non-convex functions). In regression, L_1 and L_2 loss (shown in Figure 12) are usually used, leading to the following empirical risks $\mathcal{L}_1$ and $\mathcal{L}_2$ on a dataset D:

$$\mathcal{L}_1 = \frac{1}{N}\sum_{i=1}^{N} L_1(y^{(i)}, f(x^{(i)}|\theta)) = \frac{1}{N}\sum_{i=1}^{N} |y^{(i)} - f(x^{(i)}|\theta)|, \tag{10}$$

$$\mathcal{L}_2 = \frac{1}{N}\sum_{i=1}^{N} L_2(y^{(i)}, f(x^{(i)}|\theta)) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2}(y^{(i)} - f(x^{(i)}|\theta))^2. \tag{11}$$



(a) L_1 loss function.



(b) L_2 loss function.

Figure 12: Example of loss functions for regression task. The horizontal axis shows the deviance $\epsilon = y - f(x|\theta)$ of a model $f(x|\theta)$ w.r.t. a true target y corresponding to the feature x. The vertical axis shows the loss value for a given deviance/residual.

In binary or multi-class classification, one common loss function is the cross-entropy loss. If the neural network is a classifier, then the output layer consists of n_c neuron units, where n_c is the number of classes the target variable y can have. By introducing the *one-hot encoding*, we can derive a vector which assigns the class membership (indexed as 1/True and 0/False). In a classification task with n_c different classes c, the class label c_j of the i−th data point can be encoded by a label vector $y^{(i)}$ as stated below:

$$y^{(i)} = (l_1, l_2, ..., l_{n_c})^{\mathsf{T}},\ l_j = \begin{cases} 1, & \text{if } c_i = c_j \\ 0, & \text{else.} \end{cases} \tag{12}$$

This encoded vector can be interpreted as vector of *class probabilities* because the provided label is the ground truth and encoded as 100% probability for this specific class.

Therefore, the softmax activated (see equation (7)) output layer $\hat{y} = f(x|\theta) = o$ yields the predicted one-hot encoded target variable.
The *cross-entropy* loss between y and $\hat{y} = f(x|\theta) = o$ is defined as

$$L_{\text{CE}}(y, \hat{y}) = -\sum_{j=1}^{n_c} y_j \log(\hat{y}_j), \tag{13}$$

and the empirical risk with cross-entropy loss is computed with

$$\mathcal{L}_{\text{CE}} = \frac{1}{N} \sum_{i=1}^{N} L_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n_c} y_j^{(i)} \log(\hat{y}_j^{(i)}). \tag{14}$$

2.2.3 Regularization

Since the objective in training neural networks is to minimize empirical risk, the value in equation (9) should decrease during training. If the model is good, the value of $\mathcal{L}$ will be small and the model performs bad if the empirical risk $\mathcal{L}$ is comparatively large. If the empirical risk on the training set decreases and the empirical risk for an unseen test set increases, we face the problem of overfitting. The model f_θ has learned the training data *too well* and does not generalize well on unseen test data anymore. Therefore, one naive way in machine learning is to split the entire data set into training and validation set with the ratios $\frac{2}{3}$ and $\frac{1}{3}$ for each set respectively, where the validation set is held out during training. During training of the neural network, the training loss and validation loss can be monitored and used for *early stopping* as a way to avoid overfitting. For the early stopping method, the training of the neural network will be stopped if the validation error increases but the training error still decreases as illustrated in Figure 13.

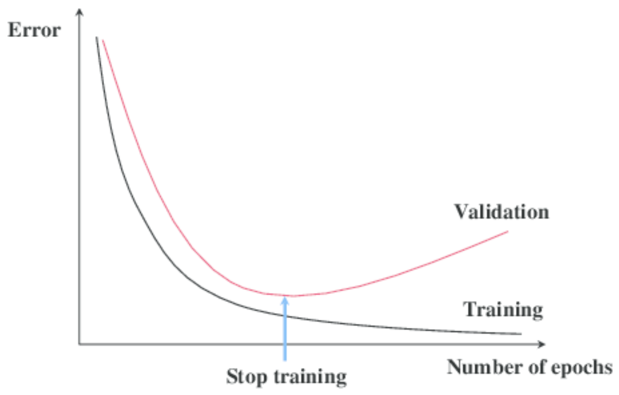


Figure 13: Early stopping is applied when the validation error increases but the training error still decreases. This method is often used as a regularization method when training neural networks.

Other regularization methods for neural networks are the parameter norm penalty $\Omega(\theta)$, e.g. weight decay, or the dropout method. The weight decay (L_2) regularization is similar to ridge regression, where the 2-norm is applied to the learnable parameter θ in order to shrink the components of the parameters and prevent the model from overfitting.

$$\mathcal{L}_{\text{reg.}} = \mathcal{L} + \lambda \Omega(\theta) = \mathcal{L} + \lambda ||\theta||_2^2, \tag{15}$$

where $\lambda > 0$ states the coefficient of the norm penalty and $\mathcal{L}_{\text{reg.}}$ needs to be minimized. Another choice for $\Omega(\theta)$ could be the 1-norm, as done in the lasso regression. The dropout method [Srivastava et al. (2014)] is another simple technique to regularize a deep neural network. The main idea in dropout is to randomly drop hidden units (along with their connections) as shown in Figure 14. By including dropout, the neural network cannot rely on any hidden node too much, since each node has a random probability of being removed. Therefore, the neural network will be cautious to give high weights to certain features, because they might disappear.

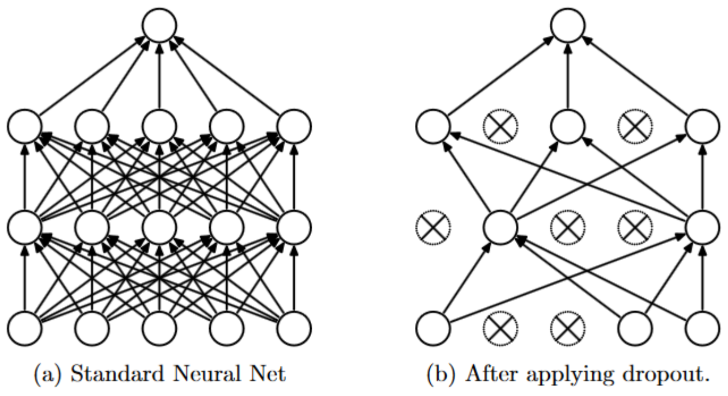


Figure 14: Dropout neural network. **Left (a):** A standard MLP with two hidden layers. **Right (b):** An example of a thinned neural network produced by applying dropout to the network on the left. Crossed units have been dropped. Source: Srivastava et al. (2014)

For model evaluation, when comparing different complex model architectures, sophisticated cross-validation methods [Hastie et al. (2001)] are often applied, where the entire dataset is split into training and test set and the cross-validation is executed on the training set.

2.2.4 Training

Training deep neural networks consists of two parts: *forward-* and *back-propagation*. The forward-propagation consists of computing the predicted output $\hat{y} = f(x|\theta)$ by feeding the input x through the network. Subsequently the loss between the true target y and predicted target $\hat{y}$ is computed. In the backpropagation process, the partial derivatives of the loss with respect to all $\theta^{(i)}$ in each layer are calculated in order to update them, such that in the next forward propagation the loss is smaller than before. The final goal is to minimize the loss function on a training set.

2.2.5 Optimization

Defining a loss function at the output layer of a neural network enables us to measure the performance of the model with respect to its empirical loss in equation (9). The next step is to improve the model by varying the model parameters θ in such a way that the loss decreases. Hence, we turned the machine learning problem of supervised learning into a numerical optimization problem, where we want to minimize the empirical risk. Since deep neural networks mostly model complex data structures and are non-convex, no closed-form solution for minimizing the empirical risk exists. One of the widest used optimization algorithm is the **gradient descent**[7] algorithm. It is a first-order optimization algorithm because it requires the gradient / first derivative of a function, which needs to be minimized.
For updating the model parameter, one has to compute the derivative of the objective with respect to θ and change the parameter in the opposite direction of the gradient, i.e. $-\nabla_\theta \mathcal{R}_{\text{emp}}(f|\theta)$, because we are minimizing the objective function.

2.2.5.1 Gradient Descent

The goal of gradient descent is to minimize a differentiable function in an iterative procedure. The key idea is the following: suppose you are standing on a mountain and want to get to the ground. By iteratively stepping into the direction of steepest descent we will finally arrive at the (local) minimum, which states the ground. The size of step we take in each iteration depends on a learning rate α.
Gradient descent is a first-order order optimization algorithm since it involves the first derivative of an objective function.
In general, gradient descent works in the following way:
assume we have a function $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ that is differentiable and we want to minimize. In this case, $\mathcal{L}$ is the empirical (regularized) risk of a predictive model f parameterized with θ. The optimization problem is stated in equation (9) or (15). The update rule now states to move the model parameters in the direction of steepest descent,

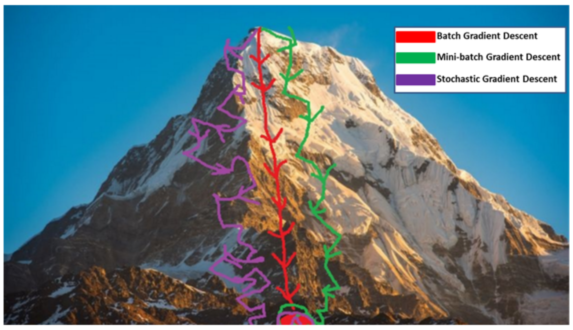
$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}. \tag{16}$$

In general, there are three methods to perform gradient descent [Dabbura (2017)]. *Batch gradient descent* uses the **entire** dataset D in order to perform **one** gradient update. This method can be very memory inefficient and computational expensive when dealing with many samples and complex networks because all instances and results need to be saved in memory when computing the gradient $\nabla_\theta \mathcal{R}_{\text{emp}}(f|\theta)$. Nevertheless, this method approximates the gradient at best and reduces the variances since the gradient is averaged over all N samples.

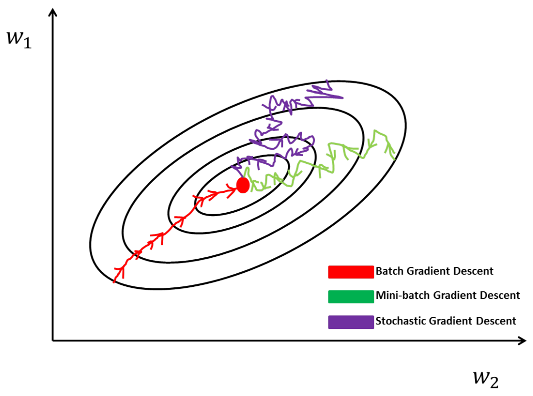
[7]Since we want to minimize the empirical risk we do gradient *descent*. In case we want to maximize a function, gradient *ascent* will be used.

Another method is *Stochastic gradient descent* that allows to update the model parameters, after one random sample $x^{(j)}$ is fed into the model to approximate the gradient for the entire dataset [Bischl (2019a)]. This method includes high variance since the gradient for the entire dataset is approximated with only one example j, that means $\nabla_\theta L(y^{(j)}, f(x^{(j)}|\theta))$. During training of the model, the convergence can be very slow as will be illustrated in the next Figure 15.
As a compromise between the two variants, *mini-batch (stochastic) gradient descent* performs a gradient update after a certain number of random samples have been forwarded in the model. We call this number **batch-size**.
Here, we estimate the gradient $\nabla_\theta \mathcal{R}_{\text{emp}}(f|\theta)$ with the gradient of a randomly small chosen subset of batch-size m:

$$\nabla_\theta \mathcal{R}_{\text{emp}}(f|\theta) = \frac{\sum_{i=1}^{N} \nabla_\theta L(y^{(i)}, f(x^{(i)}|\theta))}{N} \approx \frac{\sum_{i=1}^{m} \nabla_\theta L(y^{(i)}, f(x^{(i)}|\theta))}{m}. \tag{17}$$



(a) Gradient descent to find the (local) minimum of walking down a mountain.



(b) Gradient descent minimizing a function that depends on two parameters $W = (w_1, w_2)^\mathsf{T}$.

Figure 15: Gradient descent variant's trajectories towards reaching the minimum (red point). Each arrow describes one gradient update step. As the batch-size m decreases, the more variance our gradient estimate gets and we will get more 'zig-zag' arrows.

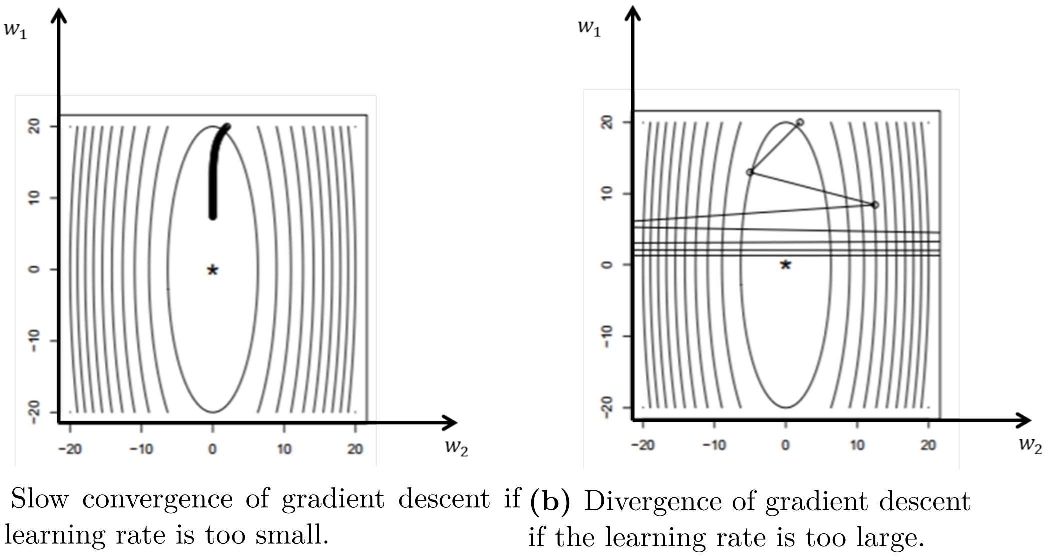
Figure 15b shows a simplified case, where the loss function only depends on two parameters w_1 and w_2. The larger ellipses far from the optimal minimum describe parameter combinations, where the objective (loss) is large and the smaller ellipses, where the loss is small. Here, we have to calculate the derivative of the objective with respect to the parameters w_1 and w_2 via

$$\nabla_W \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2} \right)^\mathsf{T}. \tag{18}$$

Now the parameter update in this example would be as follows

$$(w_1, w_2)^\mathsf{T} \leftarrow (w_1, w_2)^\mathsf{T} - \alpha \nabla_W \mathcal{L}. \tag{19}$$

The learning rate α plays a key-role in the convergence of the algorithm. If the step size is too small, the training process may converge *very* slowly. If the step size is too large, the process may not converge and rather diverge because it jumps around the optimal point. The behaviour of gradient descent with varying learning rates is illustrated in Figure 16.



(a) Slow convergence of gradient descent if the learning rate is too small.

(b) Divergence of gradient descent if the learning rate is too large.

Figure 16: Gradient descent trajectories for small and high learning rate α. The objective function only depends on two parameters $W = (w_1, w_2)^\intercal$. Source: Bischl (2018*b*)

To sum it up, in practice mostly *mini-batch stochastic gradient descent* is used because of the computational efficiency when calculating gradients over a smaller subset of data. Additionally, the stochastic component assists to leave a local minimum, where an exact gradient descent approach might get stuck. For the learning rate α, it is common to decrease it during training, e.g. exponential learning rate decay or specific learning rate schedules [Suki (2017)]. In addition to that, during the years many novel optimization techniques, which are all based on gradient-descent, were developed to accelerate training of deep neural networks and overcome one potential problem such as being stuck in a local optima. For more in-depths over various optimization techniques, Ruder (2016) provides a detailed list of gradient-based variants.

2.2.5.2 Backpropagation

Now that we learned to improve the simple model earlier, which only consists of two parameters $W = (w_1, w_2)^\intercal$, we need to take a further look when dealing with deeper neural networks, which usually consist of many parameters. The update rule in equation (16) stays the same. We just need to think about, how to **efficiently** compute the gradients. As stated in Section 2.2.2.1, deep neural networks consist of many hidden layers (let the number be n_l), where each layer consists of a weight matrix and bias vector. Therefore, our entire network is parameterized with

$$\theta = \{\theta^{(0)}, \theta^{(1)}, ..., \theta^{(n_l)}\} = \{W^{(0)}, b^{(0)}; W^{(1)}, b^{(1)}; ...; W^{(n_l)}, b^{(n_l)}\},$$

where the parameter update rule reads (for each weight matrix and bias vector):

$$W \leftarrow W - \alpha \nabla_W \mathcal{L}, \tag{20}$$

$$b \leftarrow b - \alpha \nabla_b \mathcal{L}. \tag{21}$$

The empirical risk in equation (9) however, does only directly depend on the parameters from the last hidden layer connecting to the output layer, i.e. $(W^{(n_l)}, b^{(n_l)})$. To efficiently compute the gradient of the (batch) cost / empirical risk function with respect to all network parameters, the **backpropopagation** algorithm was proposed by Rumelhart et al. (1986). One of the main ideas in backpropagation is that *(gradient) information* flows from the cost function backwards (on the so called backward pass) through the network. Furthermore, this gradient information describes, how the cost depends on a specific parameter. The backpropagation algorithm exploits the chain-like structure of composing functions in neural networks. Suppose we have a shallow[8] network with three hidden layers, i.e.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))).$$

The model can be explained with stacked operations (matrix multiplication, activation function, ...) and the chain rule of differentiation can be used to compute derivatives of the composition of two or more functions [Bischl (2018*b*)]:

- Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$
  $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$.

- If $y = g(x)$ and $z = f(y)$, the chain rule yields:

$$\frac{dz}{dx_i} = \sum_{j=1}^{n} \frac{dz}{dy_j} \frac{dy_j}{dx_i}, \tag{22}$$

  or in vector notation:

$$\nabla_x z = \left(\frac{dy}{dx}\right)^\top \nabla_y z, \tag{23}$$

  where $\frac{dy}{dx}$ is the $(n \times m)$ jacobian matrix of g.

- In case x and y are one-dimensional, the chain rule is stated as[9]

$$\frac{d}{dx} z = \frac{d}{dx}[f(g(x))] = g'(f(x)) \cdot f'(x).$$

[8]Shallow neural networks contain a small number of hidden layers, mostly up to three.

[9]Derivative of two-composed function, where each component is one-dimensional: 'outer derivative with inner function times inner derivative'.

Computational Graph

Computational graphs are very helpful tools to visualize and understand the chain rule. As mentioned in Section 2.2.2, every neuron consists of two general operations: matrix multiplication and activation. Within a computational graph each node represents a variable, where operations are applied among one or more variables as visualized in Figure 17.

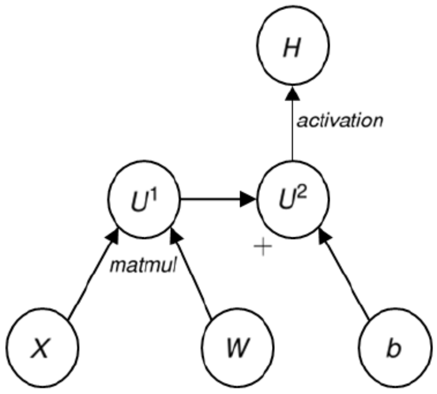
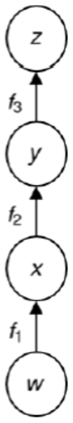
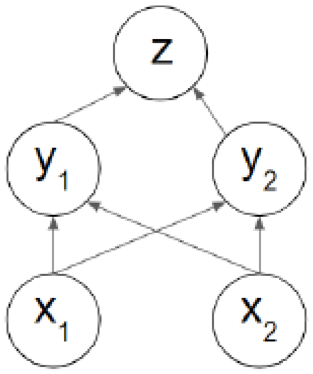


Figure 17: The computational graph for the expression $H = \sigma(XW + b)$. Source: Bischl (2018*b*)

To illustrate the expressive power of computational graphs in combination with the chain rule of calculus, consider the two graphs below in Figure 18.



(a) Computational graph such that $x = f_1(w), y = f_2(x), z = f_3(y)$



(b) Computational graph such that $(y_1, y_2) = f_1(x_1, x_2), z = f_2(y_1, y_2)$

Figure 18: Examples computational graphs. Source: Bischl (2018*b*)

By **iteratively** applying the chain rule from equation (23) to get $\frac{dz}{dw}$ in example 18a results to

$$\begin{aligned}\frac{dz}{dw} &= \frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw} \\ &= f_3'(y)f_2'(x)f_1'(w) \\ &= f_3'(f_2(f_1(w)))f_2'(f_1(w))f_1'(w),\end{aligned}$$

and computing $\nabla_x z$ in example 18b yields to

$$\nabla_x z = \begin{bmatrix} \frac{dz}{dx_1} \\ \frac{dz}{dx_2} \end{bmatrix} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_2}{dx_1} \\ \frac{dy_1}{dy_2} & \frac{dy_2}{dx_2} \end{bmatrix} \begin{bmatrix} \frac{dz}{dy_1} \\ \frac{dz}{dy_2} \end{bmatrix} = \left(\frac{dy}{dx} \right)^\top \nabla_y z.$$

It will be helpful and beneficial when computing partial derivatives (w.r.t. weights and biases) from the cost of the output layer, to save those partial derivatives and when computing the partial derivatives of one previous (hidden) layer, to use those saved ones because they are required for computation due to the chain rule. To elaborate this thought and based on this fundamental rule we can compute the derivatives

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} \text{ and } \frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$$

of the cost function w.r.t. all parameters associated with neurons not directly connected with the output layer.
As mentioned earlier, a common problem in training deep neural networks is the *vanishing gradient problem*, when computing partial derivatives as a product of intermediate partial derivatives (see equation (22)). The vanishing gradient problem appears if activation functions like sigmoid or tanh are used (Figure 11a and 11b) because their derivatives can easily saturate towards zero. Therefore, the choice of activation function and weight initialization is crucial for the success of training deep neural networks. A detailed and illustrative description of the backpropagation algorithm with the update rule are provided by Graves (2008) and Nielsen (2018).

2.2.6 Recurrent Neural Network

Recurrent neural networks (RNNs) are a class of neural networks dealing with sequential data. Sequential data is a stream of (finite) data which is interdependent and has variable lengths. Examples of sequential data are time series data, texts or audio. In a text, a single sentence can have a different meaning than the entire flow of sentences. This lies in the fact how human process information during reading because reading the entire sequence of words is crucial in order to understand the text. The same holds for time series data, e.g. stock market data: a single point means the current price but a full day's sequence of this stock market price shows the movement of this stock and allows to take decision whether to buy or sell.
In contrast to convolutional neural networks and feedforward networks, RNNs are comprising the idea of **memory** by allowing cyclical connections between hidden units. The motivation for RNNs can be inspired by the way how humans read a sentence: one word at a time. So, if we read a sentence from beginning to end, we retain some information about the words that we have already read and use this information to understand the meaning of the entire sentence. Therefore, the classical

RNN cell has the ability to retain some information about past inputs. The success for the use of RNNs are mainly due to the application of **long-short term memory** (LSTM) units [Hochreiter & Schmidhuber (1997)] and **gated recurrent units** (GRUs) [Cho et al. (2014)]. These two variants of RNNs are mostly used nowadays when working with sequential data because they can handle long-term dependencies, i.e. remembering information for long periods. In theory, the classical vanilla RNN (as explained in Section 2.2.6.1) can handle long-term memory as well, but suffers from the vanishing gradient problem due its (simple) definition of recurrent cell by deploying tanh activation function only. Classic RNN is known to have strong short-term memory but weak long-term memory because distant past information has to propagate through many layers to the current position. LSTM and GRU cells however, have more complex definitions of the recurrent cell by adding *gates* in order to forget, update or reset the states and overcome the vanishing gradient problem.

2.2.6.1 Vanilla Recurrent Neural Network

In this Section we will explain the basic workflow for a simple RNN following the example of Graves (2008) using a single, self connected hidden layer, as shown in Figure 19. The recurrent connections in the hidden layer allow a *memory* of previous inputs to persist in the network's internal state, and thereby influence the network output.

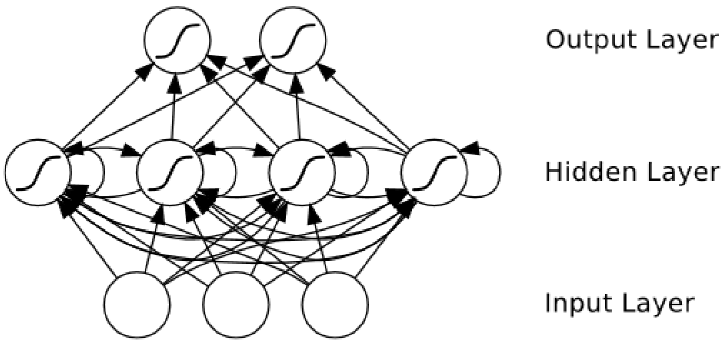


Figure 19: A simple recurrent neural network. The input layer consists of three units and the hidden layer aims to keep track of the history by its recurrent connections. The RNN is fed with one sample $x_t = (x_{(1,t)}, x_{(2,t)}, x_{(3,t)})^\intercal$ at timestep t to predict the outcome $y_t = (y_{(1,t)}, y_{(2,t)})^\intercal$. Source: Graves (2008)

Assume a sequence of vectors $x_{1:t} = (x_1, x_2, ..., x_t)$, where $x_t \in \mathbb{R}^3$ is the input data point at timestep t (e.g. stock market price for three indices A, B, C).
The RNN handles the variable-length sequence $x_{1:t}$ by having a recurrent hidden state, whose activations $h_t \in \mathbb{R}^4$ at each time t is dependent on that of the previous time h_{t-1} and the current input x_t. Hence, the definition of the recurrent hidden

state h_t is formulated as

$$h_t = \begin{cases} 0 & , t = 0 \\ \phi(h_{t-1}, x_t) & , \text{otherwise}, \end{cases} \tag{24}$$

where ϕ is a non-linear activation function such as the composition of a tanh, see equation (4), with an affine transformation as described in Section 2.2.2.1.
For the vanilla RNN, the update of the recurrent hidden state in equation (24) is computed with

$$h_t = \tanh\left(W_{xh} x_t + W_{hh} h_{t-1} + b\right), \tag{25}$$

where $W_{xh} \in \mathbb{R}^{3\times 4}$ denotes the weight matrix for the connection between input x_t and hidden state h_t at timestep t and $b \in \mathbb{R}^4$ the bias vector for the hidden layer. The weight matrix $W_{hh} \in \mathbb{R}^{4\times 4}$ is used for the hidden state vector. This allows memorization of information from previous timesteps.
For generalization and to abstract the calculation of the hidden state h_t, all the operations included for its computation can be formulated in a recurrent cell block A. Viewing an RNN as an unrolled graph makes it easier to generalize to networks with more complex update dependencies (such as LSTMs or GRUs), which are defined within a recurrent cell block A, see Figure 20.

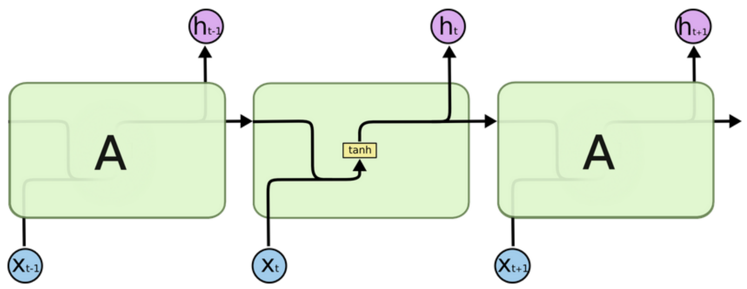


Figure 20: The repeating module in a standard RNN contains a single layer of affine transformation as stated in equation (25). Note that the input x_t and the hidden state activations h_t are vectors. Source: Olah (2015)

The recurrence in Figure 20 is illustrated by passing a sequence of three input vectors. The hidden state h_t is affected by the current input vector x_t and the previous hidden state h_{t-1} as defined in equation (25).
For the cell block A in vanilla RNN, it is worth mentioning that the weights within a RNN are **shared** for each layer for each timestep. To explain this further, note that the weights W_{xh} and W_{hh} in equation (24) are shared over all time steps $t = 1, ..., T$. This has the advantage that the number of parameters to learn in a RNN decreases in contrast to feedforward networks. One drawback for vanilla RNN comes with the vanishing gradient problem when backpropagating the errors in the backpropagation through time (BPPT) algorithm [Graves (2008)].
The vanishing gradient in vanilla RNN is caused due to its simple recurrent cell

block A with tanh activation. This will lead to forget the information from input samples seen in the very beginning of a sequence, as illustrated in Figure 21.

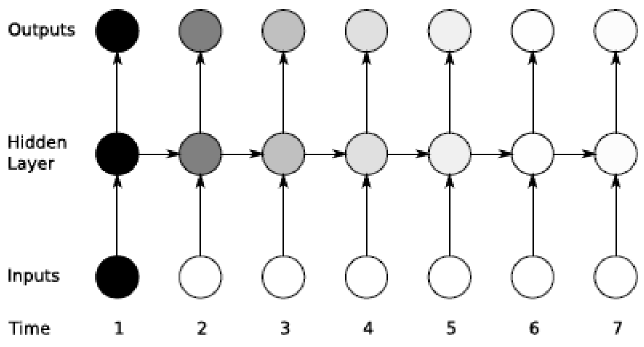


Figure 21: The vanishing gradient problem for (vanilla) RNNs. The shading of the nodes in the unrolled network indicates their sensitivity to the inputs at $t = 1$ (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden (recurrent) layer, and the network 'forgets' the first inputs. Note that each circle is a vector and illustrates a layer. Source: Graves (2008)

Mathematically, the reason for vanishing gradient in vanilla RNN lies in the multiplication of the derivatives of the hidden states at previous timesteps, which due to the tanh activation are restricted in range $(0, 1)$ as illustrated in Figure 11b.

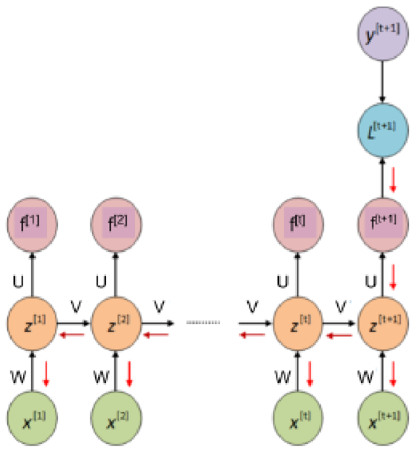


Figure 22: The vanishing gradient in vanilla RNN is caused due to the multiplication of many partial derivatives for different timesteps. Since each partial derivative is bounded within $(0, 1)$, the product of those small partial derivatives will saturate towards zero. Source: Bischl (2018*c*)

The vector z in Figure 22 denotes the hidden state and V the weight matrix for the recurrent hidden layer. The goal in that example is to predict the target at timestep $(t + 1)$ denoted as $f^{(t+1)}$. $L^{(t+1)}$ states the loss for predicting $f^{(t+1)}$ when compared to the true target $y^{(t+1)}$.

For the BPPT algorithm, the partial derivatives w.r.t. the hidden state for all timesteps have to be computed by

$$\frac{dL}{dz^1} = \frac{dL}{dz^{t+1}} \frac{dz^{t+1}}{dz^t} \frac{dz^t}{dz^{t-1}} \cdots \frac{dz^2}{dz^1},$$

which can saturate towards zero since each partial derivative is restricted within (0,1).
Therefore, many modifications on the recurrent cell block A, such as LSTM or GRU cells, have been introduced to overcome problems like vanishing gradient.

The output layer in RNNs also enables a variable-length sequence $y_{1:t} = (y_1, y_2, ..., y_t)$, depending on the model task as illustrated in Figure 23.

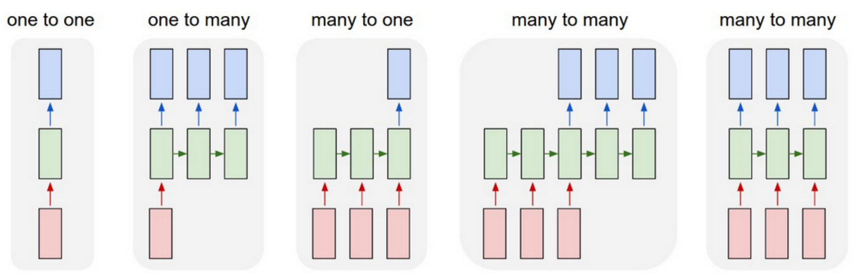


Figure 23: RNNs can be used in tasks that involve multiple inputs and/or multiple outputs. Each rectangle represents a vector and arrows represent functions, such as nonlinear composition of affine transformations. Input vectors are red, output vectors are in blue and green vectors hold the RNN's hidden state. Source: Karpathy (2015)

Different problem settings are:

- Many-To-One: Sentiment analysis, document classification.
- One-To-Many: Image captioning.
- Many-To-Many: Language modeling, machine translation, time-series prediction.

Following the example in Figure 19, the output at timestep t is two-dimensional with $y_t \in \mathbb{R}^2$. For example, y_t could be the predicted stock prices for two other stock indices D, E at time t. Finally, we need to define the operation for the output layer.

$$y_t = \mathrm{act}(W_{hy}h_t + b_y), \tag{26}$$

where $\mathrm{act}(\cdot)$ is an output activation function such as softmax, sigmoid or identity function (in case we want to do regression). $W_{hy} \in \mathbb{R}^{4\times 2}$ represents the weight matrix between the hidden state layer and the output layer and $b_y \in \mathbb{R}^2$ the bias vector for the output layer.

Character-Level Language Models

Knowing the structure of RNNs with the hidden state layer(s) and shared weights over all timesteps, this paragraph shows a small example of a character-level language model as 'many-to-many' model proposed by Karpathy (2015). To train this model, we will input a chunk of text into the RNN and ask it to predict the probability distribution of the next character given a sequence of previous characters.

Suppose we have a defined vocabulary of only four possible letters $\mathcal{V} = \{h, e, l, o\}$ and the goal is to train an RNN on the training sequence 'hello'. Since we have a character-level model, this sequence consists of four different training samples. We conclude that, first the probability of 'e' should be high given the context of 'h', second the probability of 'l' should be high given the context of 'he', third the probability for the character 'l' should also be high given 'hel' and finally 'o' should have a high probability given the context of 'hell'.
By one-hot encoding each one of the four training samples into 4−dimensional one-hot vectors[10], we can feed the RNN the four training samples as a sequence as illustrated in Figure 24 below (many-to-many task).

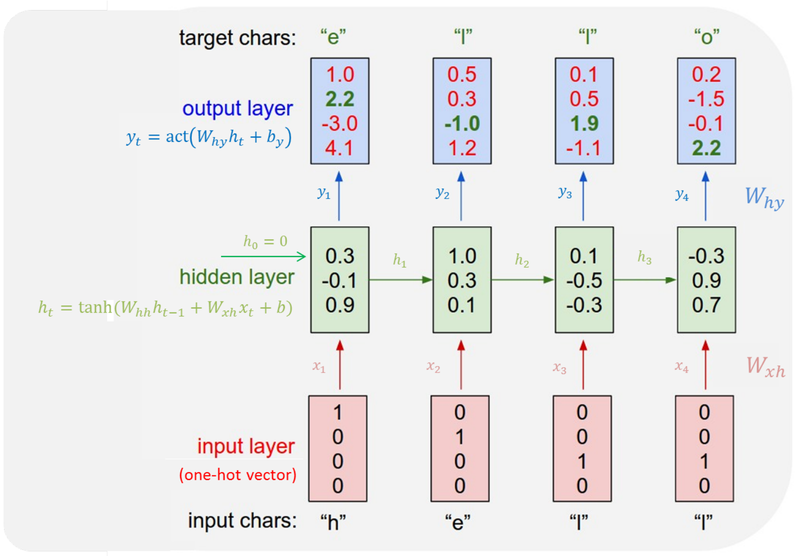


Figure 24: An example RNN with 4−dimensional input and output layers and one hidden layer of three neurons. This diagram shows the activations in the forward pass, when the RNN is fed the characters 'hell' as input. The output layer contains confidence values that are assigned by the RNN for the next character (note the vocabulary is $\mathcal{V} = \{h, e, l, o\}$). The objective in training is that the green numbers in the output layer are high and the red numbers low. No activation is applied on the output layer. Usually softmax activation, see equation (7), is composed on the output layer to obtain probability values for each character of the vocabulary. Here, the act$(\cdot)$ function is the identity function. Source is modified from: Karpathy (2015)

For training, we see that in the first timestep the RNN inputs the character 'h' and assigns a confidence value of 1.0 for the next character to be 'h' again, 2.2 for the next letter to be 'e' and so on. Since the training data string is 'hello', the predicted next character when inputting the sequence 'hell' should be 'o', as indicated by the green value in the output vector at time step four, y_4. Since the RNN consists of differentiable operations, we can run the BPPT algorithm to adjust the weights $\theta = \{W_{hh}, W_{xh}, W_{hy}, b, b_y\}$ such that the green values in the output layer increase. If we were to feed the same sample sequence 'hell' to the RNN again, all the green values in the output layer would be slightly higher because the confidence values increased after one backpropagation update.
It is worth noticing that the first time the character 'l' is input in timestep three

[10]Since the vocabulary consists of only four letters.

$x_3 = (0,0,1,0)^\mathsf{T}$, the target ($y_3$) is ‘l’, but the second time ‘l’ is input in timestep $t = 4$, the target is ‘o’. Hence, the RNN cannot only rely on the input (x_4) but needs the hidden state h_3 from the recurrent connections to keep track of the context and predict the correct next character y_4.

2.2.6.2 Application of RNNs in Drug Discovery

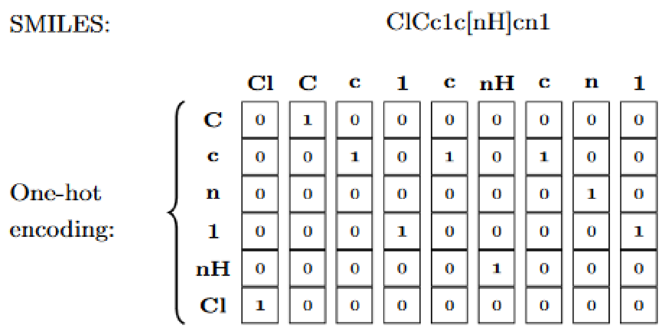
Training language models is one of the common tasks in natural language processing. Recall that an RNN takes a sequence of input vectors $x_{1:t} = (x_1, ..., x_t)$ and an initial hidden state vector $h_0 = 0$ to return a sequence of hidden states for each timestep $h_{1:t} = (h_1, ..., h_t)$ as well as a sequence of output vectors $\hat{y}_{1:t} = (\hat{y}_1, ..., \hat{y}_t)$. Given a sequence of characters $x_{1:t} = (x_1, ..., x_t)$, language models predict the distribution of the next $(t+1)$th character x_{t+1}, i.e. y_t[11]. At each timestep t, the hidden state from the previous timestep h_{t-1}, along with the next character x_t are inputs to the hidden layer to produce a new hidden state h_t, see equation (24), which then affects the final prediction output $\hat{y}_t$ (see Figure 24 and equation (26)). Those computations can be summarized with

$$\begin{aligned} \text{RNN}(h_0, x_{1:t}) &= h_{1:t},\ y_{1:t}\,, && (27)\\ h_t &= A(h_{t-1}, x_t),\ \text{ where } A \text{ is a recurrent cell, see Figure 20}, && (28)\\ \hat{y}_t &= O(h_t), && (29) \end{aligned}$$

where $O(\cdot)$ is a composition of affine linear transformations and activations, e.g. one fully-connected layer as defined in equation (26).

H. S. Segler et al. (2017) trained a character-based language model using the SMILES representation of compounds and its vocabulary as described in Section 2.1.2. For example, if the model receives the sequence `c1ccccc`, there is a high probability that the next symbol will be `1`, which closes the ring, and yields benzene. Assume the SMILES characters are contained in a vocabulary $\mathcal{V}$ and $|\mathcal{V}| = n_v$ is the number of characters from this vocabulary. The input characters are encoded as one-hot vectors [Graves (2013)] regarding the SMILES vocabulary similar to equation (12) and exemplified in Figure 25.
Recall that the SMILES vocabulary contains symbols of atoms and notations for bonds, branches and ring openings and closings.

[11] If we split the training data set into feature set and label set. Since language models are a supervised learning task, given a sample x we want to predict the label y. Here: given a sequence of characters $x_{1:t} = (x_1, .., x_t)$, we want to predict the next character $y_t = x_{t+1}$.



SMILES: ClCc1c[nH]cn1

One-hot encoding:

	Cl	C	c	1	c	nH	c	n	1
C	0	1	0	0	0	0	0	0	0
c	0	0	1	0	1	0	1	0	0
n	0	0	0	0	0	0	0	1	0
1	0	0	0	1	0	0	0	0	1
nH	0	0	0	0	0	1	0	0	0
Cl	1	0	0	0	0	0	0	0	0

Figure 25: Depiction of an one-hot encoded representation derived from the SMILES of a molecule. Here a reduced vocabulary is shown, while in practice the vocabulary is much larger, covering all tokens (unique characters) present in the training data. Source: Olivecrona et al. (2017)

The RNN language model needs to be able to deal with long-term dependencies because it has to learn the SMILES grammar with all the defined rules in Section 2.1.2. Especially, the RNN needs to learn the SMILES grammar in terms for ring openings and closings as illustrated in Figure 26.

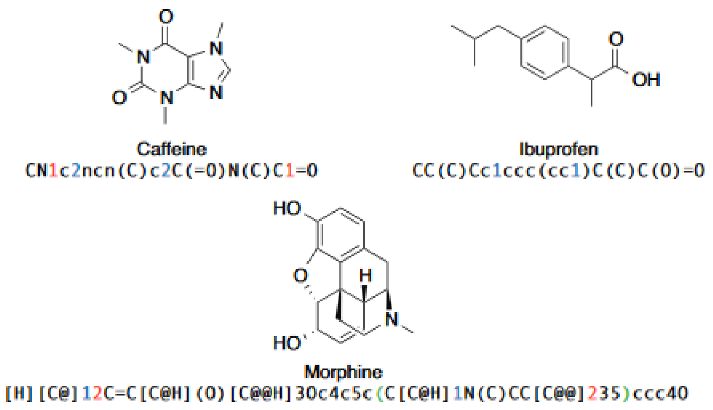

Caffeine
CN1c2ncn(C)c2C(=O)N(C)C1=O
Ibuprofen
CC(C)Cc1ccc(cc1)C(C)C(O)=O
Morphine
[H][C@]12C=C[C@H](O)[C@@H]3Oc4c5c(C[C@H]1N(C)CC[C@@]235)ccc4O

Figure 26: Examples of molecules and their `SMILES` representation. To correctly create SMILES, the model has to learn long-term dependencies, for example to close rings (indicated by the colored numbers) and brackets. Source: H. S. Segler et al. (2017)

More formally, given a sequence of one-hot encoded SMILES characters
$x_{1:t} = (x_1, ..., x_t)$ the probability given by the network, parameterized by a set of weights combined in θ, to the input sequence $x_{1:t}$ is decomposed as

$$p_\theta(x) = p_\theta(x_1, ..., x_t) = \prod_{j=1}^{t} p_\theta(x_j | x_{j-1}, ..., x_1), \tag{30}$$

where we want to maximize this probability by fitting the model to a dataset D. Since the RNN contains an output layer for modeling the class probabilities of the next character, $\hat{y} \in (0,1)^{n_v} \subset \mathbb{R}^{n_v}$, RNNs usually use cross-entropy as loss function, since language modeling is a supervised task as exhibited in Figure 24.
Given a timestep t, we have the one-hot encoded character x_t and want to predict the next character $\hat{y}_t$, where the next character label y_t is given and the RNN model

keeps track of the history by having the hidden state h_t. The probability distribution of $p_\theta(\hat{y}_t|x_t, ..., x_1) = \text{RNN}_\theta(x_t|h_{t-1}) \overset{(29)}{=} O(h_t)$ is computed using softmax activation function (see equation (7)) to create probability values for each character in the SMILES vocabulary. As each timestep t the RNN model outputs (class) probabilities $\hat{y}_t$ for each character of the vocabulary, it is straightforward to use cross-entropy loss (see equation (13)) as a loss function. Categorical cross-entropy implies that we try to optimize the logarithmized probability of the correct character. As an example we will denote this dependency as $\hat{y}_t = \text{RNN}_\theta(x_t|h_{t-1}) = \text{softmax}(W_{hy}h_t + b_y) \in (0,1)^{n_v}$. Hence, the cross-entropy loss at timestep t is defined with

$$L_{\text{CE}}(y_t, \text{RNN}_\theta(x_t|h_{t-1})) = -\sum_{j=1}^{n_v} y_{t,j} \log(\hat{y}_{t,j}). \tag{31}$$

Suppose we obtained a pre-processed dataset D[12] for language modeling with N one-hot encoded sequences $\{x^{(i)}\}_{i=1}^N$ and its corresponding one-hot encoded next-character labels $\{y^{(i)}\}_{i=1}^N$. Empirical risk minimization as done in equation (9) over a dataset leads to

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{n_v} y_j^{(i)} \log(\hat{y}_j^{(i)}) \to \min_\theta, \tag{32}$$

where the partial derivatives of the empirical risk $\mathcal{L}$ w.r.t. network weights can be efficiently calculated with backpropagation through time algorithm [Graves (2008)]. Once the model is trained, the sampling procedure can be described with Figure 27.

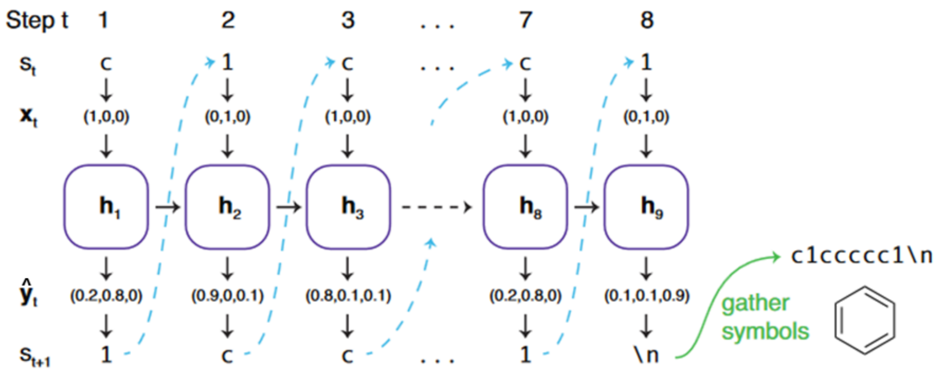
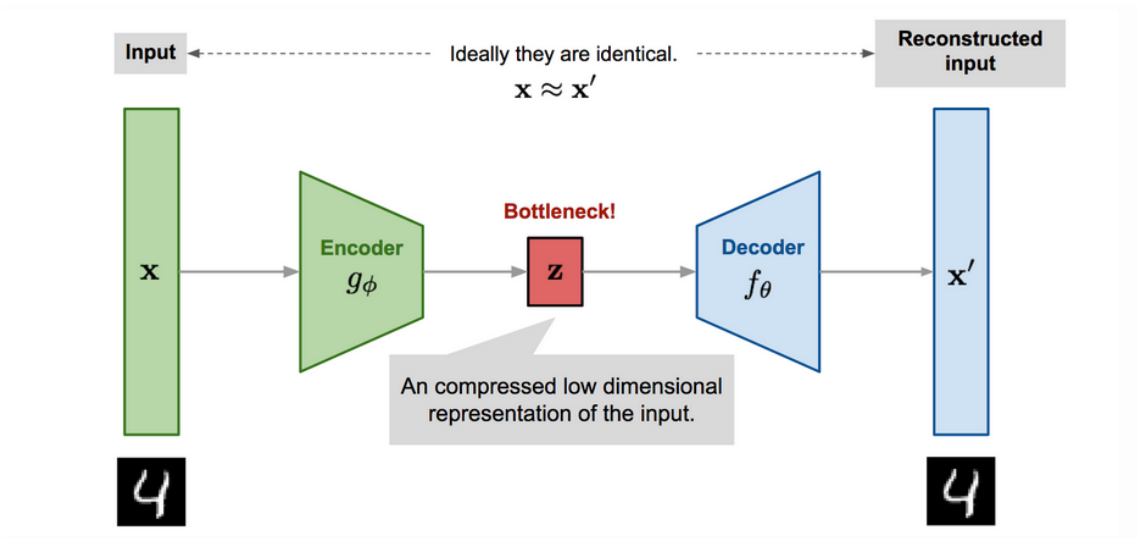


Figure 27: Sampling process. Start with a random seed symbol s_1, here `c`, which gets converted into a one-hot vector x_1 and input into the model. The model updates its internal state h_0 and h_1 and outputs $\hat{y}_1$, which is the probability distribution over the next symbols. Here, sampling yields $s_2 = 1$. Converting s_2 to x_2, and feeding it to the model leads to updated hidden state h_2 and output y_2, from which can be sampled again. This iterative symbol-to-symbol procedure is continued until the end-of-line token `\n` is sampled. Here the result yields the benzene ring. The hidden state enables to keep track of opened brackets and rings, to ensure that they will be closed again. Here the SMILES vocabulary $\mathcal{V} = \{$`c,1,\n`$\}$ is used. Source: H. S. Segler et al. (2017)

[12]Note that when implementing, the sequences need to be split by timestep. For example, if we have the word 'hello' and set the split for $t = 1$, the corresponding features and labels (x, y) will be (h,e), (e,l), (l,l), (l,o) which furthermore must be one-hot-encoded. Therefore, RNNs get three-dimensional tensors as input with the shape=(batch-size, time-step, n_v).

2.3 Autoencoders

An autoencoder falls under the category of **unsupervised learning**, where the objective is to learn a **representation** of features x, e.g. through manifold learning or dimensionality reduction. It is called unsupervised because in contrast to supervised learning, no corresponding target variable y is needed.
The basic idea of an autoencoder is to obtain a model that is able to reconstruct its input, where the autoencoder consists of two neural networks, an encoder network and decoder network, as illustrated in Figure 28 below.



Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 28: Illustration of the autoencoder model architecture on the MNIST dataset. The MNIST dataset is a large set of handwritten digits that is commonly used for training various image processing systems. Source: Weng (2018)

The two neural networks have the following tasks:

- **Encoder network** g_ϕ: It encodes the original high-dimensional input x into a latent low-dimensional code z. The input size is usually larger than the output (code) size, formally $z = g_\phi(x)$.

- **Decoder network** f_θ: It reconstructs the original feature from the compressed code, formally $x' = f_\theta(z) = f_\theta(g_\phi(x)) \approx x$.

The case above is an *undercomplete autoencoder* because the autoencoder would be of no use, if it simply learns the identity $f_\theta(g_\phi((x)) = x$. In fact, we want the autoencoder to learn **useful** and **significant** properties of the features (by compressing them into a continuous bottleneck code). Also, classical principal component analysis (PCA) can be viewed as an autoencoder, if the encoder and decoder networks are just applying linear transformations and the reconstruction loss is the L_2 loss. For a proof of the statement earlier please refer to Khapra (2019).

The parameters (ϕ, θ) are learned together end-to-end by minimizing the reconstruction error with L_2 loss as stated in equation (11), leading to the empirical risk

$$\mathcal{L}_{\text{AE}} = \frac{1}{N}\sum_{i=1}^{N} L_2(x^{(i)}, f_\theta(g_\phi(x^{(i)}))) = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - f_\theta(g_\phi(x^{(i)})))^2, \tag{33}$$

where as optimization algorithm stochastic batch gradient descent and backpropagation (see Section 2.2.5.2) for updating the encoder and decoders network weights can be applied to minimize this empirical risk.

Once the autoencoder is trained, the latent code is often used for various downstream tasks such as supervised learning by taking it as input feature for a predictive model. In case of a clustering task, the clustering can also be applied in the latent space. Therein the main idea is that by compressing the feature representation into the latent code, samples might be *disentangled* and lie within different groups in the encoded latent space.
Since many variants of autoencoder exist and are a field of active research, a summary of the fundamental variants and extensions is provided by Weng (2018) and Goodfellow et al. (2016).

2.3.1 Translation Model to Learn Molecular Descriptors

The objective by Winter et al. (2018) is to learn informative molecular descriptors (see Section 2.1) from low-level molecular encodings such as SMILES or InCHI. In contrast to the basic idea of autoencoders, where the autoencoder has the purpose to reconstruct its input, Winter et al. (2018) borrow ideas from neural machine translation [**Seq2Seq** model by Sutskever et al. (2014) to translate between English and French text.]: it translates between two semantically equivalent but syntactically different representations of molecular structures, compressing the meaningful information in both representations in a low-dimensional representation code vector, called `cddd` (Continuous Data-Driven Descriptor).
For example, one possible translation model would receive as input an InCHI representation of a compound, encode it into the latent space, which is the desired molecular descriptor, and then decode that molecular descriptor to the canonical SMILES representation of the respective compound as displayed in Figure 29.

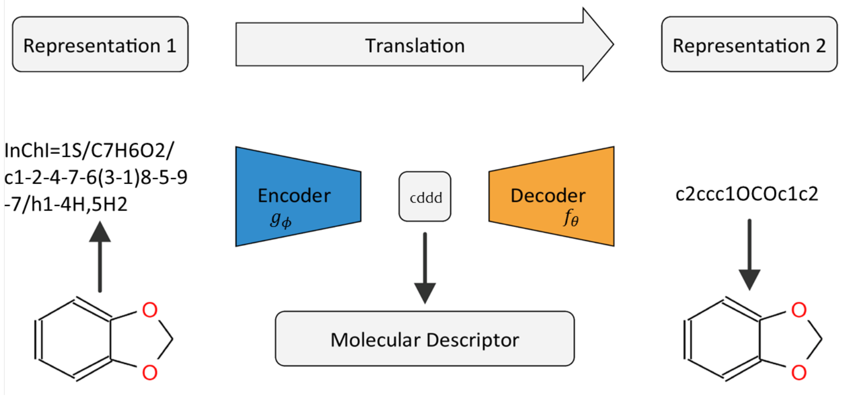


Figure 29: General architecture of a translation model using the example of translating between the InCHI and SMILES representation of 1,3-Benzodioxole. Source: Winter et al. (2018)

The translation model was trained on a large dataset of approximately 72 million compounds. Since the translation model works with sequential data, tokenization of sequences into one-hot vector representations was done as illustrated earlier in Figure 25. By defining a lookup-table for both, SMILES and InCHI vocabulary, the SMILES vocabulary consists of 38 unique characters and the InCHI vocabulary of 28 unique characters. The translation model itself comprises two neural networks as shown in Figure 29. For implementation details and network architectures please refer to the supplementary information (SI) from Winter et al. (2018).
Once the translation model is trained, the molecular descriptor can be extracted for any compound and utilized as molecular descriptor for several downstream tasks, such as predictive modeling in quantitative structure-activity relationships (QSAR) tasks. Since the goal was to learn good feature representations that could be used for further downstream tasks, Winter et al. extended the translation model with an additional predictive model forecasting nine continuous molecular properties, $a \in \mathbb{R}^9$, which contributes into the overall loss function. By including this additional model,

the translation model is forced to learn **meaningful** continuous representations. The predictive (regression) model is a three-layer fully connected neural network d_η that takes as input the molecular descriptor `cddd` and outputs a molecular property vector of dimension nine. It is trained simultaneously with the translation model. The encoder g_ϕ and decoder network f_θ are both RNNs as illustrated in Figure 30.

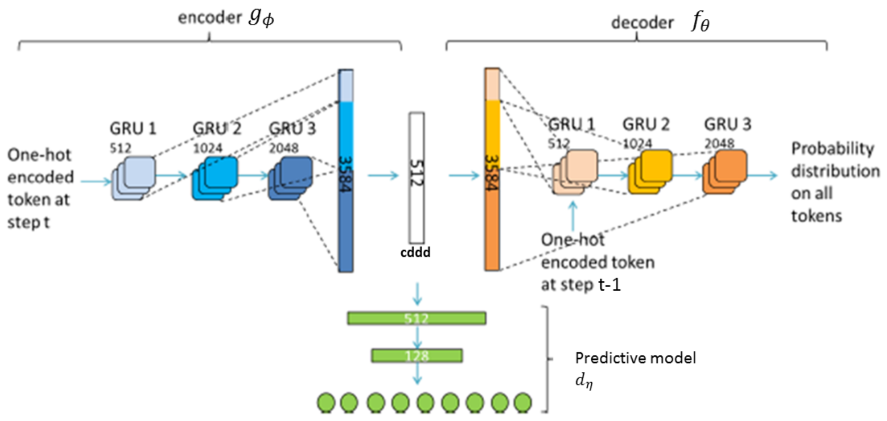

encoder g_ϕ
decoder f_θ
One-hot encoded token at step t
GRU 1
512
GRU 2
1024
GRU 3
2048
3584
512
cddd
3584
GRU 1
512
GRU 2
1024
GRU 3
2048
Probability distribution on all tokens
One-hot encoded token at step t-1
512
128
Predictive model d_η

Figure 30: The final model architecture comprises the translation model, where the encoder and decoder network each use three-stacked GRU layers [Cho et al. (2014)] with sizes 512, 1024, 2048. Additionally, the prediction network is included. Source is modified from SI: Winter et al. (2018)

To explain the translation process further, the encoder RNN g_ϕ takes as input the one-hot encoded token/character at timestep t, computes the hidden state for each of the three GRU layers and maps the concatenated (hidden) cell states from the three GRU layers (colored blue in Figure 30) of g_ϕ to one fully-connected layer, which then outputs the molecular descriptor as 512−dimensional vector activated with tanh function. The decoder RNN f_θ takes as input the latent `cddd`-representation and maps it into one fully-connected layer of a size $512 + 1024 + 2048 = 3548$, where the activated neurons of this layer are used to initialize the hidden states for the three recurrent GRU layers of the decoder (colored orange in Figure 30). Since the translation model is a **Seq2Seq-autoencoder**, the decoder network predicts the class probability for each character of the SMILES vocabulary at timestep t, as the input for the encoder was also a token at timestep t. Hence, the decoder network needs as input the one-hot encoded token at timestep $(t-1)$ *and* the initialized hidden states from the processed `cddd`-embedding in order to predict the character at timestep t. The hidden state from the last GRU layer is mapped to an output layer to predict probabilities for the different tokens via one fully-connected layer with softmax activation function similar to the model by H. S. Segler et al. (2017) explained in Section 2.2.6.2. The complete translation model is trained on minimizing cross-entropy between this probability distribution and the one-hot transformed correct characters in the target sequences, stating the translation loss $\mathcal{L}_{\phi,\theta}$ between encoder and decoder, *as well as* minimizing the mean-squared error for the property prediction $\mathcal{L}_{\phi,\eta}$, via the prediction network d_η.

The total empirical risk containing cross-entropy and L_2 loss is defined with

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[\sum_{j}^{n_v} y_j^{(i)} \log(\hat{y}_j^{(i)})\right] + \frac{1}{N}\sum_{i}^{N}\left(a^{(i)} - d_\eta(g_\phi(x^{(i)}))\right)^2, \text{ where } \hat{y}^{(i)} = f_\theta(g_\phi(x^{(i)})) \tag{34}$$

and $a^{(i)}$ is the i-th molecular property vector of a compound. Recall that this function is minimized w.r.t. ϕ, θ and η. Since d_η inputs a `cddd`, when backpropagating errors, useful gradient information can be passed to the encoder network g_ϕ, adjusting its parameters to create better molecular descriptors. This enforces that the translation model, besides performing well in translation, is also well suited to extract meaningful molecular descriptors from the input sequence x.
The overall best translation model **Sml2canSml**, also considering the predictive modeling objective, is achieved when translating SMILES to their canonical form.

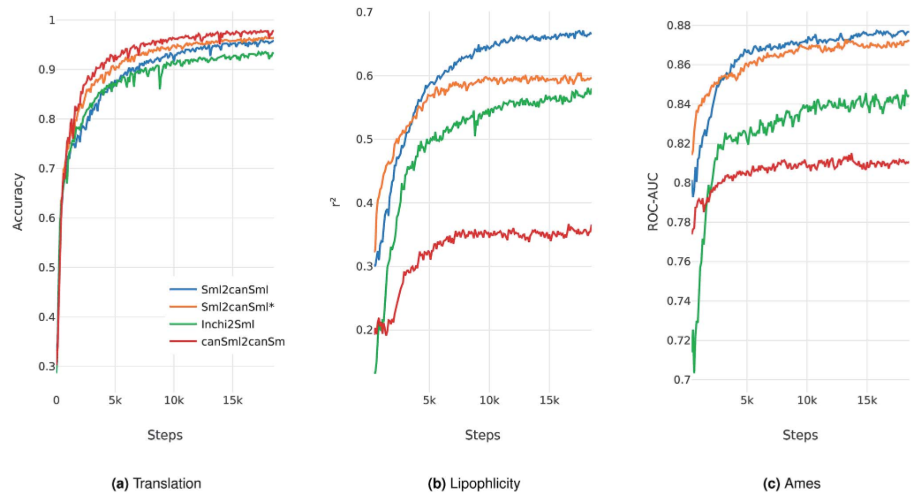


Figure 31: Performance of the best model on four different translation tasks during the first 20000 training steps. The Sml2canSml* run was trained without the additional predictive model d_η. (a) Translation accuracy, (b) Mean performance on the lipophicity regression task, (c) Mean performance on the Ames (bioactivity) classification task. For (b) and (c), the translation model at the respective step was utilized to extract the molecular descriptor `cddd` and fed into a SVM to model both tasks (on a QSAR validation set). Source: Winter et al. (2018)

Figure 31 shows the comparison for different translation tasks (from which sequence type to translate from and to) regarding the translation accuracy and secondly displays the predictive performance of the molecular descriptor on two additional validation tasks. Regarding the translation accuracy, if no predictive modeling validation task is considered, the translation from SMILES to canonical SMILES as well as InCHI to SMILES performs good and the pure autoencoding task from canonical SMILES to canonical SMILES performs best. However, when looking at the validation tasks in Figure 31a and 31b, the pure autoenconding task leads to molecular descriptors that are not well suited for the two predictive modeling tasks.
This strengthens the initial idea that the translation between two syntactically different sequences enforced the translation model to capture the 'true' molecular essence that both input and output sequences have in common.

2.4 Generative Adversarial Networks

The generative adversarial network (GAN) [Goodfellow et al. (2014)] is an unsupervised learning method that aims to estimate a probability distribution of the features of a real dataset D, i.e. learn p_r. Classical traditional approaches accomplish this by learning a parametric family of densities $\{p_\theta\}_{\theta \in \mathbb{R}^d}$ and finding the parameter θ^* that maximizes the likelihood on the real data. Suppose the dataset contains N i.i.d samples, leading to the dataset $D = \{x^{(i)}\}_{i=1}^N$.
The maximum likelihood problem can be formulated as

$$\begin{aligned} \theta^* &= \arg\max_\theta \prod_{i=1}^N p_\theta(x^{(i)}) \\ &= \arg\max_\theta \log \prod_{i=1}^N p_\theta(x^{(i)}) \\ &= \arg\max_\theta \sum_{i=1}^N \log(p_\theta(x^{(i)})) \\ &= \arg\max_\theta \frac{1}{N} \sum_{i=1}^N \log(p_\theta(x^{(i)})). \end{aligned} \tag{35}$$

The maximum likelihood optimization is easiest achieved in logarithm space since it simplifies the objective function and does not change the optimum θ^*, as will be explained in the following. When working with the logarithm, the objective function simplifies from a product to a sum, where the derivatives of the log likelihood are numerical less prone to arithmetic underflow by multiplying several small probabilities. Multiplying with a constant such as $\frac{1}{N}$ will not change the optima either but will prove to be useful when analyzing behavior as the number of data points gets infinitely large. In the limit as $N \to \infty$ the maximum likelihood estimation is equivalent to minimizing the Kullback-Leibler divergence as shown in Appendix B.

In the GAN setting, instead of trying to *directly* estimate p_r through a parametric family $\{p_\theta\}_{\theta \in \mathbb{R}^d}$, one defines a random variable Z[13] and pass it through a parametric function $g_\theta : \mathcal{Z} \to \mathcal{X}$, such as a deep neural network, that directly generates observations following a certain distribution p_θ. By changing the parameter θ, one can change this distribution p_θ and push it closer to the real data distribution p_r, obtaining following approximation

$$\tilde{x} = g(z|\theta) \overset{p_r}{\approx} x.$$

Note that this formulation is mathematically not correct and '$\approx$' states that the generated sample $\tilde{x}$ is approximately equal to x, where x is a true sample and that

[13]The random variable Z usually has the property that it is computational cheap to sample from and follows a fixed distribution $p(z)$ on its domain $\mathcal{Z}$.

both derive from the same probability distribution p_r.
In general, the GAN framework consists of two competing instances, namely the generator G and the discriminator D. Typically those two instances are modeled using the class of neural networks.
GANs have recently gained substantial popularity and have found numerous recent applications, such as video generation [Vondrick et al. (2016); Xiong et al. (2017)], image synthesis, audio generation [Donahue et al. (2018)] and usage in bioinformatics [S & Thilak Chaminda (2017)] as well as cheminformatics [Schwalbe-Koda & Gómez-Bombarelli (2019); Elton et al. (2019)]. For that reason, GANs are an active and vivid field of research leading to publications of many new GAN variants.
This work includes the description and explanation of the classical **vanilla GAN** and two improved variants the **Wasserstein GAN** and **Wasserstein GAN-GP**.

2.4.1 Divergence Metrics

Since we are trying to approximate a real probability density function indirectly through a parametric function modeled as neural network, it is necessary to measure how close the model distribution p_θ and real distribution p_r are, or correspondingly, define a distance or divergence $\varphi(p_r, p_\theta)$. The notion of (statistical) distance between probability measures has found many applications in probability theory, mathematical statistics and information theory [Sriperumbudur et al. (2012)] but can be misleading sometimes, since statistical distance measures are mostly not metrics and do not need to be symmetric as can be seen in the next Section. This work will describe variants coming from two popular families of distances/divergences between probability measures, namely the **f-divergence/ϕ-divergence** and **integral probability metric (IPM)**. For a detailed definition of those general distance metrics and their advantages as well as disadvantages, it is worth reading the manuscript of Sriperumbudur et al. (2012). In the following we assume p_r and p_θ to be continuous probability densities on $\mathcal{X} = \mathbb{R}^p$. The logarithm operation throughout this work is stated as the natural logarithm to the base e.

2.4.1.1 Kullback-Leibler Divergence

The Kullback-Leibler (KL) divergence is a ϕ-divergence and has its origin in information theory. It is a non-symmetric measure of difference between two probability distributions $p_r(x)$ and $p_\theta(x)$ and states the information lost, when $p_\theta(x)$ is used to approximate $p_r(x)$. Formally, the KL divergence is defined as

$$\varphi_{KL}(p_r || p_\theta) = \int_x p_r(x) \log \frac{p_r(x)}{p_\theta(x)} dx. \tag{36}$$

Although the KL divergence measures the 'distance' between two probability distributions, it is not a distance in a sense as metric. The reason lies in the fact that the

KL divergence is not symmetric, i.e. $\varphi_{KL}(p_r||p_\theta) \neq \varphi_{KL}(p_\theta||p_r)$ and does not satisfy the triangle inequality. The KL-divergence holds the two following properties

1. $\varphi_{KL}(p_r||p_\theta) \geq 0$.
2. $\varphi_{KL}(p_r||p_\theta) = 0$ if and only if $p_r(\cdot) = p_\theta(\cdot)$.

These two properties can be summarized as positive definiteness.
For φ to be a metric on $\mathcal{X}$, the properties of positive definiteness, symmetry and triangle inequality have to be satisfied by φ [Makarychev (2015)].
As mentioned earlier in equation (35), the maximum likelihood estimation is equivalent to minimizing the KL divergence between the real data distribution and the model distribution. One drawback of the KL divergence is that it can get infinitely large if $p_r(x) > 0$ and $p_\theta(x) \to 0$. This is especially the case if the support of p_θ lies on a low-dimensional manifold (in initial training).
Recall that the support of a (probability density) function $p : \mathcal{X} \to (0,1)$ is the set of points in $\mathcal{X} = \mathbb{R}^n$, where p is non-zero. Hence, that set $\mathcal{A} = \text{supp}(p)$ is characterized, where $p(x) > 0$ holds $\forall x \in \mathcal{A}$.
Arjovsky & Bottou (2017) claim that it is very unlikely in the beginning of training, that all the support of p_r lies within the (low dimensional) support of p_θ. As a conclusion, if even a single real data point x lies outside of the support of p_θ, the KL divergence will explode.

2.4.1.2 Jensen-Shannon Divergence

The Jensen-Shannon (JS) divergence was firstly introduced by Lin (1991) and is a symmetrized and smoothed version of the KL divergence. The JS divergence holds the two properties mentioned earlier in the KL divergence Section and is symmetric and bounded and therefore has always finite values. In addition to that, the square root of the JS divergence yields a metric, satisfying the triangular inequality and the other properties mentioned earlier [Nielsen (2010)].
Mathematically, the JS divergence has the following form

$$\begin{aligned}
\varphi_{JS}(p_r||p_\theta) &= \frac{1}{2}\varphi_{KL}(p_r||\frac{1}{2}(p_r+p_\theta)) + \frac{1}{2}\varphi_{KL}(p_\theta||\frac{1}{2}(p_r+p_\theta)) && (37)\\
&= \frac{1}{2}\varphi_{KL}(p_r||M) + \frac{1}{2}\varphi_{KL}(p_\theta||M) && (38)\\
&= \frac{1}{2}\int_x \left[p_r(x)\log\frac{p_r(x)}{0.5(p_r(x)+p_\theta(x))} + p_\theta(x)\log\frac{p_\theta(x)}{0.5(p_r(x)+p_\theta(x))}\right]dx && (39)\\
&= \frac{1}{2}\int_x \left[p_r(x)\log\frac{2p_r(x)}{p_r(x)+p_\theta(x)} + p_\theta(x)\log\frac{2p_\theta(x)}{p_r(x)+p_\theta(x)}\right]dx, && (40)
\end{aligned}$$

where $M = \frac{1}{2}(p_r + p_\theta)$ can be interpreted as the mixture (average) distribution of the real data distribution p_r and approximated data distribution p_θ.

2.4.1.3 Wasserstein-1 Distance

The Wasserstein-1 distance is also called Earth's Mover distance (for discrete random variables) because it describes the minimum amount of 'work' required to transform (probability) mass or earth/dirt from one distribution to another, e.g. p_θ to p_r. It has a connection to optimal transport theory [Villani (2008)]. One can understand a probability distribution by how much mass $m \in [0, 1]$ it assigns to a point x. Suppose we begin with distribution p_θ and want to move mass around to change that distribution into p_r. Moving that probability mass m by distance d leads to the cost of $m \cdot d$. As an example [Hui (2018)] we can think of a transport plan γ moving boxes as illustrated in Figure 32. The distance to be considered is only on the horizontal axis, as in an univariate distribution. We obtain six boxes and want to move them from the left side to the right location as seen below (note that there is a desired shape for the discrete probability distribution on the right side).

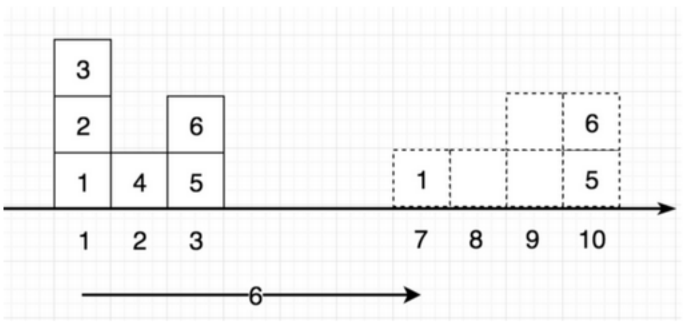
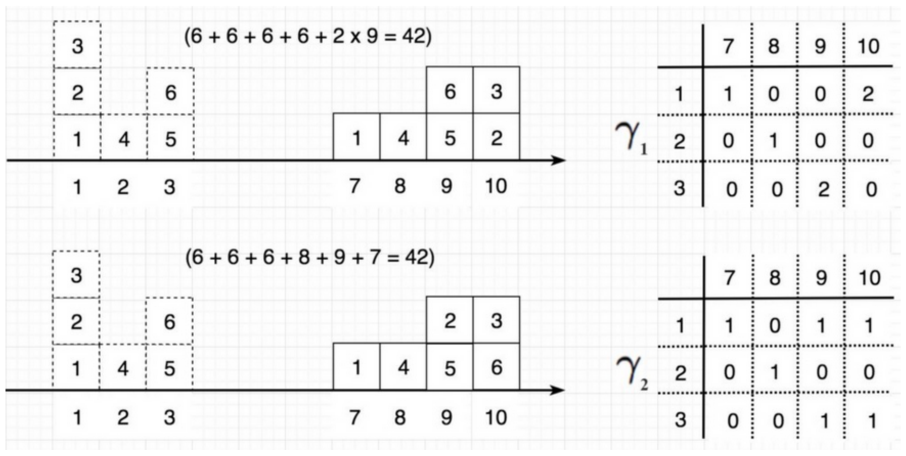


Figure 32: The goal is to move the boxes from the left side to the right side. Source: Hui (2018)

The distance for moving box 1 from location 1 to location 7 equals to $d(1, 7) = ||7-1||_1 = ||6||_1 = 6$. Suppose, we have two different transport plans γ_1 and γ_2 that tell us how many boxes from the left location to the right location were moved in the Figure 33 below. The total transport costs in both cases are the same but the transport plans are different.



γ_1

	7	8	9	10
1	1	0	0	2
2	0	1	0	0
3	0	0	2	0

γ_2

	7	8	9	10
1	1	0	1	1
2	0	1	0	0
3	0	0	1	1

Figure 33: Different transport plans can lead to the same total cost. Source: Hui (2018)

However, not all transport plans carry the same cost. Since the objective is to match the desired distribution on the right hand side p_r, many possible ways of moving the boxes are possible.
Now coming back to continuous random variables: the Wasserstein-1 distance is the cost of the cheapest transport plan γ^*. The transport plan $\gamma(x, y)$ describes, how the amount of probability mass is distributed from one point $x \in \mathcal{X}$ to another point $y \in \mathcal{Y}$, so as to make p_θ follow p_r. Usually we assume, $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^n$ describing the compact domain space for each probability density p_θ and p_r, which in most cases is the same domain space. To be a valid transport plan, two marginalization constraints must also be satisfied

$$\int_x \gamma(x, y) dx = p_r(y) \text{ and } \int_y \gamma(x, y) dy = p_\theta(x). \tag{41}$$

Those two constraints ensure that following this plan yields the correct distributions, e.g. once we finish moving the planned amount of mass from every possible x to the target y, we end up with exactly the distribution according to the desired probability distribution p_r. In this case, the transport plan can be seen as joint distribution of the starting distribution p_θ and the target distribution p_r.
As there are infinitely many sets of transport plans, which satisfy the marginalization constraints, we define $\prod(p_r, p_\theta)$ to be the set of all possible joint probability distributions between p_r and p_θ.
When treating x as starting point and y as destination point, the total amount of probability mass moved is $\gamma(x, y)$ and the moved distance equals to $||x - y||_1$. Note that when mentioning Wasserstein distance, precisely Wasserstein-1 distance is considered. In this case, we define the cost of movement as the l_1 distance between two points, hence $||x - y||_1$. For that reason, the cost of movement between two points x and y with a specific amount of mass $\gamma(x, y)$ equals to $\gamma(x, y) \cdot ||x - y||_1$. For a definition of the Wasserstein-p distance please refer to Appendix A.2.
In view of computing the total cost of a transport plan γ and taking into account that we deal with a joint probability distribution $\gamma(x, y)$, it is straightforward to minimize the expected cost with respect to the l_1 distance.
The expected cost for one valid transport plan averaged across all (x, y) pairs can be computed with

$$\int_x \int_y \gamma(x, y) ||x - y||_1 dy dx = \mathop{\mathbb{E}}_{(x,y)\sim\gamma} \left[||x - y||_1 \right]. \tag{42}$$

Computing the infimum over all valid transport plans γ of the expected cost, leads to the Wasserstein-1 distance

$$\varphi_W(p_r, p_\theta) = \inf_{\gamma \in \prod(p_r, p_\theta)} \mathop{\mathbb{E}}_{(x,y)\sim\gamma} [||x - y||_1]. \tag{43}$$

2.4.2 Vanilla GAN

The classical vanilla GAN [Goodfellow et al. (2014)] consists of two deep neural networks, namely the generator G (parameterized with θ_g that approximates the real data distribution p_r) and the discriminator D (parameterized with θ_d that estimates the probability that a passed sample comes from the real data distribution p_r). The two neural networks are trained in an adversarial fashion, such that the training objective of G is to maximize the probability of D making a mistake, i.e. fool the discriminator. The training objective of D is to minimize the probability to make misclassifications, i.e. correctly discriminate a fake sample as fake and a real sample as real. During the training phase of vanilla GAN, a random sample $z \sim p_z$ is drawn from a known latent random variable Z, usually uniformly or standard Gaussian distributed. That random vector z is then fed to the generator network, producing a fake sample $x_g = G(z)$, where $x_g \sim p_g$.[14] The observations from the real data set, $x_r \sim p_r$, together with the generated fake samples x_g are fed into the discriminator. The discriminator in vanilla GAN determines whether the input samples are real (1) or fake (0) through the discrimination function $D(x)$.
This setting corresponds to a minimax two-player game and can be formulized as follows

$$\min_G \max_D V(D,G) = \mathop{\mathbb{E}}_{x\sim p_r(x)}[\log D(x)] + \mathop{\mathbb{E}}_{z\sim p_z(z)}[\log(1 - D(G(z)))]. \tag{44}$$

The first term in the loss function is the expected logarithm of $D(x)$, the probability of correctly accepting a real sample, whereas the second term is the expected logarithm of $(1-D(G(z))$, the probability of correctly rejecting a fake sample. This loss function forces the discriminator to improve its real/fake discrimination capability. The feedback from the discriminator, i.e. $D(G(z))$, is then used by the generator to improve the quality of generated fake samples. Figure 34 below shows the workflow during vanilla GAN training.

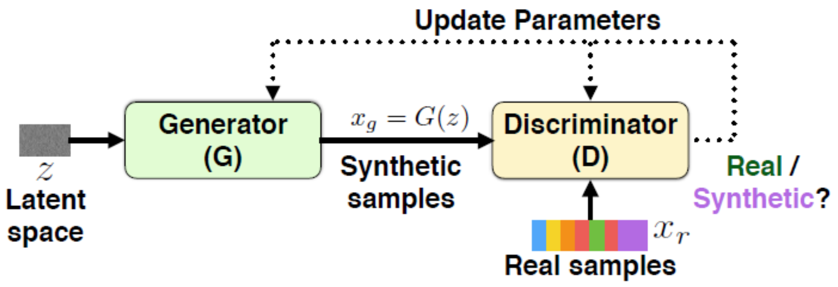


Figure 34: Vanilla GAN architecture workflow. The objective of the generator is to fool the discriminator, whereas the objective of the discriminator is to correctly classify input samples. Source: Adiga et al. (2018)

[14]x_g follows the distribution of the generator model that is parameterized with θ_g. Since a random vector z is fed into the generator network, we **implicitly** model the real data distribution p_r.

Optimizing vanilla GAN is equal to optimizing Jensen-Shannon divergence (see equation (37)) as can be seen below in the derivation. Before we proceed with the derivation, some additional equations are required.
Following holds $\forall a, b \neq 0$:

$$\textbf{if } y = a\log(y) + b\log(1-y),$$

the optimal $y, \forall y \in (0,1)$ can be computed by calculating the first derivative of the right hand side with respect to y and setting it to zero:

$$\frac{d}{dy}(a\log(y) + b\log(1-y)) = \frac{a}{y} - \frac{b}{1-y} \overset{!}{=} 0,$$

leads to ($\forall a, b \neq 0$):

$$\begin{aligned} \frac{a}{y} &= \frac{b}{1-y} \\ a(1-y) &= by \\ 1-y &= y\frac{b}{a} \\ 1 &= y\frac{a+b}{a} \\ y^* &= \frac{a}{a+b}. \end{aligned} \tag{45}$$

The optimization problem in equation (44) is formulated as

$$\begin{aligned} \min_G \max_D V(D,G) &= \mathop{\mathbb{E}}_{x\sim p_r(x)}[\log D(x)] + \mathop{\mathbb{E}}_{z\sim p_z(z)}[\log(1-D(G(z)))] \\ &= \mathop{\mathbb{E}}_{x\sim p_r(x)}[\log D(x)] + \mathop{\mathbb{E}}_{x\sim p_g(x)}[\log(1-D(x))] \\ &= \int_x [p_r(x)\log D(x) + p_g(x)\log(1-D(x))]\,dx. \end{aligned}$$

If the generator G is fixed, using equation (45) leads to the optimal discriminator

$$D^*(x) = \frac{p_r(x)}{p_r(x)+p_g(x)}. \tag{46}$$

The optimization for the generator with the optimal discriminator equals to

$$\begin{aligned} \min_G V(D^*,G) &= \min_G \int_x [p_r(x)\log D^*(x) + p_g(x)\log(1-D^*(x))]\,dx \\ &= \min_G \int_x \left[p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)} + p_g(x)\log(1-\frac{p_r(x)}{p_r(x)+p_g(x)})\right]dx \\ &= \min_G \int_x \left[p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)} + p_g(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}\right]dx \\ &=: \min_G C(G), \end{aligned} \tag{47}$$

where the term above includes the Jensen-Shannon divergence.

If we take a deeper look into the JS divergence (see equation (37)), we can derive that the term $C(G)$ from equation (47) is within the JS divergence.

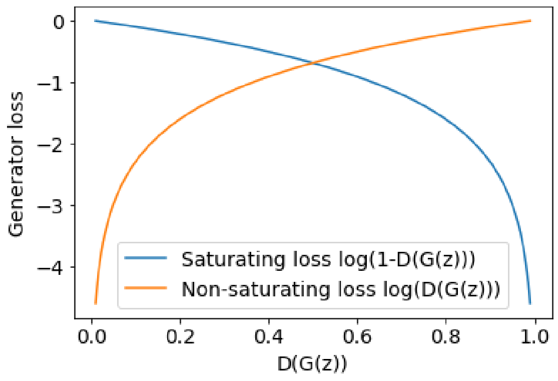
$$\begin{aligned}
\varphi_{JS}(p_r||p_g) &= \frac{1}{2}\int_x \left[p_r(x)\log\frac{2p_r(x)}{p_r(x)+p_g(x)} + p_g(x)\log\frac{2p_g(x)}{p_r(x)+p_g(x)}\right]dx \\
&= \frac{1}{2}\int_x \left[p_r(x)(\log 2 + \log\frac{p_r(x)}{p_r(x)+p_g(x)})\right]dx \\
&+ \frac{1}{2}\int_x \left[p_g(x)(\log 2 + \log\frac{p_g(x)}{p_r(x)+p_g(x)})\right]dx \\
&= \frac{1}{2}\int_x \left[p_r(x)\log 2 + p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}\right]dx \\
&+ \frac{1}{2}\int_x \left[p_g(x)\log 2 + p_g(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}\right]dx \\
&= \frac{1}{2}\left[\log 2 + \int_x p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}dx\right] \\
&+ \frac{1}{2}\left[\log 2 + \int_x p_g(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}dx\right] \\
&\overset{(47)}{=} \frac{1}{2}\left[\log(4) + C(G)\right].
\end{aligned}$$

Hence, for the optimal discriminator the generator objective is equal to minimizing the Jensen-Shannon divergence up to a constant

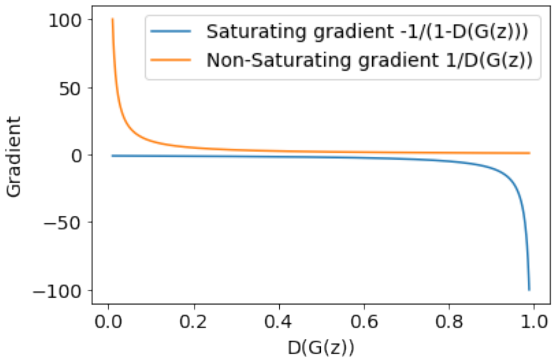
$$\min_G V(D^*, G) = \min_G C(G) = \min_G \left[2\varphi_{JS}(p_r||p_g) - 2\log 2\right]. \tag{48}$$

Since the Jensen-Shannon divergence is a statistical distance measure and satisfies the non-negativity condition $\varphi_{JS}(p_r||p_g) \geqslant 0\ \forall p_r, p_g$, the minimum value of $C(G)$ is achieved, if and only if the JS divergence is equal to zero. This is the case if $p_r(x) = p_g(x)$. Hence, the optimal generator G^* has to map any random vector z back into the $\mathcal{X}$ space, such that the generator distribution is equal to the real data distribution and obtaining optimal generator loss of $C(G^*) = -2\log 2 \approx -1.3863$. With $p_r(x) = p_g(x)$ the optimal discriminator decision is $D^*(x) = \frac{1}{2}$, which intuitively makes sense if we think about the idea of zero-sum game.
For training, once both objective functions are defined, the generator and discriminator are learned jointly by alternating gradient descent updates.
The minmax game between generator and discriminator needs to be solved in an iterative numerical approach.
'Optimizing D to completion in the inner loop of training is computationally prohibitive, and on finite datasets would result in overfitting. Instead, one alternates between d_{iters} steps of optimizing D and $g_{iters} = 1$ step of optimizing G.
This results in D being maintained near its optimal solution, as long as G changes slowly enough.' [Goodfellow et al. (2014)]

One often occurring problem in training deep learning models is the phenomenon of *exploding* or *vanishing* gradients. Early in the training of GAN, when the generator is poor and not able to create good data samples, the discriminator will directly recognize the passed input as fake ($D(G(z)) \approx 0$). Hence, the gradient $\nabla_{\theta_g} V(D,G)$ for updating the generator weights will be close to zero because the gradient of $\log(1-D(G(z))$ saturates close to zero. Recall that when computing the partial derivatives in early hidden layers, the chain rule states to compute a product of partial derivatives. So if the initial partial derivative (which comes from the objective function) is almost zero, the other partial derivatives as product will be close to zero as well.
So rather training G to minimize $\log(1-D(G(z)))$, we can train G to maximize $\log(D(G(z))$. The reformulation of this optimization is valid because each decision made by the discriminator lies within $(0,1)$, and the two generator objectives are symmetric to the vertical axis at 0.5 as visualized in Figure 35a below.
For that reason, 'the maximization of $\log(D(G(z)))$ results in the same fixed point of the dynamics of G and D but provides much stronger gradient early in learning.' [Goodfellow et al. (2014)]



(a) Generator loss functions.



(b) Generator loss derivatives.

Figure 35: Behaviour of saturating and non-saturating generator loss for its output and derivative. Non-saturating generator loss provides larger gradient values for smaller discriminator values.

In the beginning of training, the generator might output poor data samples, such that the discriminator decision is small. Hence, it is better to use the alternative non-saturating generator loss function. As introduced earlier in this Section, the classical vanilla GAN with saturating generator loss is theoretically motivated from game theory, and especially the objective in equation (44) is a **zero-sum-game** (minimax game). A zero-sum game is a game, 'in which all player's cost is always zero' [Goodfellow (2016)]. By switching to the heuristic of non-saturating generator loss, the game is no longer a zero-sum-game anymore. For the generator instead of minimizing the log probability of the discriminator being correct, the generator now maximizes the log probability of the discriminator being mistaken.

The algorithm for training vanilla GAN with the heuristic non-saturating generator loss function can be seen below.

Algorithm 1 Vanilla GAN: Minibatch stochastic gradient descent training. Default values: $d_{iters} = 1, m = 64, \alpha = 0.002$ and non-saturating generator loss.

```
Require:
    $\alpha$, the learning rate. $m$, the batch-size.
    $d_{iters}$, the number of iterations of the discriminator per generator iteration.
 1: for number of training epochs do
 2:     for $d_{iters}$ steps do
 3:         Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
 4:         Sample minibatch of $m$ data samples $\{x^{(1)}, ..., x^{(m)}\}$ from real data $x \sim p_r$
 5:         Update the discriminator $D$ by ascending its stochastic gradient:
                $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))\right]$
                                                                    ▷ max w.r.t. $\theta_d$
 6:     end for
 7:     Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
 8:     Update the generator $G$ by ascending its stochastic gradient:
            $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log D(G(z^{(i)}))$
                                                                    ▷ max w.r.t. $\theta_g$
 9: end for
10: The gradient-based updates can be used by any standard gradient-based learning
    rule. The default is ADAM optimizer with its default values.
```

Although vanilla GAN was a breakthrough in generative models, it still has some disadvantages and difficulties to train the two competing networks, such that the generator produces samples with high quality. In general, training vanilla GAN is known as slowly and unstable with several problems listed below.

Vanishing Gradient

As mentioned earlier, the optimization of vanilla GAN is performed in a numerical fashion: first train the discriminator in the inner loop and then the generator in the outer loop. If the discriminator is (almost) perfect, meaning classifying real samples as real ($D(x) \approx 1, \forall x \in \text{supp}(p_r)$) and generated samples as fake ($D(x) \approx 0, \forall x \in \text{supp}(p_g)$), the generator gradients $\nabla_{\theta_g} V(D, G)$ also tend to become zero. Hence, the generator weights θ_g cannot be updated properly since no useful gradients are available. As a result, training vanilla GAN faces a dilemma.

1. If the generator behaves badly and is poor, the generator does not have accurate gradient feedback for updating the generator weights. Switching to

non-saturating generator loss as done in Algorithm 1 might improve training stability as can be seen in Figure 35b.

2. If the discriminator behaves good and can clearly distinguish between real and fake data, the generator gradients (with saturating generator loss) will be close to zero and updating the generator weights will have very small changes. As a result, learning becomes extremely slow. In case of non-saturating generator loss the gradients of the generator would explode (see figure 35b) and the learning would diverge and become very unstable.

Mode collapse

In general, real life data distributions are multimodal. Mode collapse in generative adversarial networks is the lack of diversity in generated samples. In the worst and extreme case, mode collapse means that the generator network maps any latent random input z to one specific point $\tilde{x}$.
Theis et al. (2015) and Arjovsky & Bottou (2017) made an in-depth analysis towards training and evaluating GAN with respect to good sample quality of the generator network. One key point why GAN training (or rather general machine learning) is hard, is the choice of objective function. Theis et al. (2015) show the effects of different divergence metrics on a simple toy example, where an isotropic Gaussian was fit to data drawn from a mixture of Gaussians. When minimizing KL divergence (KLD), the fit distribution 'avoids assigning extremely small probability to any data point but assigns a lot of probability mass to non-data regions' [Theis et al. (2015)]. To illustrate this idea for image synthesis, in this case the GAN would produce samples that look really unrealistic. For minimizing the Jensen-Shannon divergence (JSD), the fit distribution 'yields a Gaussian which fits one mode well, but which ignores other parts of the data' [Theis et al. (2015)]. In this case, mode collapse happened, where the GAN always produce data points coming from one mode, e.g. the generator network always generates images of cats.

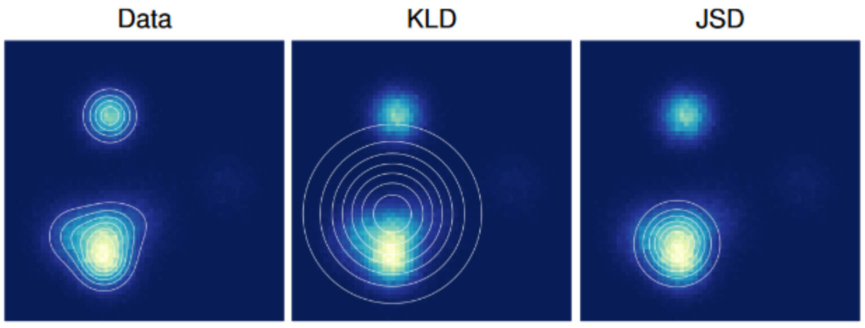


Figure 36: An isotropic Gaussian was fit to data drawn from a mixture of Gaussians by either minimizing Kullback-Leibler divergence (KLD) or Jensen-Shannon divergence (JSD). The different fits demonstrate different tradeoffs made by two measures of distance between distributions. Image is modified from source: Theis et al. (2015)

It is therefore worth investigating the GAN training for different divergence metrics regarding probability distributions as done by Arjovsky et al. (2017), leading to a new variant of GAN described in the next Section 2.4.3.

2.4.3 Wasserstein GAN

The Wasserstein GAN (WGAN) is a variant of the vanilla GAN and minimizes the Wasserstein-1 distance as stated in equation (43) between the distribution of real data and the distribution of generated data [Arjovsky et al. (2017)]. However, the infimum over the set of all possible joint distributions $\gamma \in \prod(p_r, p_\theta)$ is highly intractable. On the other hand, the Kantorovich-Rubinstein duality [Villani (2008); Santambrogio (2015); Herrmann (2017)] enables formulating the primal problem in equation (43) into its dual form

$$\varphi_W(p_r, p_\theta) = W(p_r, p_\theta) = \sup_{||f||_L \leq 1} \mathop{\mathbb{E}}_{x \sim p_r}[f(x)] - \mathop{\mathbb{E}}_{x \sim p_\theta}[f(x)], \tag{49}$$

where the supremum (least upper bound) is over all $1-$Lipschitz functions $f : \mathcal{X} \to \mathbb{R}$. Now, we do not need to find the optimal transport plan[15] γ^*, which satisfies the two marginalization constraints in equation (41), but instead a function f that is $1-$Lipschitz continuous. The mathematical definition of K$-$Lipschitz continuity and its application in a sketch of proof formulating the primal problem to its dual version is provided in the Appendix A.1 and A.3.
Intuitively, a Lipschitz continuous function is restricted in how fast it can change. Let d_X and d_Y be distance functions[16] on two compact spaces $\mathcal{X}$ and $\mathcal{Y}$. A function $f : \mathcal{X} \to \mathcal{Y}$ is $K-$Lipschitz if there exists a real constant $K \geq 0$, such that for all $x_1, x_2 \in \mathcal{X}$ the following property holds

$$d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2). \tag{50}$$

Consider the example of a real-valued function $f : \mathbb{R} \to \mathbb{R}$. This function is called $K-$Lipschitz if and only if there a exists a real constant $K \geq 0$, such that for all $x_1, x_2 \in \mathbb{R}$, when using l_1-norm in $\mathbb{R}$, following constraint holds

$$\begin{aligned} |f(x_1) - f(x_2)| &\leq K|x_1 - x_2| \\ \frac{|f(x_1) - f(x_2)|}{|x_1 - x_2|} &\leq K. \end{aligned}$$

The constraint above restricts the slope of a secant between two points of the $K-$Lipschitz function by an upper bound K. The linear function $f(x) = x$ is $1-$Lipschitz continuous on $\mathbb{R}$ as shown in Example 1 in Appendix A.1.

[15] Also called joint probability density of p_r and p_θ. In terms of optimal transport theory, this is often called *coupling*.

[16] E.g., the l_p norm applied on the difference of two points x_1, x_2 leading to $d_X = ||x_1 - x_2||_p$.

Since the optimization over all $1-$Lipschitz functions is still intractable, the objective in equation (49) can be approximated by considering $K-$Lipschitz functions.
If we replace the supremum over $1-$Lipschitz functions with the supremum over $K-$Lipschitz functions, then the supremum is $K \cdot W(p_r, p_\theta)$ because every $K-$Lipschitz function is $1-$Lipschitz if we divide it by K. The supremum over $K-$Lipschitz functions $\{f : ||f||_L \leq K\}$ is still intractable but approximating is easier: suppose we have a parameterized function family $\{f_w\}_{w \in \mathcal{W}}$, where w are some weights and $\mathcal{W}$ is the set of all possible weights for this function family. Further suppose that these functions f are all $K-$Lipschitz for some $K \geq 0$. It is always possible to approximate the supremum with a maximum in case the supremum cannot be reached,

$$\begin{aligned} \max_{w \in \mathcal{W}} \mathop{\mathbb{E}}_{x \sim p_r}[f_w(x)] - \mathop{\mathbb{E}}_{x \sim p_\theta}[f_w(x)] &\leq \sup_{||f||_L \leq K} \mathop{\mathbb{E}}_{x \sim p_r}[f(x)] - \mathop{\mathbb{E}}_{x \sim p_\theta}[f(x)] \\ &= K \cdot W(p_r, p_\theta). \end{aligned} \tag{51}$$

The reason for this approximation is to consider solving the optimization problem,

$$\max_{w \in W} \mathop{\mathbb{E}}_{x \sim p_r}[f_w(x)] - \mathop{\mathbb{E}}_{x \sim p_\theta}[f_w(x)]. \tag{52}$$

If the supremum in equation (51) is attained for some weight $w \in \mathcal{W}$, then the Wasserstein-1 distance $W(p_r, p_\theta)$ was successfully computed, scaled by a constant K. Nevertheless, the authors of the WGAN claim that the supremum probably will not be achieved by solving the above optimization. In this case, the approximation quality depends on what $K-$Lipschitz functions are missing from $\{f_w\}_{w \in \mathcal{W}}$. Coming back to the framework of GANs, where the discriminator is competing against a generator, the Wasserstein GAN attempts to solve following optimization problem

$$\min_G \max_{D \in \mathcal{D}} \mathop{\mathbb{E}}_{x \sim p_r(x)}[D(x)] - \mathop{\mathbb{E}}_{z \sim p_z(z)}[D(G(z))]. \tag{53}$$

It is worth mentioning that the discriminator takes the role as a *critic* and outputs a real-valued number instead of a probability $\in (0, 1)$ for an observation to be true or fake. Furthermore, D has to come from a set $\mathcal{D}$ that states the set of $1-$Lipschitz continuous functions. The critic is trained to learn a $K-$Lipschitz continuous function to help computing the approximation of Wasserstein-1 distance. As the loss decreases in training, the Wasserstein-1 distance gets smaller and the samples generated by the generator model gets closer to the real data distribution. One can observe that the maximum in equation (53) is obtained, when as large as possible values are allocated to samples from p_r and as as small as possible values to samples from p_g. Meanwhile the minimum over the generator network G attempts to minimize that difference as a competing counterpart towards the critic.
Hence, the generator network G is forced to push the distribution p_g as close to p_r such that the Wasserstein-1 distance is equal to zero. The Wasserstein-1 distance is

zero if and only if the generated data distribution p_g is exactly the real data distribution p_r. The WGAN has several significant practical benefits over the standard vanilla GAN (in equation (44)) [Arjovsky et al. (2017)].

1. A meaninful loss metric that correlates well with the generator's convergence and sample quality.

2. Improved stability of the optimization process.

The first point can be explained that within the WGAN algorithm the critic D is trained in the inner optimization relatively well up to convergence, before the outer optimization for the generator is proceeded. As the overall loss function decreases, one can observe that the generated samples by G have high quality and are like samples from the true data distribution. The second point goes along with the first point. Since the authors of the WGAN advise to train the critic up to convergence, useful gradient information can be passed to the generator[17] because the better the critic is trained, the better the approximation of the Wasserstein-1 distance is achieved. Arjovsky et al. (2017) claim that one major drawback of vanilla GAN was the vanishing gradient, if the discriminator was trained too long and could tell any generated sample from the generator G as fake. In that case as seen in Figure 35b, the gradients passed to the generator are almost zero. But with WGAN, the vanishing gradient problem is solved as the gradient of the overall objective in equation (53) w.r.t. $G(z)$ is linear as shown in Figure 37 below.

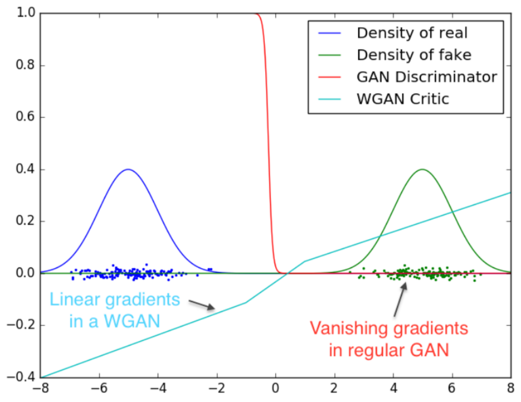


Figure 37: Optimal discriminator and critic when learning to differentiate two Gaussians. The discriminator of a vanilla GAN saturates and results in vanishing gradients. The WGAN critic however, provides very clean gradients on all parts of the space. Source: Arjovsky et al. (2017)

It is important to remember that the WGAN algorithm only works if the critic D is a $1-$Lipschitz function. The reason for this constraint lies in the Kantorovich-Rubinstein duality that enables to formulate the primal problem (infimum) to its dual version (supremum), see Appendix A.3. The authors of WGAN suggest to restrict the model weights w of the critic into a compact space, e.g. a fixed box

[17]Since the generator G is coupled through the critic D through $D(G(z))$.

$\mathcal{W} = [-0.01, 0.01]^l$. In terms of MLPs, this means that the values of the weight matrices and bias vectors for each hidden layer are clipped into the range $[-0.01, 0.01]$ after each gradient update. Reasons, why weight clipping can enforce the 1−Lipschitz continuity is explained by Anil et al. (2019). Nevertheless, Arjovsky et al. (2017) argue that 'weight clipping is clearly a terrible way to enforce a Lipschitz constraint' since the lower and upper clipping bounds are hyperparameters and affect the training of WGAN substantially: 'if the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic up to optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big [because the chain rule requires a product of partial derivatives]'. For that reason, research on the WGAN has been made, leading to the *Improved WGAN* that will be explained in the next Section 2.4.4.
Just like in vanilla GAN training, Wasserstein GAN is achieved in the same way. First we fix the generator G and train the critic D for d_{iters} steps (up to convergence). Then we fix the critic and train the generator for $g_{iters} = 1$ step. By training the critic up to convergence, we hope to approximate the Wasserstein-1 distance well, such that when backpropagating errors for the generator model via the critic, useful (non-saturating) gradient information can be passed backwards.

Algorithm 2 Wasserstein GAN with weight clipping:
Default values: $d_{iters} = 5$, $\alpha = 0.00005, c = 0.01, m = 64$.

```
Require:
    $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch-size.
    $d_{iters}$, the number of iterations of the critic per generator iteration.
 1: for number of training epochs do
 2:     for $d_{iters}$ steps do
 3:         Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
 4:         Sample minibatch of $m$ data samples $\{x^{(1)}, ..., x^{(m)}\}$ from real data $x \sim p_r$
 5:         Update the critic $D$ by ascending its stochastic gradient:
                $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ D(x^{(i)}) - D(G(z^{(i)})) \right]$$
                                                                    ▷ max w.r.t. $\theta_d$
 6:         Clip the critic weights: $\theta_d \leftarrow \text{clip}(\theta_d, -c, c)$
 7:     end for
 8:     Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
 9:     Update the generator $G$ by descending its stochastic gradient:
                $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} -D(G(z^{(i)}))$$
                                                                    ▷ min w.r.t. $\theta_g$
10: end for
11: The gradient-based updates can be used by any standard gradient-based learning
    rule. The default is RMSProp optimizer with its default values.
```

2.4.4 Improved Wasserstein GAN

It was proposed by Arjovsky et al. (2017) to train a generator and critic network by minimizing the primal Wasserstein-1 distance (equation (43)). The authors claim that the proposed distance measure holds better properties compared to the Jensen-Shannon divergence (equation (48)) from vanilla GAN in terms of convergence and sample quality. The change of divergence metric in WGAN introduced a new optimization problem (see equation (53)) that required the critic network to lie within the space of $1-$Lipschitz functions. The authors of WGAN enforced this constraint through weight clipping, i.e. by constraining the entries of weight matrices and bias vectors of the critic to be smaller than a given value in magnitude.
However, the weight clipping method can lead to undesired behaviour as analyzed by Gulrajani et al. (2017) and will be summarized later in this Section. In the *Improved Wasserstein GAN* algorithm, Gulrajani et al. (2017) propose a regularization term based on results from optimal transport theory [Villani (2008)]. This regularization term is a **gradient penalty** term, penalizing any deviation of the gradient 2-norm of the critic network (w.r.t its input) from the value one. The main idea for this regularization term comes from their stated proposition and logical conclusion from this, regarding an optimal critic function f^*.

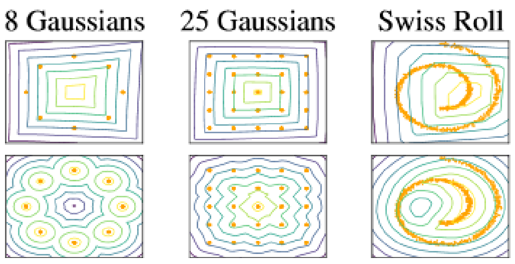
Proposition 1. *Let $\mathbb{P}_r$ and $\mathbb{P}_g$ be two distributions in $\mathcal{X}$, a compact metric space. Then, there is a $1-$Lipschitz function f^* which is the optimal solution of the problem $\max_{||f||_L \leq 1} \mathbb{E}_{y\sim\mathbb{P}_r}[f(y)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]$ (equation (49)). Let π^* be the optimal coupling between $\mathbb{P}_r$ and $\mathbb{P}_g$, defined as the minimizer of (the primal Wasserstein-1 objective) $W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\pi\in\prod(\mathbb{P}_r,\mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\pi}[||x-y||]$, where $\prod(\mathbb{P}_r, \mathbb{P}_g)$ is the set of joint distributions $\pi(x,y)$ whose marginals are $\mathbb{P}_r$ and $\mathbb{P}_g$, respectively. Then, if f^* is differentiable, $\pi^*(x=y)=0$, and $x_t = tx + (1-t)y$ with $0 \leq t \leq 1$, it holds that $\mathbb{P}_{(x,y)\sim\pi^*}\left[\nabla f^*(x_t) = \frac{y-x_t}{||y-x_t||}\right] = 1$.*

Corollary 1. *f^* has gradient norm one almost everywhere under $\mathbb{P}_r$ and $\mathbb{P}_g$, on each secant $\overline{xy}$, where the points (x,y) are samples from the optimal coupling: $(x,y)\sim\pi^*$.*

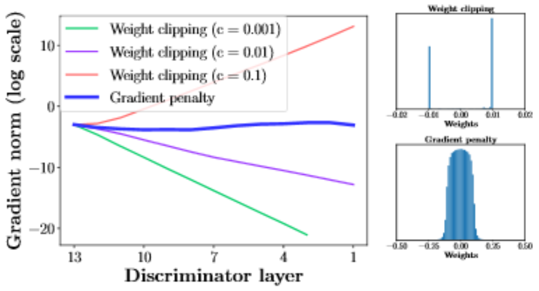
A proof for Proposition 1 is provided by Gulrajani et al. (2017).
Proposition 1 makes use of the $1-$Lipschitz continuity: for all (x,y) in the support of the optimal coupling π^*, the maximal norm of a partial derivative at any point into any direction is one. So now, when considering the line between x and y, for each point $x_t = tx + (1-t)y$ the partial derivative has norm equal to one into the direction pointing from the real data point x to the generated data point y (which are coupled by the optimal π^*). Now using the conclusion from the sentence before, since the maximal norm of a partial derivative at any point into any direction is one, the chosen direction is the direction of maximal descent/ascent, leading to the gradient. Therefore, the gradient for each point between x and y has a norm of one.

Difficulties with weight constraints

Gulrajani et al. (2017) found out that ‘weight clipping in [classical] WGAN leads to optimization difficulties’. In addition to classical hard clipping of the magnitude for each weight, Gulrajani et al. (2017) tried different weight constraints, such as L1 and L2 weight decay (see equation (15)). Nonetheless, soft constraints still led to difficulties as mentioned in their paper. In general, Gulrajani et al. (2017) state two main problems that are caused by weight clipping as illustrated in Figure 38.



(a) Value surfaces of WGAN critics trained to optimality on toy datasets using (top) weight clipping and (bottom) gradient penalty. Critics trained with weight clipping fail to capture higher moments of the real data distribution.



(b) (left) Gradient norms of WGAN critics during training on the *Swiss-Roll* dataset either explode or vanish when using weight clipping, but not when using gradient penalty. (right) Weight clipping (top) pushes weights towards two values (the extremes of the clipping range), unlike gradient penalty (bottom).

Figure 38: Gradient penalty in WGAN does not exhibit undesired behaviour like weight clipping. Source: Gulrajani et al. (2017)

Capacity underuse

Applying hard clipping on the weights on a lipschitz continuous function restricts the critic towards much simpler functions. In order to illustrate this, several experiments on the toy datasets *8-Gaussians, 25-Gaussians, Swiss-Roll* were conducted as shown in Figure 38a. For those toy datasets, the metric space is two-dimensional, hence $\mathcal{X} \subset \mathbb{R}^2$. In those experiments Gulrajani et al. (2017) compared the critic network behaviour regarding its value function $D(\cdot)$ in WGAN with weight clipping against the WGAN with gradient penalty. The authors held the generator network fixed to be the real data added with standard Gaussian noise. For both algorithms, the critic was trained up to convergence and a level set / contour plot on the critic’s value over a batch of fixed generator samples was plotted. The yellow dots in the *Gaussians* toy datasets show the mode of data whereas the yellow dots in the *Swiss-Roll* visualize real data points. For both algorithms, the yellow contour lines correspond to high values and purple lines to low values for the critic D. The WGAN with weight clipping (first row in Figure 38a) did not capture the modes very well in contrast to WGAN with gradient penalty (second row).

Exploding and vanishing gradients
As mentioned in the end of Section 2.4.3, the weight clipping procedure can lead for arbitrary small or large clipping bounds to vanishing or exploding gradients. This was also investigated by Gulrajani et al. (2017) in Figure 38b. The authors trained a feedforward network critic on the *Swiss-Roll* dataset comparing WGAN with weight clipping and gradient penalty. When updating the model weights during backpropagation (see Section 2.2.5.2), the partial derivatives for the early layers either explode or vanish due to the multiplication of large or small partial derivatives computed from the back layers.

Gradient Penalty
In order to enforce the critic D to be 1−Lipschitz continuous, the gradient penalty term is included when updating the critic in the inner loop. ‘A differentiable function is 1−Lipschitz if and only if it has gradients with euclidean norm at most one everywhere, so we consider directly constraining the gradient norm of the critic’s output with respect to its input’ [Gulrajani et al. (2017)].

$$GP = \lambda \underset{\hat{x}\sim p_{\hat{x}}}{\mathbb{E}} \left[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2 \right]. \tag{54}$$

Concluding from Corrolary 1, Gulrajani et al. (2017) choose that the point $\hat{x}$ is sampled uniformly from a secant between pairs of points from the **marginals** $x \sim p_r$ and $y \sim p_g$. Formally, we obtain the linear dependency, as shown in Figure 39.

$$\hat{x} = tx + (1-t)y \text{ , where } t \sim U(0,1), x \sim p_r, y \sim p_g. \tag{55}$$

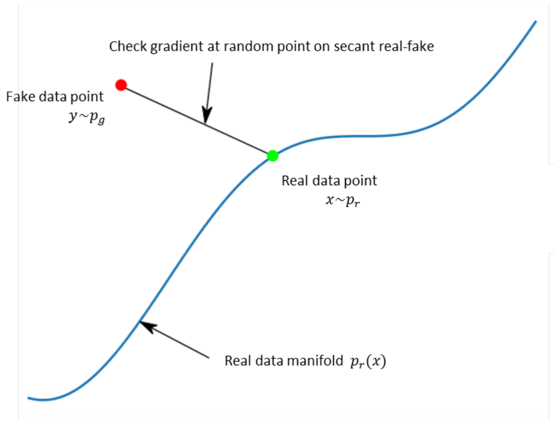

Check gradient at random point on secant real-fake
Fake data point
$y \sim p_g$
Real data point
$x \sim p_r$
Real data manifold $p_r(x)$

Figure 39: $\hat{x}$ is sampled uniformly from the marginals $x \sim p_r$ and $y \sim p_g$.
Source: modified from Viehmann (2017)

This sampling approach though, does not follow Corrolary 1 because Proposition 1 states that the optimal critic D^* will have gradient norm one (almost everywhere) only between pairs x and y that are sampled from the **optimal coupling** $\pi^*(x, y)$

and not the **marginals** p_r and p_g respectively, which Kodali et al. (2018) and Wei et al. (2018) identified as potential caveats. Kodali et al. (2018) suggest to use a *local penalty* for real data points instead of the coupled penalty of marginals. Referring to Figure 39, the selected point would then be closer to the green point. Wei et al. (2018) take further analysis and propose another regularization term that directly works with the definition of Lipschitz continuity (equation (50)) for noisy (real data) points. Nonetheless for completion, Gulrajani et al. (2017) suggest following maximization problem for solving the dual problem with gradient penalty regularization term as defined in equation (54) to enforce 1−Lipschitz continuity.

$$\max_{w \in W} \mathop{\mathbb{E}}_{x \sim p_r}[f_w(x)] - \mathop{\mathbb{E}}_{x \sim p_g}[f_w(x)] - \lambda \mathop{\mathbb{E}}_{\hat{x} \sim p_{\hat{x}}}[(||\nabla_{\hat{x}} f_w(\hat{x})||_2 - 1)^2]. \tag{56}$$

The minmax WGAN-GP optimization problem is again solved in alternating fashion within two inner loops, i.e.

$$\min_G \max_{D \in \mathcal{D}} \mathop{\mathbb{E}}_{x \sim p_r(x)}[D(x)] - \mathop{\mathbb{E}}_{z \sim p_z(z)}[D(G(z))] - \lambda \mathop{\mathbb{E}}_{\hat{x} \sim p_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2]. \tag{57}$$

Algorithm 3 Wasserstein GAN with gradient penalty:
Default values: $d_{iters} = 5$, $\alpha = 0.0001$, $\lambda = 10$, $m = 64$, $\beta_1 = 0.5$, $\beta_2 = 0.9$.

```
Require:
    $\alpha$, the learning rate. $\lambda$, the gradient penalty coefficient. $m$, the batch-size.
    $d_{iters}$, the number of iterations of the critic per generator iteration.
 1: for number of training epochs do
 2:     for $d_{iters}$ steps do
 3:         Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
 4:         Sample minibatch of $m$ data samples $\{x^{(1)}, ..., x^{(m)}\}$ from real data $x \sim p_r$
 5:         Sample minibatch of $m$ random numbers $\{t^{(1)}, ..., t^{(m)}\} \sim U(0,1)$
 6:         Compute $\{\hat{x}^{(1)}, ..., \hat{x}^{(m)}\}$, where $\hat{x}^{(i)} = t^{(i)}x^{(i)} + (1 - t^{(i)})G(z^{(i)})$
 7:         Update the critic $D$ by ascending its stochastic gradient:
                $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[D(x^{(i)}) - D(G(z^{(i)})) - \lambda(||\nabla_{\hat{x}^{(i)}} D(\hat{x}^{(i)})||_2 - 1)^2\right]$
                                                                        ▷ max w.r.t. $\theta_d$
 8:     end for
 9:     Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from prior $z \sim p_z$
10:     Update the generator $G$ by descending its stochastic gradient:
                $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} -D(G(z^{(i)}))$
                                                                        ▷ min w.r.t. $\theta_g$
11: end for
12: The gradient-based updates can be used by any standard gradient-based learning
    rule. The default is ADAM optimizer with $\beta_1$ and $\beta_2$ from default values.
```

3 Dataset

The datasets for training a generative model for drug discovery (Section 4.3 and 4.4) were provided by *BenevolentAI*'s GuacaMol Benchmark [Brown et al. (2019)] and are split into training, test and validation sets.
The datasets for training the generative models were extracted from ChEMBL-24 database [Mendez et al. (2018)]. One main advantage of ChEMBL-24 is that it only contains chemical structures which have been synthesized and tested against a biological target[18], such as Dopamine receptor D2 (DRD2) [Olivecrona et al. (2017)] or EGF-Receptor and BACE1 [Winter et al. (2019)].
Another benchmark study named MOSES [Polykovskiy et al. (2018)] used the ZINC database [Irwin & Shoichet (2005)] as basis and applied filtering. One disadvantage of the ZINC database is that it contains molecules which have not been synthesized yet.
The datasets provided by GuacaMol have been further preprocessed including following steps [Brown et al. (2019)]:

1. removal of salts.
2. charge neutralization.
3. removal of molecules with SMILES strings longer than 100 characters.
4. removal of molecules containing any atomatic element other than from the set {`H, B, C, N, O, F, Si, P, S, Cl, Se, Br, I`}.
5. removal of molecules with a larger ECFP4 similarity[19] than 0.323 compared to a holdout set consisting of ten marketed drugs (celecoxib, aripiprazole, cobimetinib, osimertinib, troglitazoe, ranolazine, thiothixene,albuterol, fexofenadine, mestranol). This allows to define similarity benchmarks for targets that are not part of the training set.

The training set consists of 1 273 104 unique SMILES representations. Test set and validation set each contains 238 706 and 79 568 unique samples. Those two sets will not be included in the training of the GAN used for learning the ChEMBL data space in Section 4.3. The reason for that decision lies in the fact that a fair way of conducting the *distribution-learning* benchmark from GuacaMol is wanted.
After a short analysis it turns out that the training set consists of 1 272 852 canonical SMILES. Nevertheless, the training of the GAN in Section 4.3 will be performed on the provided full training set that includes the 252 non-canonical SMILES.

[18]In de novo drug design this methodology is often called *inverse QSAR*: the objective is to find compounds which are biological active against a target, i.e. have high (predicted) binding-affinity

[19]The ECFP4 is a bit vector representation for molecules using molecular fingerprints as illustrated in Figure 2. For computing the similarity between two bit vectors the *Tanimoto coefficient* was selected.

4 Application

The application of this thesis is divided into two general parts, using the three explained variants of generative adversarial networks for continuous data.
Section 4.2 describes the training of GANs to learn multivariate normal data as a proof-of-concept experiment.
The main idea behind this is to come up with an optimal network architecture and algorithm for training the generative model. Furthermore, training a GAN on multivariate normal data has the purpose to show that GANs are powerful generative models that can learn a (dense) data distribution, even in high dimensional space.
Section 4.3 explains the training of GANs to learn the distribution of *continuous data-driven molecular descriptors* (`cddd`) as described in Section 2.3.1.
In both two parts, different optimization parameters and network architectures were extensively analyzed by trying out different settings. However, due to the scope of this thesis, a comprehensive evaluation of different architectures, optimization algorithms, activation functions, and weight initializations would be infeasible for this work. This study uses the vanilla GAN with non-saturating generator loss (Algorithm 1), Wasserstein GAN with weight clipping (Algorithm 2) and Wasserstein GAN with gradient penalty (Algorithm 3) algorithms for learning the respective data spaces in Section 4.2 and Section 4.3. At the beginning of each training epoch, several evaluation metrics are computed to display, whether the GAN is able to synthesize reasonable and good samples. Note that those evaluation metrics are not included in the overall optimization objective of the aforementioned algorithms. Since we want to exploit the power and capacity of GANs, additional loss terms, e.g. the mean of a set of generated samples being close to the mean of a set from real samples, are not included in the overall optimization objective. This in practice, however, is possible and depends on the application for each machine-learning engineer.

4.1 Technical Information

This study utilizes the `Pytorch` [Paszke et al. (2017)] deep learning framework as backend with the programming language `Python 3.6` [Van Rossum & Drake Jr (1995)] as frontend to train neural networks. Since training deep neural networks is computationally expensive, the Pytorch library with *gpu-support* was selected. This library utilizes other libraries such as `CUDA` and `cuDNN` [Chetlur et al. (2014)] which are highly optimized for parallel computation of linear algebra operations on GPUs (graphical processung units).
All models were trained on a linux cluster with seven Tesla M40 GPUs, each consisting of 24 GB ram. For visualization and plotting graphs and training processes, either `matplotlib` [Hunter (2007)] or `tensorboardX` [Huang (2017)] was used.

4.2 Learning Multivariate Normal Distribution

For learning a multivariate normal distribution with a specific mean vector $m_d \in \mathbb{R}^d$ and identity covariance matrix $I_d \in \mathbb{R}^{d \times d}$, the dimension was selected to be $d = 50$. The mean value vector m_d was set to $\vec{4}$. Since the data generating process (DGP) is known, a random dataset of n_{samples}=1 000 000 samples following the above distribution was generated and saved on the local disk. This dataset was retrieved in the training process and has the matrix shape of $1000000 \times d$.
The random noise distribution was selected to be the uniform distribution with lower bound -1 and upper bound $+1$, i.e. $Z \sim U(-1, 1)$.

4.2.1 Evaluation Metrics

At the beginning of each training epoch, a batch of b fake samples is synthesized by the generator network. Let $\tilde{x}^{(i)}$ with $\tilde{x}^{(i)} \in \mathbb{R}^d$ be the $i-$th sample of the batch $i = 1, ..., b$. Let $\widetilde{X} = [\tilde{x}^{(1)\mathsf{T}}, ..., \tilde{x}^{(b)\mathsf{T}}]$ be the fake sample (batch) data matrix with $\widetilde{X} \in \mathbb{R}^{b \times d}$. In order to evaluate the generated samples, a mean value metric and two covariance metrics are computed. As the generative model gets better, we expect those computed metrics to decrease with increasing epoch number. As stated at the beginning of Section 4, the upcoming introduced evaluation metrics are not included into the optimization objective to see, whether the GAN can learn the true data distribution without any additional loss function terms.

Mean Criterion

The mean criterion has the purpose to evaluate whether the GAN can model the first moment of the real data distribution. With increasing training epoch we expect each synthesized sample from the generator network to be close to $\vec{4} \in \mathbb{R}^d$, which is the true mean vector of the normal distribution from the DGP. In conclusion, when computing the mean over the rows for each (column)-dimension of the batch data matrix $\widetilde{X}$, we expect the row-mean vector $\bar{x}$ to be close to $\vec{4}$. The row-mean of the batch matrix is computed with

$$\bar{x} = (\bar{x}_1, ..., \bar{x}_d)^{\mathsf{T}}, \tag{58}$$

where $\bar{x}_j = \frac{1}{b}\sum_{i=1}^{b} \widetilde{X}(i,j)$ is the mean of the $j-$th column and $\widetilde{X}(i,j)$ denotes the element in row i and column j of the batch data matrix. Finally, the mean evaluation criterion c_m is defined with

$$c_m(\widetilde{X}) := \frac{1}{d}\sum_{j=1}^{d} |\bar{x}_j - 4|, \tag{59}$$

where we expect c_m to decrease and converge towards zero during training.

Covariance Criteria

In order to define the two upcoming covariance criteria, the estimated covariance matrix from a generated batch $\widetilde{X} \in \mathbb{R}^{b \times d}$ needs to be computed with

$$\hat{\Sigma}_{\widetilde{X}} = \frac{1}{b-1} \widetilde{X}_c^{\intercal} \widetilde{X}_c, \tag{60}$$

where $\widetilde{X}_c = \widetilde{X} - 1_b \bar{x}^{\intercal}$ denotes the centered batch matrix, 1_b the $b-$dimensional unit (column) vector and $\bar{x}$ the row-mean vector from equation (58).
The correlation matrix is obtained with the estimated covariance matrix

$$\hat{R}_{\widetilde{X}} = D^{-1} \hat{\Sigma}_{\widetilde{X}} D^{-1}, \tag{61}$$

where D is the matrix of square-rooted diagonal elements from the estimated covariance matrix, i.e. $D = \sqrt{\text{diag}(\hat{\Sigma}_{\widetilde{X}})}$.
The first covariance criterion c_{l_1} is based on the l_1 norm and computes the sum of absolute differences between sample correlation matrix $\hat{R}_{\widetilde{X}}$ and unit correlation matrix I_d.

$$c_{l_1}(\widetilde{X}) := \frac{1}{d^2} \sum_{i=1}^{d} \sum_{j=1}^{d} |\hat{R}_{\widetilde{X}}(i,j) - I_d(i,j)|, \tag{62}$$

where (i,j) is the element in the $i-$th row and $j-$th column of the respective matrix.
The second covariance criterion c_{fb} is based on the frobenius norm. The frobenius norm of a matrix $\Sigma \in \mathbb{R}^{n \times m}$ is defined as

$$||\Sigma||_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} \Sigma(i,j)^2}\,. \tag{63}$$

In order to compare the estimated correlation matrix with the unit correlation matrix, the second covariance criterion is defined as

$$c_{fb}(\widetilde{X}) := \frac{1}{d} \left[||R_{\widetilde{X}}||_F - ||I_d||_F \right] . \tag{64}$$

Similar to the mean criterion metric, the two covariance criteria are expected to decrease with increasing training epoch.

In this proof-of-concept showcase, several network architectures were tested and their evaluation metrics compared. The final results with the training settings as well as network structures are displayed in the next Section. For all experiments, either ADAM optimizer [Kingma & Ba (2014)] or RMSprop optimizer [Hinton (2012)] were chosen to update the network parameters. In the first experiment, the three GAN variants were compared to each other. The intention was to confirm whether the Wasserstein GAN with gradient penalty (Algorithm 3) is superior to vanilla GAN and Wasserstein GAN with weight clipping. For that reason, the same network architectures with different optimizers were selected.

4.2.2 Results

In this proof-of-concept experiment, batch normalization as well as layer normalization [Ba et al. (2016)] for the generator network were tested. It turns out that adding batch normalization layers [Ioffe & Szegedy (2015)] in the generator network is crucial for generating good samples. In batch normalization, an activated batch $B \in \mathbb{R}^{b \times d}$, will be normalized by subtracting the batch-mean $\mu \in \mathbb{R}^d$ and dividing by the batch standard deviations $\sigma \in \mathbb{R}^d$ along the batch dimension b. Layer normalization computes the layer-mean and standard deviations along the feature dimension d to obtain $\mu \in \mathbb{R}^b$ and $\sigma \in \mathbb{R}^b$. Normalizing is conducted in the similar way along the feature dimension. The architectures for selected generator and discriminator network are shown below.

Table 3: Illustration of the generator network architecture. It consists of three fully connected hidden layers with batch normalization and leaky ReLU activation [Xu et al. (2015)].

Name	Type	Input size	Output size
input	input: $z \sim U(-1, 1)$	100	-
FC1	linear	100	256
	batch normalization	256	256
	leaky ReLU	256	256
FC2	linear	256	512
	batch normalization	512	512
	leaky ReLU	512	512
FC3	linear	512	256
	batch normalization	256	256
	leaky ReLU	256	256
output	linear	256	50
	batch normalization	50	50

Table 4: Illustration of the discriminator/critic network architecture. Depending on the algorithm, sigmoid activation function (equation (3)) is deployed in the output layer. This only holds for the vanilla GAN Algorithm 1.

Name	Type	Input size	Output size
input	input: $x \simeq \mathcal{N}(4, I)$	50	-
FC1	linear	50	128
	leaky ReLU	128	128
FC2	linear	128	256
	leaky ReLU	256	256
FC3	linear	256	512
	leaky ReLU	512	512
output	linear	512	1

The learning rates for the generator and discriminator/critic networks were set to $\alpha_g = 0.0002$ and $\alpha_d = 0.0004$ for both RMSprop- and ADAM optimizer.
All GAN variants were trained for $n_{\text{epochs}} = 150$ epochs. At the beginning of every epoch, $b = 5000$ samples were generated and the metrics from equation (59), (62) and (64) computed for evaluation. In this experiment, the same architectures for generator and discriminator network were used (see Table 3 and 4) for all three GAN variants with the only difference of optimizer choice. All GAN variants are able to generate data with mean value $\mu = 4$ as demonstrated in Figure 40.
The Wasserstein GAN with weight clipping and RMSprop optimizer performs best regarding the mean value evaluation criterion, followed by its improved version with gradient penalty (and ADAM optimizer) and lastly the vanilla GAN (both ADAM

or RMSprop).

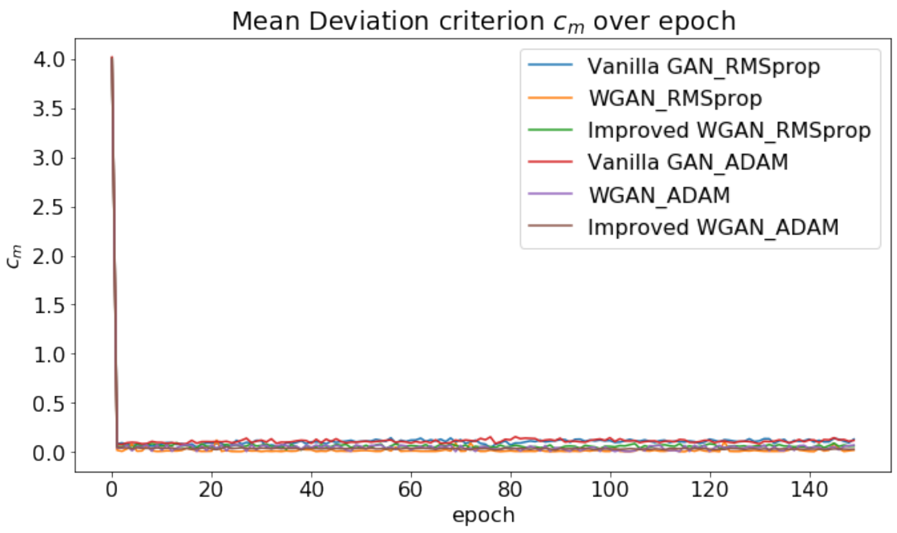


Figure 40: Mean evaluation criterion. Every GAN variant is able to generate samples with mean value $\vec{4}$ very quickly after even one epoch of training.

When analyzing the capability to model the second moment, the corresponding evaluation curves for the Wasserstein GAN with weight clipping are unstable and fluctuate strongly as shown in Figure 41. The Wasserstein GAN with gradient penalty and ADAM optimizer (as suggested by default in Algorithm 3) seems to be most robust regarding the two covariance criteria, generating samples that come from a normal distribution $\mathcal{N}(\mu = 4, \Sigma = I_{50})$.

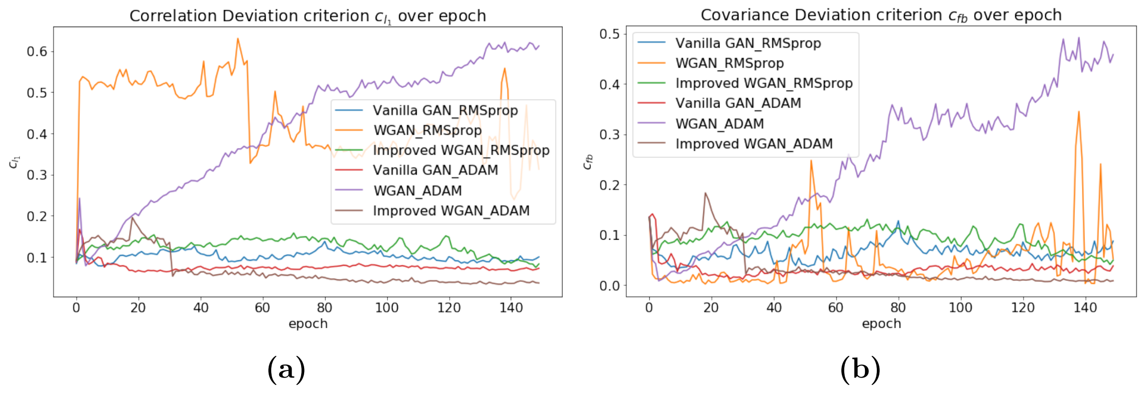


Figure 41: The two covariance evaluation criteria suggest that Improved WGAN with ADAM optimizer (Algorithm 3) is the best method to choose. The generator is able to generate samples which have a mean value of $\vec{4}$ as well as generate samples, where all column features of the samples have a (very) low pairwise correlation.

When analyzing the l_1-criterion, the generator from Improved WGAN with ADAM optimizer seems to produce samples, where its feature columns have low correlation as indicated in Figure 41a. Ideally, the estimated correlation matrix from the batch evaluation data matrix is approximately the identity matrix.
For the frobenius norm criterion, the Improved WGAN with ADAM optimizer performs best as well. Considering the results from this experiment, the Improved

WGAN and ADAM optimizer is chosen as the best algorithm for learning multivariate normal data. Of course, an extensive hyperparameter search can be conducted. Since the goal was to try out different settings and (empirically) show that WGAN with gradient penalty is superior to vanilla GAN and WGAN with weight clipping, no further investigation in hyperparameter tuning was conducted.

Another interesting evaluation step is to select the generator network for the epochs $i = 0, 1, 50, 150$, sample 5000 observations and extract any arbitrary column out of the generated batch data matrix, e.g. the first column. Knowing the data generating process for the multivariate normal distribution with identity matrix as the covariance matrix, we conclude that the joint probability can be factorized as a product of independent univariate Gaussians [Do (2008)]. So when training the GAN, as shown in Figure 40, a distribution shift in the univariate case with increasing epoch towards $\mathcal{N}(\mu = 4, \sigma^2 = 1)$ is expected. To observe this expectation, a kernel density estimation (KDE) on the generated samples was computed. Since the Improved WGAN with ADAM optimizer is learning the true data very fast[20] even after one epoch, we perceive that the univariate Gaussian for the generator model shifts towards the true Gaussian with mean value of four as illustrated in Figure 42.

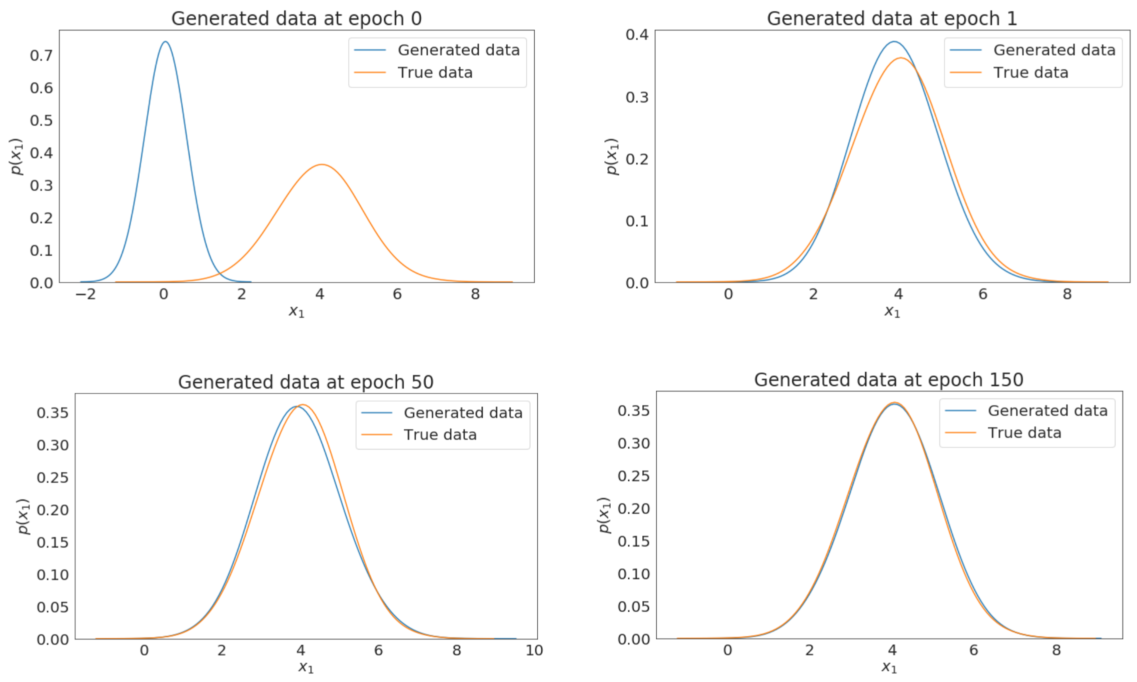


Figure 42: The generator network is learning to produce samples that follow an univariate $\mathcal{N}(\mu = 4\ \sigma^2 = 1)$ even after one epoch of training. For this plot, the first column of the training and generated batch data matrix was selected. The reason why the KDE of generated samples in epoch zero looks Gaussian is because the weights of the generator network are initialized using Gaussian random numbers with zero mean and variance depending on the hidden layer size as stated in the end of Section 2.2.2.1 with the *Xavier Initialization Rule* [Glorot & Bengio (2010)].

[20]The results after one epoch are so strong since the dataset is large with 1 million samples and the training was performed with a batch-size of $m = 256$. In this case, approximately $1000000/256 = 3906.25$ generator updates are executed within one epoch.

4.3 Learning ChEMBL space using CDDD Representations

The first step was to encode the SMILES representation of compounds in the training set $\mathcal{D}$ into their continuous vector representation by using the translation model from Winter et al. (2018) (see Section 2.3.1). We denote the (discrete) space of all (valid) SMILES as $\mathcal{S}$ and assume $\mathcal{D} \subset \mathcal{S}$.
Since the final goal is to perform the *distribution-learning* benchmark from GuacaMol and show the benefits on operating on a continuous-learned data space $\mathcal{C}$, the translation model by Winter et al. (2018) was trained on the provided training set $\mathcal{D}$, to fairly evaluate the GAN in the benchmark.
The encoder network of the translation model translates a SMILES representation into a 512−dimensional vector (which we will denote as `cddd`), where its components are bounded within the range $(-1, 1)$.
Consequently, the `cddd` space holds the property $\mathcal{C} \subseteq (-1, 1)^{512}$.
The decoder network of the translation model translates the `cddd` back into a (canonical) SMILES. This step is required to evaluate the GAN, to verify if the model is able to generate `cddd`-vectors, which after translating back to SMILES representation, are indeed molecules. Recall that the motivation is to generate compounds to enrich chemical compound libraries. Since the `cddd` representations are latent representations, interpreting them is difficult. Therefore, the usage of the decoder network is indispensable to obtain the SMILES representation that is interpretable.

4.3.1 Evaluation Metrics

One main controversy in deep generative modeling research is how to quantitatively evaluate the performance of one model as well as compare it to other generative models. The construction of evaluation metrics in Section (4.2) was straightforward since we knew the data generating process and held the view that the data space was compact and easy enough to learn from.
For evaluating the GANs in this drug discovery use case, we need to think about the target space to evaluate from: should the evaluation metric be a function of the operating space, i.e. the continuous space $\mathcal{C}$, or of the discrete (valid) SMILES space $\mathcal{S}$? Since most evaluation metrics in drug discovery are based on SMILES representations, the decoder network of the translation model is used to retrieve the SMILES representation of generated samples.
Nonetheless, I hold the view that constructing evaluation metrics in the corresponding $\mathcal{C}$ space can improve the GAN due to the consequence of being able to *directly backpropagate* errors of generated `cddd` vectors. This idea can be explained by how GAN training is proceeded: if we had an additional network d_η that evaluates a generated sample $x_g = G(z)$ via $d_\eta((G(z))$, we can use gradient information from this respective term, to update the generator weights by using feedback from the additional network. Note that this is similar as using the feedback from the discrim-

inator network and also included by Winter et al. (2018) in the additional L_2-loss. A possible implementation for an extra network d_η is explained after the description of the first evaluation metric.

For this *distribution-learning* task on a feature space $\mathcal{C}$ with a large training set of approximately 1.2 million samples, it suffices to exploit the capacity of GANs and evaluate on the discrete SMILES space.
The construction of more sophisticated evaluation metrics (especially in the case of focused drug discovery, see Section 4.4) is left for further research.
For evaluating the goodness of the generator network, at the beginning of each training epoch b fake samples were generated. After synthesizing the b fake samples, the decoder network of the translation model was utilized to transform the generated `cddd` samples back to the SMILES representation. The upcoming evaluation criteria are using the SMILES representation as input and are not included into the overall objective of the GAN.
So formally, in every evaluation step we generated $b = 5000$ fake `cddd` samples $\{\tilde{x}^{(i)}\}_{i=1}^{b}$, decoded them back so SMILES $\{\tilde{s}^{(i)}\}_{i=1}^{b}$, where $\tilde{s}^{(i)} = \text{dec}(\tilde{x}^{(i)})$, and computed the evaluation metrics based on the decoded sequences $\tilde{S} := \{\tilde{s}^{(i)}\}_{i=1}^{b}$.
The upcoming presented evaluation metrics are motivated by the GuacaMol benchmark paper [Brown et al. (2019)].

Validity

As stated in Section 2.1.2, the SMILES representation [Weininger (1988)] encodes the topological 2D information of a molecule into a string, based on common chemical bonding rules within a predefined grammar. Hence, when decoding the generated `cddd` back into character-based sequences, it is possible to have invalid expressions that cannot be parsed back to a valid molecule. The python library `RDKit` [Landrum (2006)] was utilized to input the decoded `cddd` samples into a wrapped `RDKit` function $val(\cdot)$, which returns 1, if the sequence can be successfully parsed and 0, otherwise.

$$c_{\text{validity}}(\tilde{S}) := \frac{1}{b}\sum_{i=1}^{b} val(\tilde{s}^{(i)}), \tag{65}$$

where we expect this measure to increase and converge towards one over the training of the generative adversarial network.
In order to guide the generator network G to synthesize `cddd` samples, which after decoding back to strings are indeed valid SMILES, a loss for invalid SMILES can be introduced. One could include an additional *validation network* d_η whose task it is, to discriminate whether a `cddd` $= G(z)$ is valid in terms of the SMILES grammar. The loss function could be binary cross-entropy loss. By backpropagating errors for invalid `cddd=G(z)` samples, the sampling process for the generator network

G would be adjusted such that in the upcoming sampling process the generator network produces samples that are more valid. This network d_η however, has to be pre-trained on valid and invalid `cddd` samples which can be problematic in our case, but is still possible. If the validation network d_η was included in the training process as an additional loss, it would be beneficial to weight the loss with a small parameter $\beta > 0$ in order to confine the effect of the validation network when doing backpropagation. The reason for confining the feedback of d_η is to reduce possible bias due to the fact that d_η is also a predictive model.
The SMILES validation loss w.r.t. one batch $B = \{G(z^{(i)})\}_{i=1}^b$ could be defined as

$$\mathcal{L}_{\text{sv}} = -\frac{1}{b}\sum_{i=1}^{b}\left[v^{(i)}\log(d_\eta(G(z^{(i)}))) + (1 - v^{(i)})\log(1 - d_\eta(G(z^{(i)})))\right], \tag{66}$$

where $v^{(i)}$ is the validity label computed earlier with the `RDKit` function in equation (65). If this loss was included, it would be indispensable to initially pretrain the validation network for n_{sv} epochs, e.g. 50 epochs. After d_η has learned to discriminate between invalid and valid `cddd` samples by minimizing equation (66) w.r.t. η, the loss in equation (66) is used to update parameters of the generator network, when the evaluation step is proceeded with the beginning of epoch 51. Hence, the gradient updates w.r.t. the validation and generator networks are performed with

$$\eta \leftarrow \eta - \zeta\nabla_\eta\mathcal{L}_{\text{sv}}, \quad \text{for epochs zero to } n_{\text{sv}}, \tag{67}$$

$$\theta_g \leftarrow \theta_g - \beta\nabla_{\theta_g}\mathcal{L}_{\text{sv}}, \quad \text{for epochs } (n_{\text{sv}}+1) \text{ to } n_{\text{epochs}}, \tag{68}$$

where $\zeta > 0$ is the learning rate for updating the parameters of the validation network. This additional loss however, was not included and tested due to time constraints and scope of this thesis.

Uniqueness

A generative model would not make sense if it always generates the same sample. Especially in drug discovery, it is desired to explore chemical space and generate new (unseen) compounds. In the procedure of the uniqueness metric, duplicates in $\{\tilde{s}^{(i)}\}_{i=1}^b$ are removed and the number of remaining sequences obtained. Let that number be n_u, where $n_u \leq b$. The uniqueness metric is then computed via

$$c_{\text{uniqueness}}(\tilde{S}) := \frac{n_u}{b}, \tag{69}$$

where we expect this measure to increase and converge towards one, meaning that the generator network is able to produce unique samples. Note that this metric does not include the validity check. This means that within the set of unique samples lies the possibility that also invalid decoded sequences are present.

Novelty

The generative model should produce a variety of samples and not follow the problem of mode collapse by only generating duplicate observations. Apart from achieving a high uniqueness score, it is also desired that the generated SMILES are indeed novel and not present in the training reference set. Let n_n denote the number of novel molecules which are not present in the training set. This number is calculated by iterating all samples from $\tilde{S}$ through the training set and check whether it is present (returning 0) or not (returning 1). Hence, the novelty measure is calculated as

$$c_{\text{novelty}}(\tilde{S}) := \frac{n_n}{b}, \tag{70}$$

where we expect this measure to increase and converge towards one. Like the uniqueness metric, this evaluation metric does not include the validity check.

Fréchet Chemnet Distance (FCD)

The Fréchet Chemnet Distance (FCD) by Preuer et al. (2018) measures how close the distribution of generated compounds p_g is to the distribution of compounds in a training set p_r, by considering chemical and biological information. To obtain a numerical representation of a molecule, which comprises valuable chemical and biological information, the hidden layer activations from a third-party network called *ChemNet*[21] are extracted. Since *ChemNet* was trained to predict biological activity, Preuer et al. (2018) claim that the activations of the penultimate hidden layer comprise biological and chemical properties and are reasonable numerical descriptors (because the input layer consists of chemical compounds and the output layer is the layer for predicting biological activity).
Precisely, the mean m and covariance C (see equation (58) and (60)) of those activations for the generated set $\tilde{S} = \{\tilde{s}^{(i)}\}_{i=1}^{b}$ and a randomized reference (training) set $\mathcal{D}_s \subset \mathcal{D}$, with $|\mathcal{D}_s| = b$, are required to obtain the FCD[22] measure.

$$c_{\text{FCD}}(\tilde{S}, \mathcal{D}_s) = ||m_r - m_g||_2^2 + \text{Tr}(C_r + C_g - 2(C_r C_g)^{\frac{1}{2}}), \tag{71}$$

where low FCD values indicate similar molecular distributions. During the training of the generative model, we desire this measure to decrease. For computing the FCD score, the open-source python library *FCD*[23] distributed on github was used.

[21]This network was trained to predict biological activity on a dataset of about 6000 assays available in three major drug discovery databases (ChEMBL [Mendez et al. (2018)], ZINC [Irwin & Shoichet (2005)], PubChem [Wang et al. (2017)]).

[22]The FCD measure is the Wasserstein-2 distance (equation (79)) between two multivariate normal distributions. When computing the estimated sample means and covariances of activations for generated set (m_g, C_g) and training reference set (m_r, C_r), Preuer et al. (2018) assume that those activations come from a multivariate normal distribution. Hence the Wasserstein-2 distance for two multivariate normal distributions has a closed-form solution stated in equation (71).

[23]https://github.com/bioinf-jku/FCD

4.3.2 Results

The initial motivation for the first experiment was to compare the three GAN variants described in Section 2.4 with respect to the earlier defined evaluation metrics. For that reason, the three GAN variants were trained with the same network architecture but different optimizers. For training vanilla GAN, RMSProp and ADAM optimizers were used, to see differences in evaluation metrics between the two optimizers. For the WGAN with weight clipping, RMSprop optimizer was used. The WGAN with gradient penalty used the ADAM optimizer. The learning rates for discriminator and generator were set to $\alpha_d = 0.0009$ and $\alpha_g = 0.0003$ for all three GAN algorithms. The noise distribution Z was selected as the $100-$dimensional Gaussian with zero mean and unit covariance. The baseline network architectures for comparing the three GAN variants are presented in Table 5 and 6.

Table 5: Illustration of the generator network architecture. It consists of two fully connected hidden layers with batch normalization and ELU activation [equation (6), Clevert et al. (2015)]. The output layer contains the tanh activation (equation (4)) since the `cddd` space is bounded within $(-1, 1)^{512}$.

Name	Type	Input size	Output size
input	input: $z \sim \mathcal{N}(0,1)$	100	-
FC1	linear	100	256
	batch normalization	256	256
	ELU	256	256
FC2	linear	256	512
	batch normalization	512	512
	ELU	512	512
output	linear	512	512
	batch normalization	512	512
	tanh	512	512

Table 6: Illustration of the discriminator/critic network architecture. Each hidden layer consists of leaky ReLu activation with no batch normalization. Depending on the algorithm, sigmoid activation (equation (3)) is deployed in the output layer. This only holds for the vanilla GAN Algorithm 1.

Name	Type	Input size	Output size
input	input: $x \simeq \mathcal{C}$	512	-
FC1	linear	512	256
	leaky ReLU	256	256
FC2	linear	256	128
	leaky ReLU	128	128
output	linear	128	1

The GANs with baseline generator and discriminator networks were trained for $n_{\text{epochs}} = 200$ epochs with a batch-size of $m = 256$. The results illustrated in Figure 43 show that WGAN with gradient penalty performs best in terms of FCD measure, followed by WGAN with weight clipping and the two vanilla GANs. Interestingly, all three algorithms with the selected baseline architecture for generator and discriminator network are producing on average around 94% valid compounds/molecules with almost perfect uniqueness score and high novelty measure. It seems that vanilla GAN, optimized with RMSProp (blue line) is exploring the `cddd` space the most, which can be seen in the novelty plot in Figure 43. Out of 5000 generated samples almost every sample is indeed novel. On average around 94% valid SMILES are generated. However, the molecules that are generated from the two vanilla GANs are not firmly ChEMBL-like as shown in the FCD plot.

Since the goal is to learn a generative adversarial network that can generate ChEMBL-like data as provided in the training dataset, the WGAN with gradient penalty Algorithm 3 was selected for the upcoming experiments.

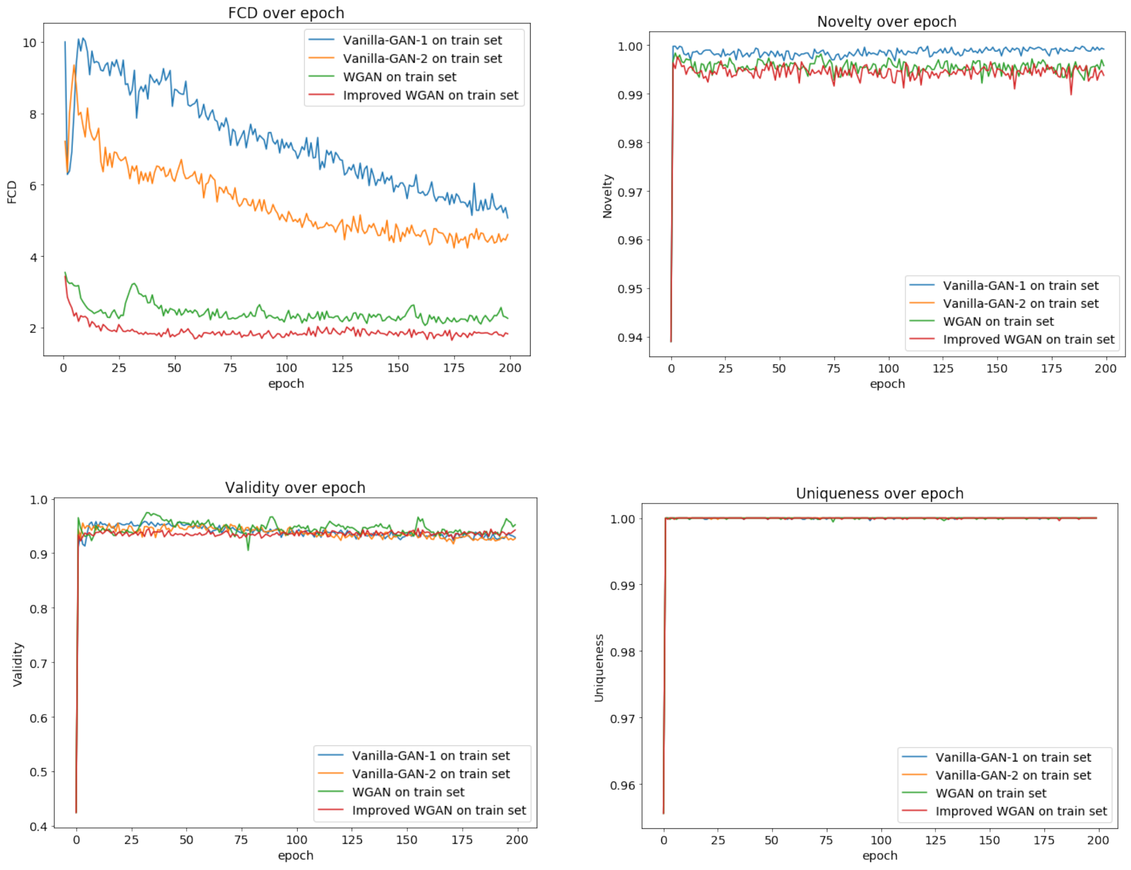


Figure 43: Evaluation metrics for baseline models. The Improved WGAN with gradient penalty is producing samples which are most likely coming from ChEMBL space, as shown in the FCD plot. All GAN variants peform well in validity, novelty and uniqueness.

The evaluation metrics in Figure 43 are based on the models trained on the provided training set of GuacaMol with approximately 1.2 million samples (see Section 3). Since GANs are known to be trained for many epochs on large datasets in general, the problem of overfitting the training data might occur. In order to empirically show that the Improved WGAN model is superior to the other three baseline models, the FCD metric was computed with respect to the provided test and evaluation set from GuacaMol. Those two sets together consist of 318 274 samples. The novelty metric was not computed with respect to the test and validation set because novelty is not an indicator if a model has been overfitted. The novelty metric could act as an synthetic-indicator for generated samples, which are existent in the unseen set. If the generated samples were in the unseen sets, the GAN would be a good model because it can generate compounds that really exist.
Recall that the FCD metric considers the ‘biochemial’ distribution of generated samples compared to the training reference set. Since the goal is to obtain a GAN model that is capable of sampling ChEMBL-like data, the FCD metric was chosen as deciding metric. A smaller validity for a GAN model is in practice neither bad

nor expensive. Remind that one advantage of GANs is that sampling can be conducted efficiently, since sampling from Z is computational cheap and feasible. So if generated samples are not valid, the sampling process can be executed so long, until the desired number of valid samples is achieved. For a generative model, it would be advantageous if it can generate valid samples without errors. One way to improve the validity metric could be to include the additional validation loss from equation (66). Since the validity reaches on average about 94%, the validation network from equation (66) was not included into the training process. The FCD metric plot for the baseline models with respect to the test and evaluation sets from GuacaMol is shown in Figure 44.

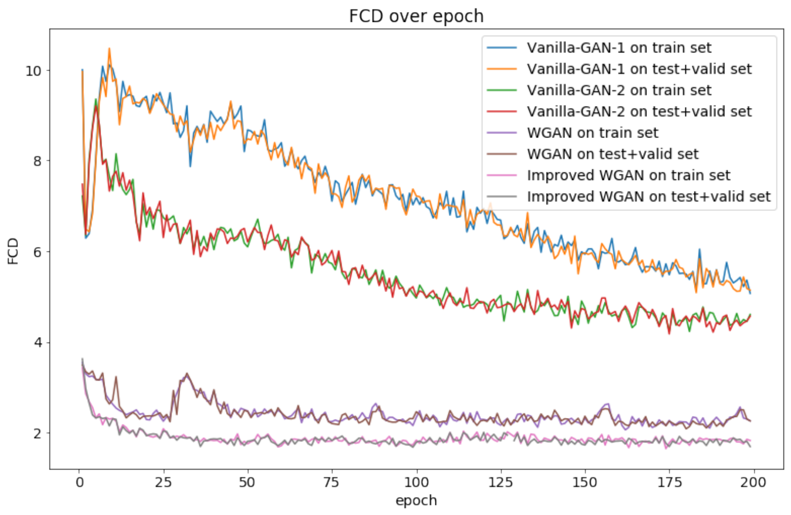


Figure 44: FCD metric for baseline models regarding training and test+validation set. The FCD metric as deciding evaluation metric is not overfitted by the Improved WGAN model as shown in the test+validation set curves.

As the FCD plot in Figure 44 shows the superiority of Improved WGAN, the final algorithm for the next experiments was Algorithm 3.
In order to explore the capacities of the Improved WGAN Algorithm 3, different network architectures were tested in a second experiment and compared w.r.t. the evaluation metrics. Especially the improvement in the FCD metric for deeper generator networks was questioned. Remember that GANs are implicitly modeling the true data distribution through a deterministic generator network. If the generator network gets deeper, it is reasonable to hypothesize that the underlying true data distribution can be approximated better.
Furthermore, it was interesting to investigate, how the complexity of the critic affects the generated samples. Recall that the critic network D is crucial for estimating an approximation of the Wasserstein-1 distance as stated in equation (53). As claimed by Arjovsky et al. (2017) and Gulrajani et al. (2017), *useful* gradient information will be passed to the generator from the critic, the better the critic network approximates the Wasserstein-1 distance. For the second experiment, several GAN models following the Improved WGAN Algorithm 3 were trained for $n_{\text{epochs}} = 50$ epochs

with different optimizers and learning rates. The results and plots are illustrated in Appendix C.1. After conducting the parameter experiment, it turned out that batch normalization in the generator network and neither batch nor layer normalization should be included in the critic network. Additionally, including leaky ReLU as an activation function in both, generator and critic network seems to stabilize the training and generate novel molecules after evaluating the FCD metric.
In the parameter experiments, hidden layer sizes from one up to six were tested. The number of neurons in each hidden layer was set arbitrarily to the exponential with base 2. The final setting is listed below in Table 7 and 8.

Table 7: Illustration of the selected generator network architecture.

Name	Type	Input size	Output size
input	input: $z \sim \mathcal{N}(0,1)$	100	-
FC1	linear	100	128
	batch normalization	128	128
	leaky ReLU	128	128
FC2	linear	128	256
	batch normalization	256	256
	leaky ReLU	256	256
FC3	linear	256	512
	batch normalization	512	512
	leaky ReLU	512	1024
FC4	linear	1024	1024
	batch normalization	1024	1024
	Leaky ReLU	1024	1024
FC5	linear	1024	512
	batch normalization	512	512
	Leaky ReLU	512	512
output	linear	512	512
	batch normalization	512	512
	tanh	512	512

Table 8: Illustration of the selected critic network architecture.

Name	Type	Input size	Output size
input	input: $x \simeq \mathcal{C}$	512	-
FC1	linear	512	512
	leaky ReLU	512	512
FC2	linear	512	256
	leaky ReLU	256	256
FC3	linear	256	256
	leaky ReLU	256	256
FC4	linear	256	128
	leaky ReLU	128	128
FC5	linear	128	128
	leaky ReLU	128	128
output	linear	128	1

The generator and critic networks were trained using RMSprop optimizer with learning rates of $\alpha_g = 0.0002$ and $\alpha_d = 0.0006$. The learning rates were multiplied by 0.99 in each increasing training epoch in order to decrease them and stabilize training. The batch-size was set to $m = 256$ and the critic was updated for $d_{\text{iters}} = 3$ steps followed by one generator step. The coefficient for the gradient penalty term was set to $\lambda = 10$ as stated in the defaults. The training was executed five times with different seeds for $n_{\text{epochs}} = 200$ epochs. Additionally, the baseline Improved WGAN model with architectures from Table 5 and 6 was trained five times with different seeds to compare the performance to the best model.
In order to check that both models are not overfitting the training data, the FCD metric was computed for baseline and best models with respect to the test and validation set from GuacaMol (see Section 3). The evaluation metrics measured at epochs 50, 100, 150 and 200 for both models are listed in Appendix C.2.

Figure 45 shows the four evaluation metrics for the selected best model compared to the best baseline model. The mean value (solid line) for each evaluation criterion was plotted with $+/-2\sigma$ deviation (shaded area) for each epoch, since the experiment was conducted five times with different seeds.
The best model is able to generate samples that are very ChEMBL-like because it outperforms the baseline model in terms of FCD measure. Increasing the complexity of the generator network by adding more layers, helps to generate samples which are very likely to come from the training ChEMBL dataset as shown in the first plot in Figure 45.

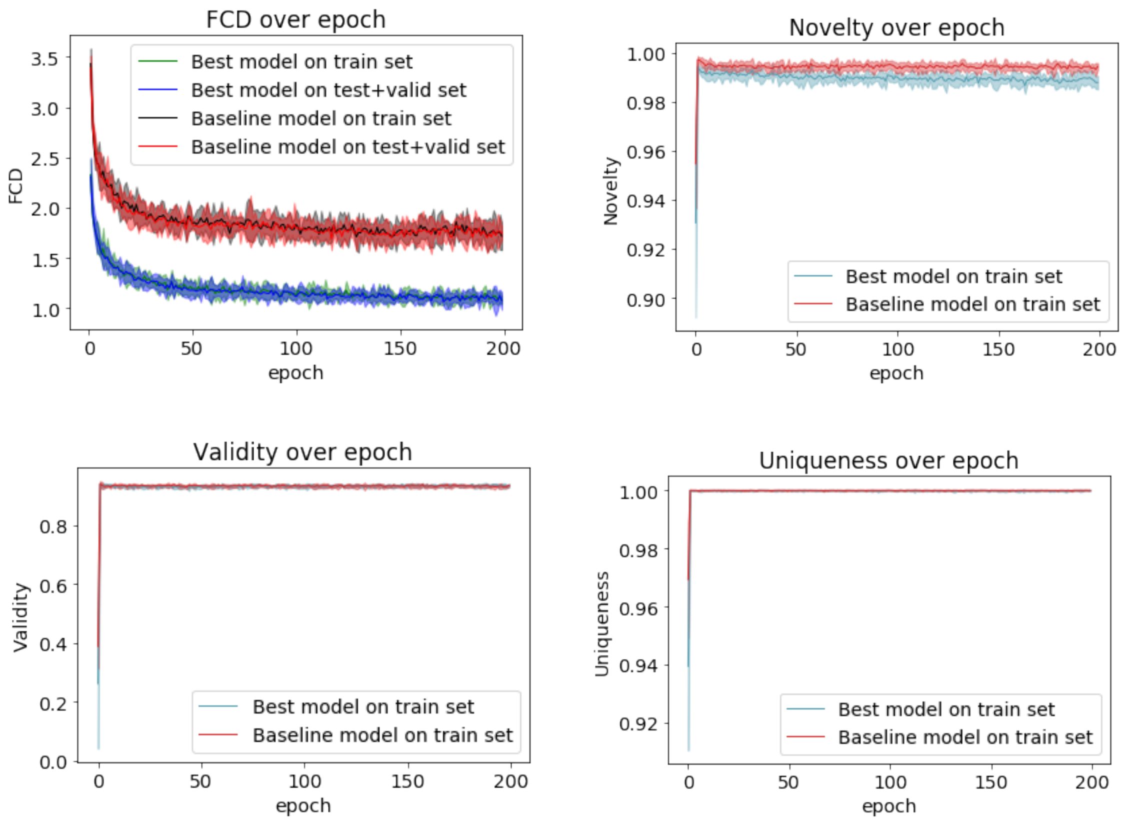
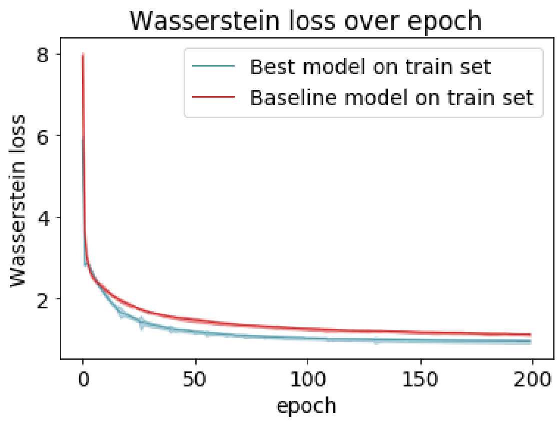
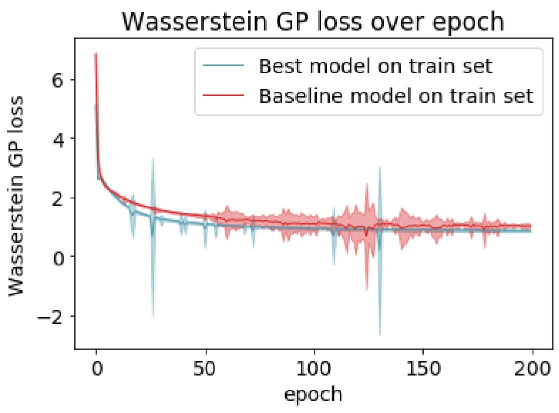


Figure 45: Evaluation metrics for the best model compared to the best baseline model.

Only in terms of novelty, the baseline model is (slightly) better than the best model. This finding is indirectly coupled with the FCD measure. Remember that the FCD score measures how close a generated set of samples is to the training reference set (ChEMBL). Since the baseline model is generating (slightly) more novel molecules, the FCD score is higher. This does not necessarily mean that the generated molecules from the baseline model are not 'good' molecules. It merely means that they are not similar to molecules from ChEMBL space. Since the objective is to obtain a GAN that can approximate the (training) reference distribution, we will hold to the best model with network architectures described in Table 7 and 8. Figure 46 shows the Wasserstein loss from equation (53) and Wasserstein loss with gradient penalty from equation (57) in order to investigate the $1-$Lipschitz continuity and judge the sample quality of the generator network.



(a) Wasserstein loss.



(b) Wasserstein loss with gradient penalty.

Figure 46: Wasserstein losses for the baseline and best model. The Wasserstein losses are multiplied with (-1) in order to see a decreasing function. Recall that the Wasserstein GAN attempts to solve a minimax optimization problem. The ideal value to be achieved is zero, i.e. when the generator distribution equals the real data distribution.

Note that Arjovsky et al. (2017) and Gulrajani et al. (2017) argued that the Wasserstein loss correlates well with the sample quality of the generator network. This means that if the (negative) Wasserstein loss decreases, the generated samples should have good quality. This claim is confirmed when comparing Figure 46a and the FCD evaluation plot in Figure 45. As the Wasserstein loss decreases, the FCD measure decreases as well. This indicates that the samples generated by the generator network are ChEMBL-like. When comparing Figure 46a and 46b the only difference in the vertical magnitude is the gradient penalty term $GP = \lambda \mathbb{E}_{\hat{x} \sim p_{\hat{x}}} \left[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2 \right]$ from equation (54), where $\hat{x}$ is a linear interpolation between real data sample x_r and generated sample $x_g = G(z)$. This term was introduced by Gulrajani et al. (2017) to enforce the $1-$Lipschitz continuity of the critic network. Remark that the Wasserstein- and Wasserstein GP losses are almost the same with increasing epoch. This implies that the gradient penalty term is close to zero, concluding that the critic network is indeed a $1-$Lipschitz function, where its gradient has norm one. When comparing the Wasserstein GP loss from the baseline model with the best model, we observe that the baseline model has higher variance, e.g. around epoch 125. This indicates that the gradient penalty term is affecting the overall Wasserstein GP loss and the baseline critic is less stable regarding the $1-$Lipschitz continuity. The reason for this might be that the critic does not have a gradient norm of one for some of the linear interpolated points $\hat{x}$, concluding that the generated points $x_g = G(z)$ are not resembling the true data distribution. This thought is also indicated by higher FCD evaluation scores for the baseline model. Hence, Proposition 1 is violated and leads to larger gradient penalty terms. Increasing the complexity of the generator network by adding more hidden layers, stabilized the training and led to better samples with lower FCD scores as shown by the best model (called *cdddGAN*), which also has a smaller Wasserstein loss than the baseline model.

4.3.3 Druglikeness of Generated Molecules

In order to additionally verify that *cdddGAN* has learned the ChEMBL training data, a set of 10 000 valid and unique molecules was generated. This generated set was used to compute the `QED` score, which is a Quantitative Estimate of Druglikeness (QED) of a molecule and was introduced by Bickerton et al. (2012).
The 'chemical beauty of drugs' measured as `QED` score is a geometric mean of eight *individual desirability functions*. Those eight individual desirability functions each measure the chemical beauty of one physicochemical property such as molecular weight, octanol-water partition coefficient (logP) or the number of hydrogen bond donors (HBD) and aromatic rings compared to a fixed reference set of 'orally absorbed approved drugs' [Bickerton et al. (2012)].
To compute the `QED` score for the generated set, the python library `RDKit` [Landrum (2006)] was utilized. In order to compare the generated set with the true ChEMBL reference training set, the `QED` score for a random subset containing 10 000 SMILES from the training set was computed. To show that the 10 000 generated valid and unique SMILES are ChEMBL-like, a histogram with absolute count values for the `QED` scores from generated set and training subset was computed and a plot generated as shown in Figure 47.

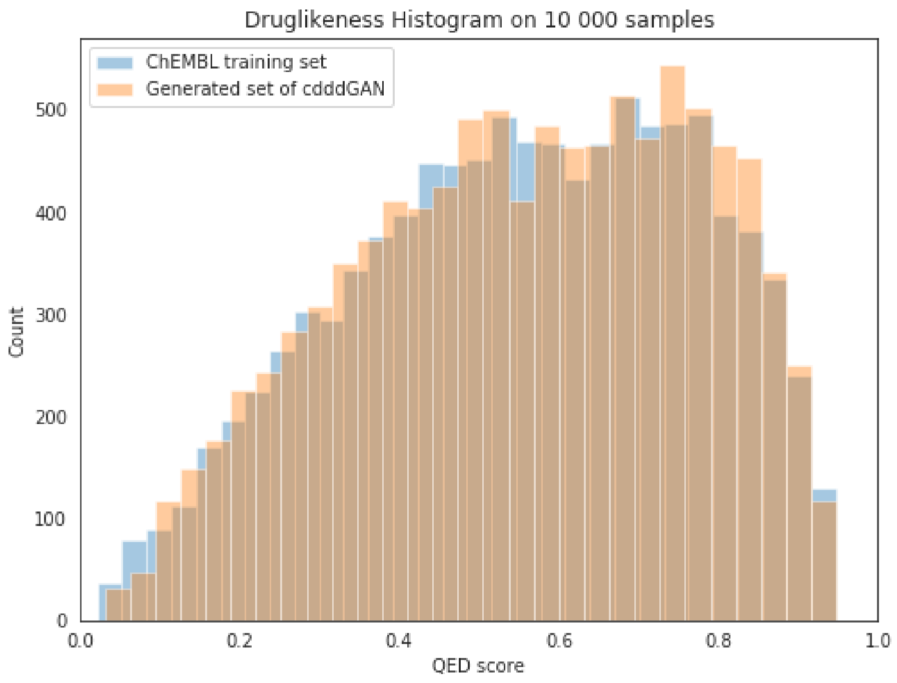


Figure 47: The `QED` absolute distribution of generated molecules and training reference set. The shape of both histograms looks equal, indicating that the generated molecules follow the `QED` distribution from the training reference set.

Figure 48 shows the 2D topological graphs of six generated compounds of *cdddGAN* with their respective SMILES representation and `QED` score. All six compounds are not present in the training set.

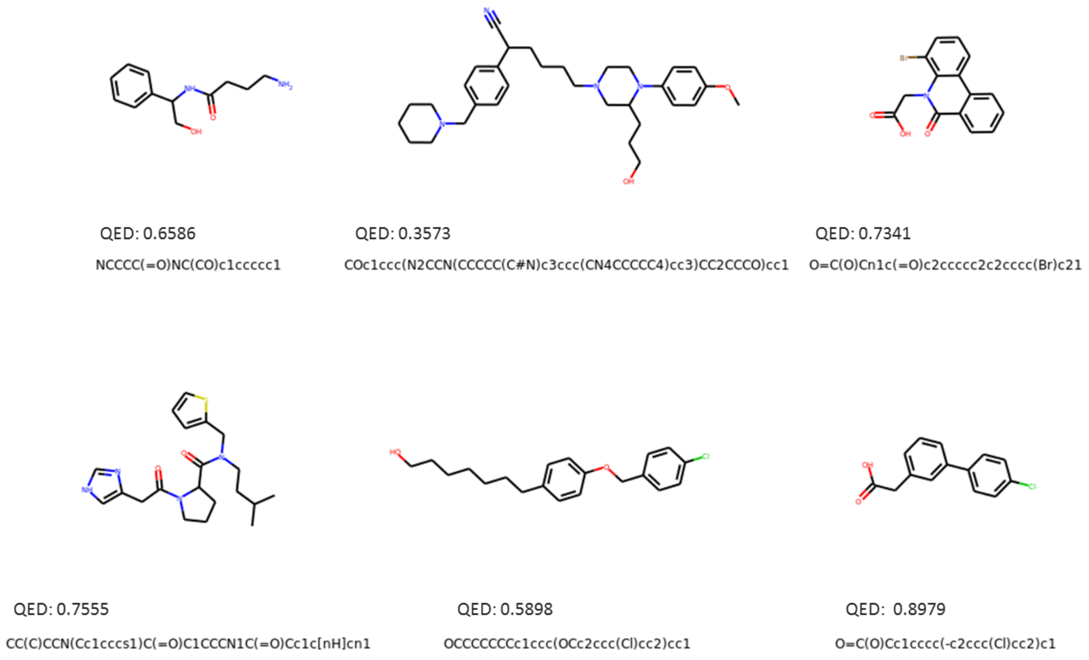

QED: 0.6586
NCCCC(=O)NC(CO)c1ccccc1
QED: 0.3573
COc1ccc(N2CCN(CCCCC(C#N)c3ccc(CN4CCCCC4)cc3)CC2CCCO)cc1
QED: 0.7341
O=C(O)Cn1c(=O)c2ccccc2c2cccc(Br)c21
QED: 0.7555
CC(C)CCN(Cc1cccs1)C(=O)C1CCCN1C(=O)Cc1c[nH]cn1
QED: 0.5898
OCCCCCCCc1ccc(OCc2ccc(Cl)cc2)cc1
QED: 0.8979
O=C(O)Cc1cccc(-c2ccc(Cl)cc2)c1

Figure 48: Generated sampels from *cdddGAN* with their QED values.

4.3.4 GuacaMol: Distribution-Learning Benchmark

The GuacaMol distribution-learning benchmark [Brown et al. (2019)] consists of five evaluation benchmarks to evaluate a generative model regarding the criteria of

1. Validity.
2. Uniqueness.
3. Novelty.
4. Fréchet Chemnet Distance (FCD).
5. Kullback-Leibler divergence (KLD).

Every single benchmark is performed five times and the results are saved. Each single benchmark outputs a score between zero and one, where one is the best achievable value and zero the worst. The final score for each benchmark is the average over the saved results. In each benchmark 10 000 samples are generated. Except for the validity benchmark, the sampling is conducted until 10 000 valid molecules are generated followed by the computation of the final benchmark result. The benchmark metrics are computed similar to the evaluation metrics in Section 4.3.1. For the FCD measure, a normalization into range (0,1) is done through

$$\mathrm{FCD} = \exp\left(-0.2 c_{\mathrm{FCD}}\right), \tag{72}$$

where c_{FCD} is from equation (71).
The KL divergence as defined in equation (36) measures how well a probability distribution p_g approximates another distribution p_r. 'For the KL divergence benchmark, the probability distributions of a variety of physicochemical descriptors for the [reference] training set and a set of generated molecules are compared, and the corresponding KL divergences are calculated' [Brown et al. (2019)]. This benchmark differs from the FCD benchmark in a sense that the FCD captures an overall numerical representation of chemical and biological properties of a molecule, whereas the KL divergence benchmark has selected specified physicochemical properties.
In total there are $k = 9$ numerical physicochemical descriptors that are calculated using the `RDKit` python library [Landrum (2006)]. Example physicochemical descriptors are `NumAromaticRings, MolLogP, NumHAcceptors`. Since some physicochemicals are discrete, e.g. `NumAromaticRings` and `NumHAcceptors`, a histogram is estimated within the GuacaMol library to compute the respective empirical distributions. For continuous descriptors such as `MolLogP`, a kernel density estimation is performed to compute the probability density function.
Once the histograms or kernel density estimations for a specific physicochemical descriptor $i = 1, .., k$ are attained, the KL divergence $\varphi_{\text{KL},i}$ between training reference set and generated molecule set can be computed.

The final KL divergence benchmark metric is a normalized value that ranges between zero and one, where one is the best achievable value.

$$\text{KLD} = \frac{1}{k}\sum_{i=1}^{k} \exp\left(-\varphi_{\text{KL},i}\right) \tag{73}$$

Note that this metric is similar to the `QED` score mentioned in Section 4.3.3 in terms of combining different physicochemical properties into a final score, which here is a normalized KL divergence score.

Results

The results of the GuacaMol distribution-learning benchmark with comparison to the generative models from the leaderboard (`https://benevolent.ai/guacamol`, accessed: 4th November 2019) are displayed in Table 9.

benchmark	**AAE**	**Graph MCTS**	**Random Sampler**	**SMILES LSTM**	**VAE**	**ORGAN**	**cdddGAN**
Validity	0.822	1.000	1.000	0.959	0.870	0.379	0.934
Uniqueness	1.000	1.000	0.997	1.000	0.999	0.841	1.000
Novelty	0.998	0.994	0.000	0.912	0.974	0.686	0.985
KLD	0.886	0.522	0.998	0.991	0.982	0.267	0.972
FCD	0.529	0.015	0.929	0.913	0.863	0.000	0.851

Table 9: Results of the GuacaMol distribution-learning benchmark. The values highlighted in red font are from our proposed *cdddGAN* model.

The *Random Sampler* in Table 9 is a practical baseline model for comparison as it selects random samples from the original training set. For that reason, the novelty

benchmark of the Random Sampler is zero. Additionally, the FCD and KLD benchmark scores of the Random Sampler are comparison values for all other generative models, as those metrics are computed based on the ground truth ChEMBL data.
The *Graph MCTS* [Jensen (2019)] is a genetic algorithm that works on the molecular graph (see Figure 8) of compounds and is identified by a high degree of validity, uniqueness and novelty. Nevertheless, this model is not able to reproduce the chemical properties of the true ChEMBL reference set as demonstrated in the poor results in the KLD and FCD benchmark scores.
The *SMILES LSTM* [H. S. Segler et al. (2017)] has overall the best evaluation scores and performs best in the KLD and FCD benchmarks. Recall that the SMILES LSTM model is a next-character prediction model (see Section 2.2.6.2) and uses the string-based SMILES representation of compounds as input, similar to *AAE, VAE* and *ORGAN*. Since the SMILES LSTM model obtains a validity benchmark score of 95.90%, it has not perfectly learned the SMILES grammar. Nonetheless, it is able to generate novel and diverse compounds that are ChEMBL-like as shown in the high FCD and KLD scores.
The *VAE* by Gómez-Bombarelli et al. (2016) is comparable to our proposed method in a sense that the authors trained a variational autoencoder similar to the translation model by Winter et al. (2018). In their VAE, the bottleneck-layer, which comprises the latent (hidden) representation of compounds, follows a multivariate normal distribution[24]. Our proposed method differs that the translation model was trained to learn a meaningful continuous representation of compounds in an initial step. Based on these continuous representations, we trained a GAN to learn the feature space of the the encoded compounds. Our *cdddGAN* model beats the VAE in terms of validity, uniqueness and novelty. However, our model does not perform as good as the VAE regarding the FCD and KLD scores that characterize 'distribution-properties' towards the ChEMBL database.
Combining the idea of variational autoencoder and generative adversarial networks, the *AAE* [Kadurin et al. (2017)] obtains outstanding results in uniqueness and novelty. In terms of KLD and FCD scores, the AAE is not able to generate samples following certain chemical properties of the reference set.
The *ORGAN* [Guimaraes et al. (2017); Sanchez et al. (2017)] model is performing at worst in all benchmarks. Remember that the ORGAN model is also a GAN but works with the SMILES representation as input.
Our *cdddGAN* model outperforms ORGAN in all benchmarks that indicates, when using GANs to learn a distribution over a training set, it is better achieved when the GAN is learning on continuous representations, e.g. `cddd`, than discrete representations, e.g. SMILES.

[24]So the generating process in their case is to sample the latent hidden representation and then use the decoder network to map back to SMILES representation.

4.4 Optimizing Molecules in Learned ChEMBL Space

Since this thesis is titled with *'De novo drug design in continuous space'*, this Section describes the training of a generative model that is able to synthesize new molecules that satisfy certain physicochemical properties. As mentioned in Section 1.1, most generative models in drug discovery are trained on the discrete SMILES representation of compounds. It is worth remembering that the novel approach conducted earlier in Section 4.3 was to train a GAN on an unsupervised-learned continuous data space. During research of this thesis, Prykhodko et al. (2019) published their generative model called *latentGAN* that is also a generative adversarial network trained on a latent representation of molecules. This latent representation is learned in an unsupervised fashion through a *heteroencoder* similar to the model by Winter et al. (2018).

The first goal of this application was to learn the distribution of a large dataset of chemical compounds using GANs. This was successfully achieved as demonstrated in the results from the GuacaMol distribution-learning benchmark in Table 9.
The next goal after successfully learning a large chemical space was to train a GAN that is able to sample novel molecules, satisfying certain physicochemical conditions. Due to the scope of this thesis, we aimed towards a **single-objective optimization** for the `QED`[25] druglikeness score, which was explained earlier in Section 4.3.3.
We followed the procedure from H. S. Segler et al. (2017), who first trained a prior generative model that can generate molecules from a large training set, and subsequently conducted the optimization step for specific biochemical conditions of interest, such as bioactivity on a defined protein. The optimization was achieved via transfer-learning, i.e. training the prior generative model on a subset of data. This subset of data was filtered from the initial training set and satisfied the desired biochemical criterion. By learning on this subset of data, the pre-trained generative model was able to narrow/bias the already learned distribution towards a new distribution of molecules that is active against a defined protein.
For this application of optimizing `QED` score, a filter was applied on the provided training set from GuacaMol to extract only SMILES representations that have a `QED` score higher than 0.9.
Applying this filter led to a new training subset of 31 778 samples.

To visualize and show that molecules with higher druglikeness scores *might* lie in a manifold within the `cddd` space, a principal component analysis (PCA) for a random subset of the training set (consisting of 1.2 million observations) and filtered

[25]To be precise, the `QED` score is an aggregation of specific desired physicochemical descriptors derived by Bickerton et al. (2012). This means that this metric actually comprises multi-objective criteria. Since the GAN is only taking into account the `QED` value for a molecule, we will still call it single-objective optimization.

set (consisting of around 31 thousand observations) was conducted. The number of components for the PCA was set to 50. Figure 49 shows the first two principal components, which explain 10.77% of the total variance, for the training and filtered dataset.

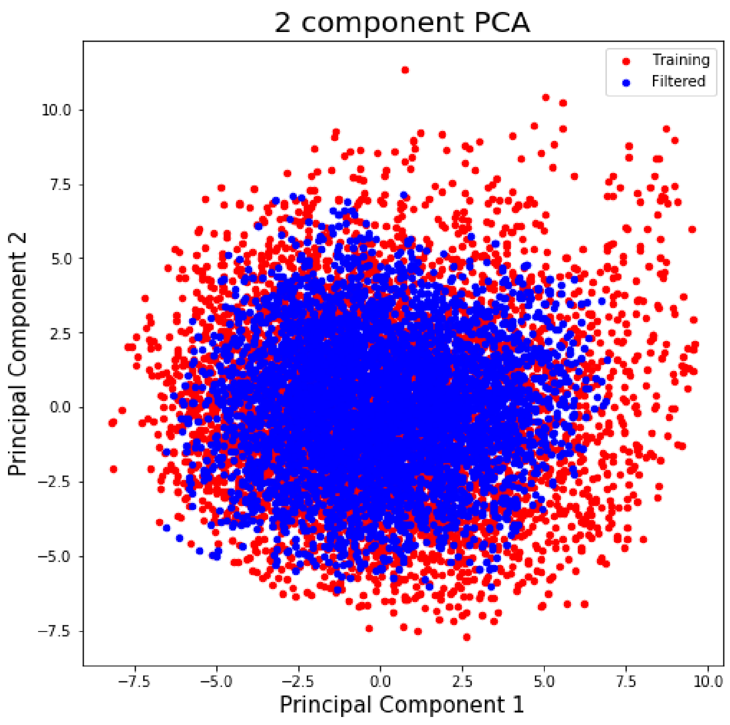


Figure 49: Principal component analysis on a random subset of 5 000 samples each from the full training set and filtered `QED` set. The PCA was conducted on the 512−dimensional `cddd` representations from the random subsets. The blue region that visualizes the 'filtered' dataset indicates that molecules with `QED` > 0.90 lie there.

For optimizing the `QED` score, we applied transfer-learning similar to H. S. Segler et al. (2017), by optimizing the best pre-trained GAN model (*cdddGAN*) from the distribution-learning (see Table 7 and 8) on the filtered dataset. In addition to the pure transfer-learning approach, we incorporated another optimization technique by Gupta & Zou (2019), which includes a feedback mechanism called *FeedbackGAN*.

4.4.1 FeedbackGAN

Gupta & Zou (2019) introduced *FeedbackGAN* in the context of generating DNA sequences and optimizing them with regard to certain biochemical properties. Their algorithm incorporates an external *function analyzer* to score generated sequences concerning the properties of interest. One advantage of their method is that the function analyzer does not have to be differentiable. It can be any black-box model and might range from a machine learning algorithm such as support vector machine, random forest, or functionality from `RDKit`, up to the opinion of an expert in biology or chemistry. The task of the function analyzer is to return a score for an input sample. Gupta & Zou (2019) declare that the feedback-loop mechanism consists of two components. 'The first component is the GAN, which generates novel gene

sequences which have not been enriched for any properties. The second component is the analyzer. [...] The GAN and analyzer are linked by the feedback mechanism after an initial number of pre-training epochs so that the generator is producing valid sequences. Once the feedback mechanism starts, [in the beginning] of every epoch a set number of generated sequences are sampled from the generator and input into the [function] analyzer.' [Gupta & Zou (2019)]
After passing the generated samples to the function analyzer, the top n_d most favorable and desirable samples which satisfy the desirability condition, are **inserted** into the training set and the n_d oldest training samples replaced by the newly inserted ones. 'As the feedback process continues, the entire training set of the discriminator is replaced repeatedly by generated [samples] that have received high scores from the [function] analyzer.' [Gupta & Zou (2019)] The workflow of *FeedbackGAN* after the initial pre-training phase is illustrated in Figure 50.

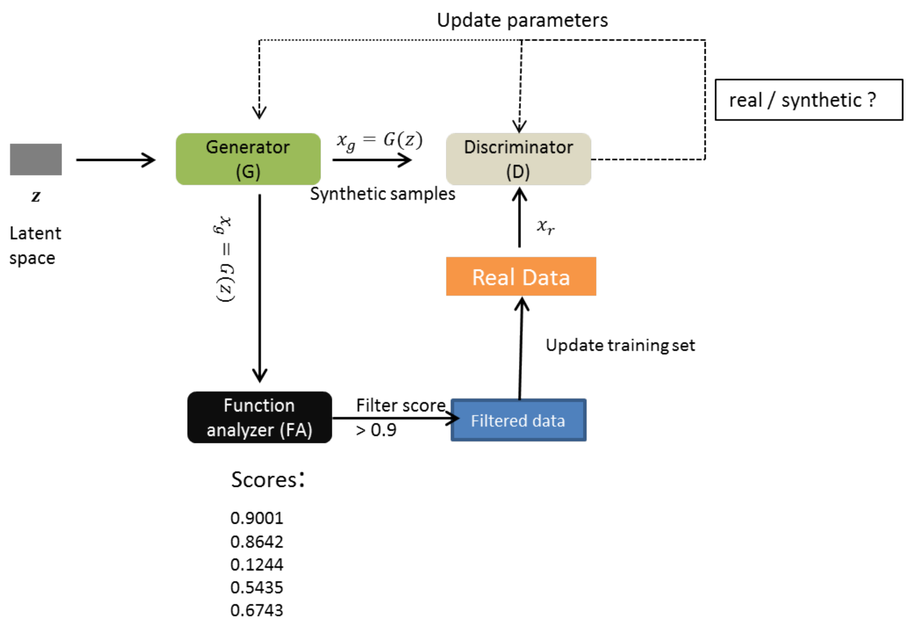


Figure 50: FeedbackGAN workflow.

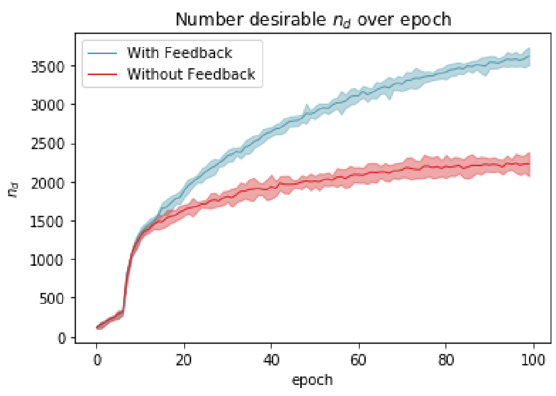
In our case, the function analyzer is the `QED` score function provided by the `RDKit` library. At the beginning of each epoch, $b = 5000$ samples are generated, translated back into SMILES representations, i.e. $\tilde{S} = \{\tilde{s}_i\}_{i=1}^b$, and only valid SMILES are evaluated by the `QED` function analyzer. After applying the function analyzer, those samples whose druglikeness value is above 0.9 are filtered and added to the desirable set $\tilde{S}_d$, if $\tilde{s}_i$ is not present in the current training set. This step prevents the presence of non-unique samples in the training set. Finally, a set $\tilde{S}_d$ that consists of $n_d = |\tilde{S}_d|$ samples is obtained. Once the desirable set $\tilde{S}_d$ is obtained, the n_d oldest samples in the training set are replaced by the n_d samples from the desirable set $\tilde{S}_d$. The GAN is trained according to Algorithm 3 over $n_{\text{epochs}} = 100$ epochs with a batch-size of $m = 128$. The optimizer settings are equal to the optimizer settings from *cdddGAN*.

4.4.2 Results

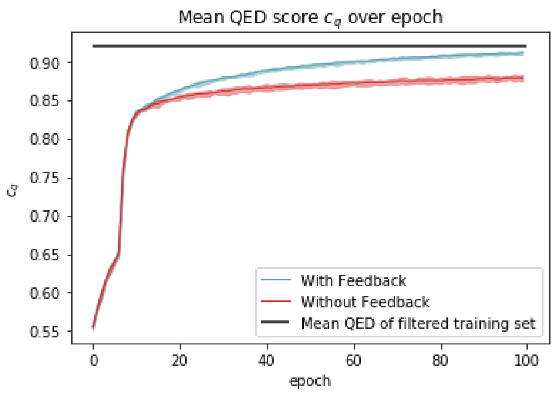
The pre-trained *cdddGAN* from the distribution-learning described in Section 4.3 was fine-tuned by training on the filtered dataset with druglikeness values larger than 0.9. One point of interest was to examine, whether the *feedback-loop* from FeedbackGAN improves the optimization step. Recall that with the absence of the feedback-loop, the approach resembles pure transfer-learning, i.e. training the GAN on a fixed dataset. Whereas when updating the dataset in every epoch with newly generated data points, the FeedbackGAN might be able to explore the chemical space with high `QED` values better, since it shifts the distribution to learn from towards the region of interest (with high `QED` values). Nevertheless, it is still true that the fixed region for the pure transfer-learning is also a region of interest, as the samples from this filtered training set are all molecules which have high `QED` values. To investigate the difference between the pure transfer-learning approach and feedback-loop mechanism, the number of generated samples that satisfy the druglikeness condition is saved. Hence, in every evaluation step, $b = 5000$ samples are generated and the number of samples n_d that satisfy the desirability condition $\texttt{QED} > 0.90$ is saved. This number should ideally increase over the training epochs, showing that the GAN is able to generate a batch of compounds that (all) satisfy the druglikeness condition. Similar to the distribution-learning benchmark, the FCD, validity, novelty and uniqueness metrics were computed in each epoch to evaluate the performance of the GAN. Additionally, the mean `QED`-score over the set of generated valid samples $\tilde{S}_{\text{valid}} \subseteq \tilde{S}$ that contains $b_{\text{valid}} \leq b$ compounds was calculated with

$$c_{\text{q}}(\tilde{S}_{\text{valid}}) := \frac{1}{b_{\text{valid}}} \sum_{\tilde{s} \in \tilde{S}_{\text{valid}}} \texttt{QED}(\tilde{s}), \tag{74}$$

where we expect this value to converge to the maximal `QED` value of 0.95.



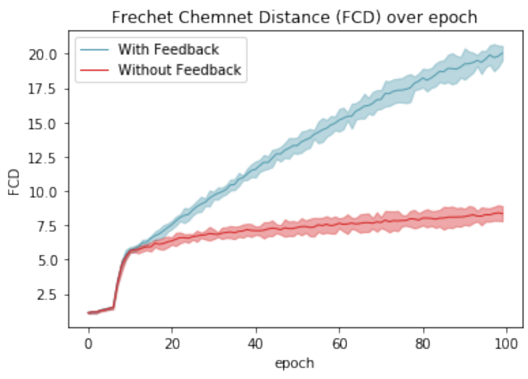
(a) Number of samples that satisfy the `QED` condition.



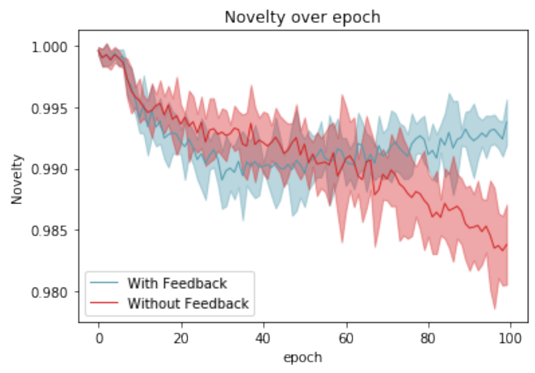
(b) Average `QED` score of generated samples. The mean druglikeness score is 0.9195.

Figure 51: **(a)** The GAN with feedback-loop mechanism is able to generate more samples that satisfy the `QED` condition. **(b)** The average druglikeness score from the valid SMILES batch should converge at least to the mean QED value of the filtered training set, which is 0.9195.

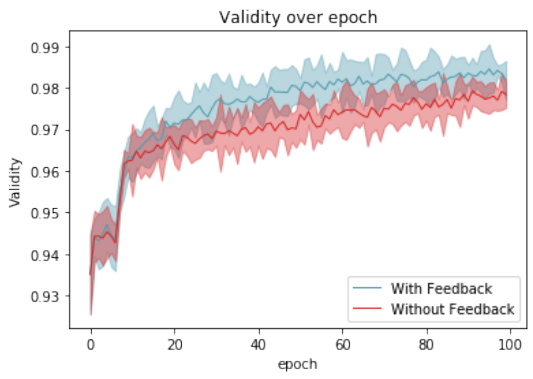
It turns out that including the feedback-loop mechanism increases the number of generated samples that meet the druglikeness condition as illustrated in Figure 51a. The experiment was conducted five times with different seeds for the settings with and without feeback-loop. Similar to the evaluation plots from the distribution-learning task, the solid lines for each curve represent mean values over the five different runs and the shaded parts visualize $+/-2\sigma$ deviations. The purpose of those shaded areas is to show how the evaluation metric differs for each seed-run. Regarding the evaluation metrics introduced in the distribution-learning benchmark, we observe that the pure transfer-learning approach with no feedback-loop performs better concerning the FCD measure.



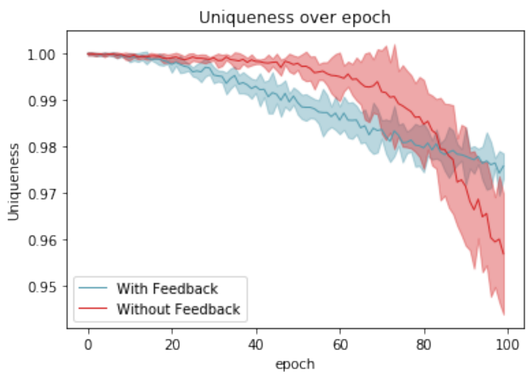
(a) FCD metric.



(b) Novelty metric.



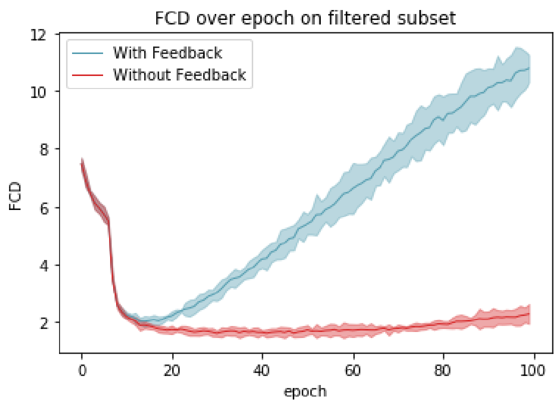
(c) Validity metric.



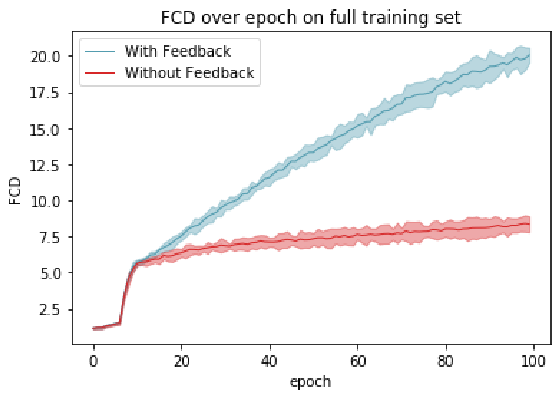
(d) Uniqueness metric.

Figure 52: Evaluation metrics for the pre-trained GANs with and without feedback-loop.

The novelty and uniqueness metrics are both still satisfying and the validity increases. The increase in validity for both models might be caused since the filtered dataset to learn from is reduced. Recall that learning the full ChEMBL training set included a large continuous space. Hence, the chance to generate invalid `cddd` representations is higher. If the probability space to learn from is smaller, the GAN is less prone to generate invalid samples. The FCD measure increases for both models. This is due to the difference between true reference (ChEMBL) training set and generated set. Remind that the FCD score in equation (71) computes whether the activations[26] a from a generated set $\tilde{S}$ and a training reference set $\mathcal{D}_s$ differ. The FCD score is a metric containing the estimated mean m and covariance C of those activations. If the respective means m_g, m_r and covariances C_g, C_r differ much, the FCD measure will increase. The difference of the two respective moments is caused since the true reference set is the full ChEMBL training set. Recall that the ChEMBL training set contains compounds with druglikeness scores ranging from 0 to 0.95 as displayed in Figure 47. So if ChEMBL is compared to the generated set, which only contains samples with high druglikeness values, it is natural that the FCD metric is high. Since the model weights for each epoch were saved, it was straightforward to compute the FCD metric regarding the filtered dataset. This was done in an additional step, leading to the following two FCD evaluation plots in Figure 53. Those two plots compare the FCD metric behavior for both models regarding the reference set to be compared with.



(a) Reference set: filtered subset with 31 778 samples.



(b) Reference set: full set with approximately 1.2 mio samples.

Figure 53: For both filtered training set and full training set, the FCD metric increases over the time of training for both models. The FCD score for FeedbackGAN increases strongly, indicating that the generated samples are not ChEMBL-like.

Figure 53a shows the FCD scores computed w.r.t. the fixed filtered dataset. The FCD score decreases in the first 15 epochs because the two pre-trained GANs are learning the distribution of compounds with druglikeness above 0.90.
With the start of epoch 20, the FCD score for FeedbackGAN increases actively because chemical space is possibly explored stronger and the dataset updated in such way that only artificial generated compounds are present. It is also worth mentioning that the FCD metric is more volatile for FeedbackGAN with increasing epoch, indicated by larger deviations in the blue shaded areas in Figure 53a. One reason for this could be the diversity of generated samples, when compared to the true fixed filtered training set. The diversity enhanced by FeedbackGAN is shown in the volatility of FCD scores with increasing epoch for each seed-run, concluding that the

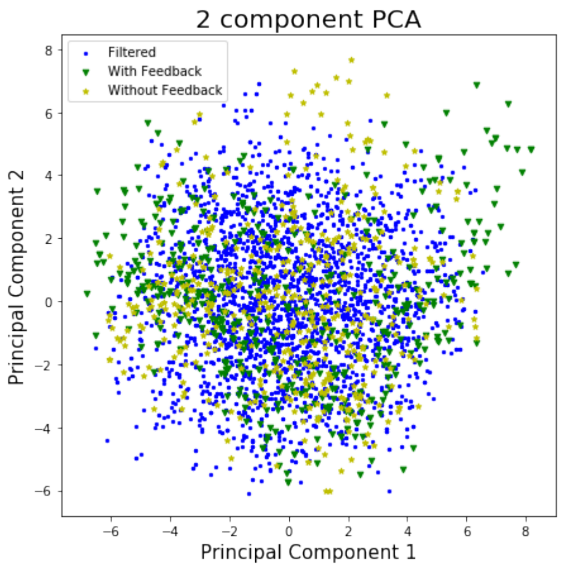
[26]Recall that the activations a are selected from a third-party network, called *ChemNet* that was trained to predict biological activity of chemical compounds on biological targets.

chemical space is probably explored stronger. However, the samples generated by FeedbackGAN are not ChEMBL-like anymore as indicated by the increasing FCD metric. Additionally, it seems that if the training of FeedbackGAN was conducted for more epochs, e.g. 100 epochs, the FCD metric would have increased steadily.
After epoch 20, the FCD metric stays almost constant for the GAN that was trained without the feedback mechanism since the exploration of chemical space is not so strong because learning is performed on a fixed dataset.
Regarding Figure 53b, the FCD increases for both GANs since the reference training set is the full training set provided by GuacaMol. Nevertheless, the FCD measure increases stronger for the generated samples if the GAN is trained with feedback-loop in both plots in Figure 53. This finding can be explained with the two plots in Figure 51: the FeedbackGAN can generate more samples that satisfy the `QED` > 0.90 condition. For example, in epoch 100 out of approximately 5000 generated valid samples, 3500 samples possess a druglikeness score higher than 0.90, whereas the GAN fine-tuned without feedback-loop only achieves around 2200 samples satisfying the condition. In addition to that, the mean `QED` score for the model with feedback-loop converges to the mean druglikeness score of the filtered training set of 0.9195, whereas the model without feedback seems to converge towards the value of 0.85.
However, it is important to mention that FeedbackGAN has a high potential to include bias in the training process. If the training set of FeedbackGAN is entirely replaced with artificial samples[27], the bias for upcoming artificial samples is reinforced. This bias moves on throughout the training process because FeedbackGAN learns the distribution of the new (biased) training data.
If the goal was to explore chemical space stronger, FeedbackGAN would be a possible optimization technique. Optimization often faces the trade-off dilemma between *exploration* and *exploitation*. It seems that FeedbackGAN explores the chemical space stronger, as indicated by the increasing FCD score. The exploitation is shown by the GAN, trained without the feedback mechanism since it learns on a fixed and limited but promising region of the search space.
For FeedbackGAN, one could also say that the exploration goes along with the exploitation because the `QED` function is used as a filter criterion. The `QED` function, however, is based on several desired physicochemical properties introduced by Bickerton et al. (2012), and inserting new samples into the training set based on only this criterion should be considered with care. One possible way to overcome this issue is by additionally computing the *Synthetic Accessibility* (SA) score [Ertl & Schuffenhauer (2009)]. The SAscore describes the compound synthetic accessibility as a score between one (easy to make) and ten (very difficult to make) and could be inserted as a second filtering step for samples that satisfy the `QED` condition. The SAscore, however, is also based on empirical results but seems reasonable as an

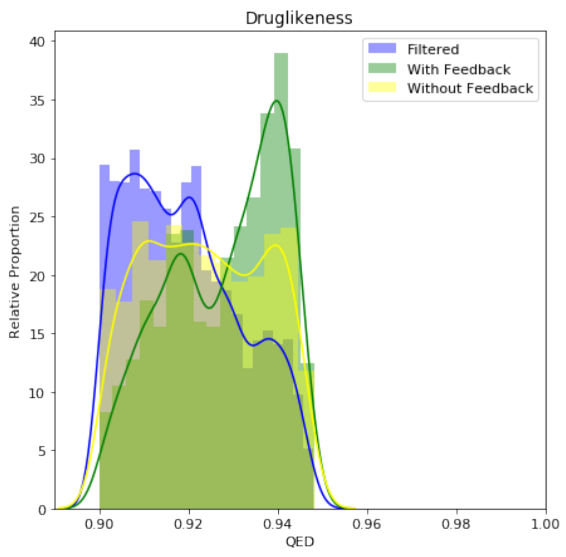
[27]This can be seen starting from epoch 20 in Figure 53a.

additional filter criterion before inserting samples that *just* satisfy the druglikeness condition. This additional method was not included but is worth investigating.

When projecting a set of generated samples on the PCA conducted earlier, Figure 54a shows that both sets from the model with and without feedback lie in the region of the filtered training set[28]. However, some samples lie outside of that region, for example in the upper right region. Figure 54a indicates that the generated samples from the FeedbackGAN might explore chemical space stronger, as the green triangles stand out of the concentrated region from the filtered training set, have high FCD metric and have high `QED` values. Since this finding also holds for generated samples from the model without feedback but with smaller FCD metric, we can assume that the two principal components are, as a *latent* representation, not sufficient and not able to visualize the true *druglikeness* property. Remind that the two prinicipal components only explained 10.77% of the total variance. Hence, we can only assume that the samples from FeedbackGAN somehow hold *other* properties which lead to high `QED` and FCD scores but also to the fact that some of the samples lie outside of the centered region.



(a) PCA on filtered and generated sets.



(b) KDE plot on filtered and generated sets.

Figure 54: **(a)** The samples from both models lie in the centered training region. **(b)** FeedbackGAN can generate more samples with high `QED` values as shown in the mode value.

Figure 54b shows that the empirical distributions between true filtered set and the set generated by FeedbackGAN differ, especially in their mode value. The empirical distribution of the GAN without feedback resembles the fixed filtered training distribution as expected. The kernel-density plot[29] suggests that FeedbackGAN can push

[28]For visualization only 3000 samples from the filtered training set were projected to the first two principal components. For the generated sets, each 500 samples from the last training epoch were projected onto the first two principal components.

[29]The KDE-plot was generated with 5000 samples each from the true filtered set, generated set with and without feedback loop for each model from the last training epoch 100.

towards regions of maximum druglikeness scores by iteratively inserting new data points with high scores, i.e. QED $\gg 0.90$ such as QED> 0.92. This thought is easier understood when plotting the mean or median QED values of generated samples from FeedbackGAN that satisfy the druglikeness condition as shown in Figure 55.

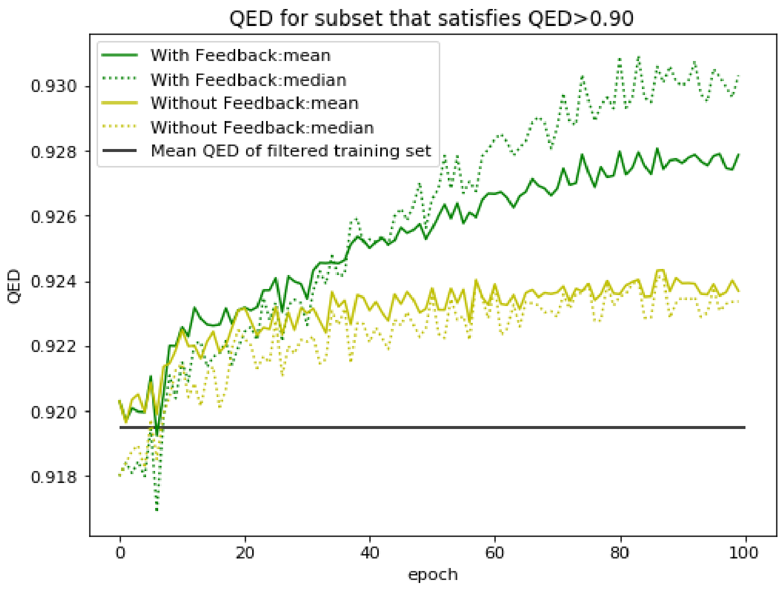


Figure 55: The QED score over epochs. Only samples of FeedbackGAN are inserted into the training set. The graphs show mean and median QED values of samples that satisfy the druglikeness condition. The median value is plotted to show that the training set for FeedbackGAN is updated with samples that almost have maximum QED score. Hence, the FeedbackGAN model learns on almost perfect drug-like samples and therefore also generates almost perfect drug-like samples. The samples generated by the GAN without feedback that satisfy the QED condition obtain on average a QED value of 0.923, whereas the FeedbackGAN model can generate samples that reach the maximum druglikeness score as demonstrated in the mode value in Figure 54b.

The final FeedbackGAN model is then able to generate new samples with maximum QED score, whereas the model without feedback seems to be restricted by the true reference (filtered) training set it has learned from in terms of exploration of chemical space. The current FeedbackGAN algorithm, however, might include bias into the training process as explained earlier.

As indicated by the increasing high FCD scores (see Figure 52a) for FeedbackGAN, *novel* and *diverse* structures are explored which are in fact not ChEMBL-like but still hold a high druglikeness score (see Figure 55).
In order to postulate that FeedbackGAN is indeed superior to pure transfer-learning, additional evaluation steps have to be included. Especially the step of inserting new samples into the training set has to be evaluated properly, for example with the SAscore. As shown in the novelty plot in Figure 52b, the model without feedback loop is also able to generate samples that are novel and not existent in the filtered training set. The generated samples without feedback seem to be more reasonable and ChEMBL-like and should be evaluated and compared to the samples from the FeedbackGAN model.

Figure 56 displays six generated molecules/compounds of the FeedbackGAN model in its last training epoch.

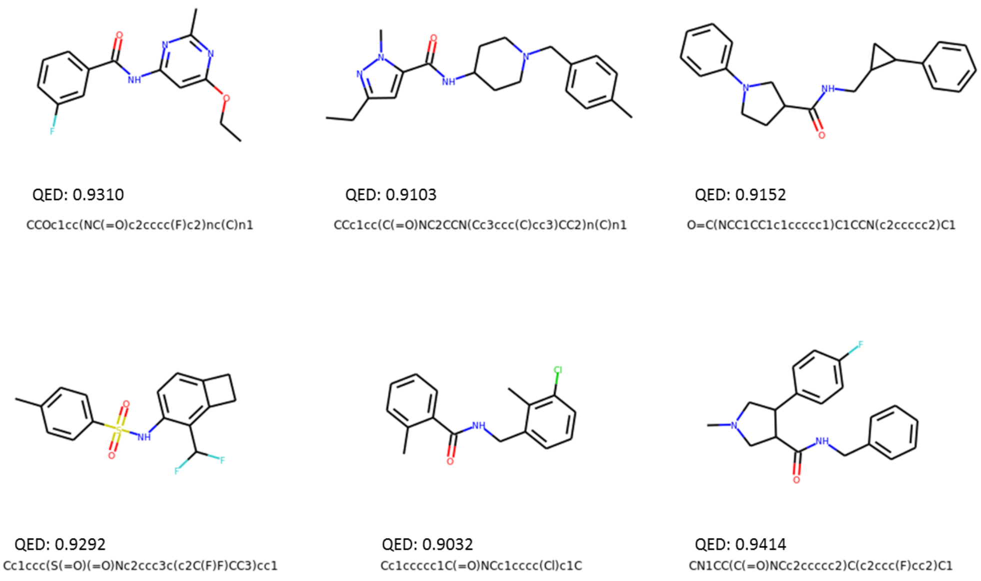

QED: 0.9310
CCOc1cc(NC(=O)c2cccc(F)c2)nc(C)n1
QED: 0.9103
CCc1cc(C(=O)NC2CCN(Cc3ccc(C)cc3)CC2)n(C)n1
QED: 0.9152
O=C(NCC1CC1c1ccccc1)C1CCN(c2ccccc2)C1
QED: 0.9292
Cc1ccc(S(=O)(=O)Nc2ccc3c(c2C(F)F)CC3)cc1
QED: 0.9032
Cc1ccccc1C(=O)NCc1cccc(Cl)c1C
QED: 0.9414
CN1CC(C(=O)NCc2ccccc2)C(c2ccc(F)cc2)C1

Figure 56: Samples generated by the FeedbackGAN model in the last training epoch with their respective druglikeness score and SMILES representation. The six randomly selected compounds are diverse and not included in the initial filtered training set.

All six compounds contain two aromatic rings meeting the desired number of aromatic rings by Bickerton et al. (2012) as listed in Table 10.

c_i	QED	logP	molWeight	TPSA	HBD	HBA	nROTB	nAROM
1	0.9310	2.5751	275.283	64.11	1	4	4	2
2	0.9103	2.6853	340.471	50.16	1	4	5	2
3	0.9152	3.4328	320.436	32.34	1	2	5	2
4	0.9292	3.8320	323.364	46.17	1	2	4	2
5	0.9032	3.8868	273.763	29.10	1	1	3	2
6	0.9414	2.7873	312.388	32.34	1	2	4	2

Table 10: Molecular statistics of compounds $\{c_i\}$. The approximate desired values from Bickerton et al. (2012) (SI) are: `[logP: 2.5, molWeight` $[\frac{g}{mol}]$`: 300, TPSA: 35 HBD: 1, HBA: 3, nROTB: 4, nAROM: 2, ALERTS: 0]`. The number of structural alerts (ALERTS) is not shown but included in the computation of the druglikeness score. The statistics were calculated using the `RDKit` python library.

All compounds except the first and fourth ones seem reasonable and synthesizable according to a computational chemist of our group. The first compound might not be stable due to the connection of the `NH` fragment next to an aromatic ring that contains nitrogen (`N`). The fourth compound contains a squared block next to an aromatic ring which is uncommon in chemical libraries of synthesized compounds, e.g. ChEMBL.

5 Discussion

At the beginning of the application part in Section 4, the capabilities of generative adversarial networks were shown in a proof-of-concept experiment by learning multivariate normal data with mean vector $\mu = 4$ and unit covariance matrix $\Sigma = I$. The dimensionality was set to $d = 50$. Three evaluation metrics were introduced that measure the goodness of synthesized samples by the generator network in the beginning of each training epoch. Often in generative modeling, evaluation metrics are a useful way of displaying the performance of the generative model. Where most generative models in image synthesis are easily evaluated by visual inspection, in our case with multidimensional data, visual evaluation of generated samples is not possible. Hence, it is worth mentioning that defining evaluation metrics in the domain of application is crucial for observing the performance of the generative model, in our case the GAN. Additionally, it is important to state that those evaluation metrics were not included into the overall objective function that was optimized over the training process.
In the proof-of-concept experiment, the three GAN variants (vanilla GAN, Wasserstein GAN with weight clipping and Wasserstein GAN with gradient penalty) were trained with different generator and discriminator architectures to investigate their performance for learning multivariate normal data. It turned out that all three variants were able to learn the true data distribution, if the generator network incorporated batch normalization. This insight was confirmed by the fact that the generated samples for each GAN variant minimized the defined evaluation criteria. Hence, the generated samples of the GANs followed a multivariate normal distribution with mean vector 4 and unit covariance matrix.

For the main application, several GANs were trained to learn a probability distribution over a training set of molecules, which were encoded in their continuous representations using the translation model by Winter et al. (2018). The provided training set by GuacaMol consists of SMILES representations and was translated into the continuous `cddd`-space $\mathcal{C}$, using the encoder network from the provided translation model in an initial preprocessing step. Since one goal was to conduct the distribution-learning benchmark from GuacaMol, four evaluation metrics namely, *validity, uniqueness, novelty* and *Fréchet Chemnet Distance (FCD)* were described and implemented in an evaluation step at the beginning of each training epoch of the GAN. Similar to the evaluation step for learning multivariate normal data, those evaluation metrics were not included into the overall objective of the GAN model. Furthermore, it is important to mention that the four listed evaluation metrics required the SMILES representation as input. Hence, in each evaluation step the generated `cddd`-representations were translated back into their SMILES representa-

tions by utilizing the decoder network of the translation model.
After training and comparing the performances between the three GAN variants, which all had shallow[30] network architectures for generator and discriminator, the Wasserstein GAN with gradient penalty Algorithm 3 was selected to be further used in the next extensive experiments. Those experiments included trying out different network architectures and optimizers. It turned out that batch normalization in the generator network was crucial for the GAN's capability to synthesize good samples that are ChEMBL-like and follow the distribution of the training set. In addition, increasing the complexity of the generator network by adding more hidden layers, facilitated the GAN to learn the true data distribution better. Hence, deeper generator networks with batch normalization could synthesize samples with smaller FCD measures. The Fréchet Chemnet Distance is a measure to show, how ChEMBL-like a set of generated samples is, and therefore the main evaluation metric taking into consideration when evaluating the GANs. Nonetheless, for the final *cdddGAN* model, besides the FCD criterion, the other three evaluation metrics validity, novelty and uniqueness were also satisfying. In order to check whether the GAN has not overfitted the training set, the FCD metric was additionally computed w.r.t. the provided test and validation set from GuacaMol. After evaluating on the two sets, we found out that our final GAN model *did not* overfit the training set.
For the validity, out of 5 000 generated samples, after decoding back into their SMILES representations, around 94% were valid. For the uniqueness and novelty metric, out of 5 000 samples, almost 100% uniqueness and novelty was achieved. Since the novelty criterion is almost perfect, our proposed *cdddGAN* is indeed a valuable generative model. It is able to generate samples that are ChEMBL-like and therefore resemble the true training set in a 'biochemical-way' and secondly able to generate valid and novel samples, which do not exist in the training reference set.

Finally in the *distribution-learning* application, the GuacaMol benchmark was performed in order to compare our trained *cdddGAN* to other state-of-the-art generative models in a fair way. After conducting the GuacaMol distribution-learning benchmark, our final GAN model outperforms the *ORGAN* [Guimaraes et al. (2017)] model in all benchmark evaluation metrics. The *ORGAN* model is also a GAN but trained with the discrete SMILES representation of compounds in contrast to our proposed model, which is trained on the unsupervised-learned `cddd` vector representation of compounds.
Our final model is comparable with the overall two best generative models *SMILES LSTM* [H. S. Segler et al. (2017)] and *VAE* [Gómez-Bombarelli et al. (2016)], where the first one is a next-character prediction model (see Section 2.2.6.2) and the second

[30]Shallow networks are networks which are not deep and mostly consists of only one or two hidden layers.

one a variational autoencoder similar to the one trained by Winter et al. (2018), but with the additional constraint that the latent bottleneck layer follows a pre-defined probability distribution. Both aforementioned generative models also work with the discrete SMILES representation of compounds.

Since we have obtained a GAN that is able to generate valid and novel compounds from a large chemical space, the next step was to use this model to fine-tune its generative process with the goal, to sample new compounds that have high drug-likeness scores, i.e. `QED` > 0.90. The optimization / fine-tuning towards generating compounds with high druglikeness values was achieved by training the learned *cd-ddGAN* model on a filtered subset of the original training data, where the filter criterion was to select only compounds with druglikeness score greater than 0.90.
The reason for applying transfer-learning was to obtain a GAN that is able to generate compounds with high druglikeness score, i.e. learn the *distribution* of drug-like compounds. One naive way to achieve the aforementioned would be to use the *cdddGAN* model, sample compounds and select those compounds that have a `QED` > 0.9. This procedure, however, is based on randomness and generating large chemical compound libraries[31] with this method would be impractical.
By filtering, we narrowed the distribution to be learned from in a sense, that the GAN had to learn the space of compounds with high druglikeness values. In addition to fine-tuning the pre-trained *cdddGAN* on the filtered dataset, which was called as pure transfer-learning approach, the feedback-loop mechanism was included in the fine-tuning of the model. In contrast to the pure transfer-learning approach, where the GAN learned from a fixed filtered set, the GAN trained with feedback-loop mechanism learned on an iteratively updated training set, where in each epoch, novel and unique generated compounds satisfying the druglikeness condition were inserted into the training set.
By enabling the feedback-loop mechanism it turned out that the GAN was able to explore the chemical space of very drug-like (`QED` > 0.90) compounds better than the pure-transfer learning GAN, and generated samples that on average converged towards the mean `QED` value of the filtered training set.
However, the FeedbackGAN algorithm has to be used with care due to the effect of including bias into the training process. Since the FeedbackGAN algorithm includes samples that *only* satisfy the `QED` condition, possible unrealistic molecules can be inserted into the training set. If the training set is completely replaced by artificial samples, the bias can proceed further and could lead to a chemical manifold that is characterized with high druglikeness but low synthetic ability. Hence, additional evaluation steps before inserting samples into the training set should be undertaken.

[31]Recall from Figure 1 that the building of large chemical libraries is crucial and the initial step in the drug discovery process. Speeding up the drug discovery process in the first steps with in silico (computer-based) methods is desired.

The FCD increased for both models since the generated compounds have `QED > 0.90` in contrast to the reference set (provided by GuacaMol) that contained compounds with `QED` values ranging from 0 up to the maximum value of 0.95.
As one goal of de novo drug design is generating novel (unknown) compounds that satisfy certain physico- and/or biochemical properties, high FCD scores for generated samples from both models do not mean that the samples are bad or poor. Since the FCD metric measures, whether a generated set is ChEMBL-like, a high FCD score for a generated set simply means that it is not ChEMBL-like. Recall that the ChEMBL database consists of compounds that have actually been synthesized. So if a compound is not ChEMBL-like, we can weakly neglect this point for the optimization of compounds in the task of *de **novo [new]** drug design*.
The FCD metric was only important for the distribution-learning benchmark since there the goal was to train a GAN to learn the probability distribution of a training set, which was extracted from the ChEMBL database.
As indicated in Figure 53b though, the generated samples from FeedbackGAN and the model without feedback should be compared due to the fact that both models increase in terms of the FCD score but the model without feedback is not increasing as strong as the model with feedback. The difference might lie in the case that bias into the training set from FeedbackGAN was included, when the complete initial training set was replaced by artifical samples from the generator network.

To sum it up, the main application starting from Section 4.3 showed the powerful capabilities of generative adversarial networks to learn a data distribution of the ChEMBL training set. The novel idea in our approach is that the training set consists of unsupervised-learned continuous representation of compounds, whereas most state-of-the-art de novo models rely on the discrete SMILES representation of compounds. We hold the view that using the unsupervised-learned continuous representation of compounds (`cddd`) provides benefits such as when using deep neural networks for any generative model, e.g. GANs or VAEs, optimizing the overall objective function becomes feasible. The feasiblity comes from the fact that the objective function is differentiable, since the included networks consists of differentiable activation functions and the generated input itself is continuous. Additionally, we hold the view that the `cddd` representation of compounds characterizes the chemical information of compounds better than the SMILES representation. Hence, optimization in the `cddd` space becomes easier as done by Winter et al. (2019) using the *particle swarm optimization* (PSO) algorithm to generate candidates of `cddd` compounds that satisfy multi-objective criteria. For most de novo generative models that used SMILES representation as inputs, algorithms from *reinforcement learning* were included in order to optimize the generated sequences towards certain biochemical properties [Olivecrona et al. (2017); Popova et al. (2018)]. The reinforcement learning approach was mainly utilized because the generated sequences are not differentiable because the next characters are sampled from a discrete multinomial distribution.

5.1 Outlook / Future Work

Future work on this application might include the formulation of **multi-objective optimization** for the (pre-trained) GAN model. If the approach from FeedbackGAN is proceeded, a *desirability-function* as proposed by Winter et al. (2019) needs to be implemented. The main idea behind FeedbackGAN is straightforward. For a generated set of compounds, compute their desirability scores and add those compounds into the training set, if the desirability condition is satisfied. For the simple single-optimization task of druglikeness (`QED`), this desirability score was already normalized between zero and one, by its definition[32].
Furthermore, a stronger evaluation of FeedbackGAN before inserting generated samples into the training set should be executed. As FeedbackGAN is prone to bias the training process, the synthetic accessibility score (SAscore) for samples with `QED>0.90` could be computed as an additional filter criterion.
The typical drug discovery process as shown in Figure 1 starts with the enrichment of large chemical libraries of compounds specified for a biological target such as a protein included in a disease. The target in this application was the druglikeness score, which in regular drug discovery processes falls into the 'Lead-to-Candidate' optimization step of compounds. The problem of selecting the biological target as the 'optimization' criterion is that an additional bioactivity prediction model is required, to validate if a set of compounds is active or not. This requires pre-training of the bioactivity model on assay data, which in practice is often limited.

Another approach that is more straightforward, is to train a conditional GAN [Mirza & Osindero (2014); Odena et al. (2017); Miyato & Koyama (2018); Gong et al. (2019)], where the condition c could be the concatenation of criteria we want to preserve. By training a conditional GAN (cGAN) we obtain a generator network that can sample new compounds $\tilde{x}$ that satisfy certain conditions c. Hence, in cGAN the generator network and discriminator network take the condition into consideration, i.e. $\tilde{x} = G(z|c)$ and for the discriminator network $D(x|c)$. By inserting additional information in form of conditions/labels, the GAN is supposed to perform better on learning the probability distribution of samples conditioned on their labels. The exploration of capabilities from cGAN in de novo drug design is an interesting and promising next step and left for research.

[32]Note that the `RDKit-QED` function only returns values between 0 and 0.95 in contrast to the stated maximum value in the original paper by Bickerton et al. (2012).

Appendices

A Derivation of Wasserstein GAN

The following paragraphs include various mathematical definitions in order to understand the initial objective of the Wasserstein-1 distance as applied in the Wasserstein GAN Algorithm 2. Furthermore, a sketch of the proof to derive the dual form of the primal Wasserstein-1 distance will be shown.

A.1 K-Lipschitz Continuity

Given two metric spaces $(\mathcal{X}, d_X)$ and $(\mathcal{Y}, d_Y)$, where $d_X : (\mathcal{X} \times \mathcal{X}) \to \mathbb{R}^+$ and $d_Y : (\mathcal{Y} \times \mathcal{Y}) \to \mathbb{R}^+$ are distance functions in the respective metric space, a function $f : \mathcal{X} \to \mathcal{Y}$ is said to be $K-$Lipschitz continuous, if there exists a (smallest possible) real constant $K \geq 0$ such that

$$d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2), \forall x_1, x_2 \in \mathcal{X}. \tag{75}$$

In many applications of GANs the metrics d_X and d_Y are l_1 or l_2 distances.
Assume that $\mathcal{X}, \mathcal{Y} \subseteq \mathbb{R}$ and we have the l_1 distance, e.g. $d_X(x_1, x_2) = |x_1 - x_2|$ and $d_Y(f(x_1), f(x_2)) = |f(x_1) - f(x_2)|$.
The upcoming two examples have the purpose to understand the lipschitz continuity on a function f and its relation to the gradient norm of f.

Example 1.
Let $f(x) = x$ be on the interval $I = [a, b]$, where $a > b$ and $a, b \in \mathbb{R}$.
For f to be K-Lipschitz continuous in the interval I, following has to hold:

$$|d_Y(f(x_1)), f(x_2))| = |f(x_1) - f(x_2)| = |x_1 - x_2| \leq K d_X(x_1, x_2) = K|x_1 - x_2|,$$

which turns into equality, if and only if the smallest $K = 1$.
Hence, $f(x)$ is $1-$Lipschitz continuous in the interval I and in fact in $\mathbb{R}$.
Additionally, note that

$$\frac{|d_Y(f(x_1)), f(x_2))|}{d_X(x_1, x_2)} = \frac{|f(x_1) - f(x_2)|}{|x_1 - x_2|} = \frac{|x_1 - x_2|}{|x_1 - x_2|} = 1,$$

where the secant between $(x_1, f(x_1))$ and $(x_2, f(x_2))$ has (maximal) slope of one. Since f itself is a linear function and $1-$Lipschitz continous, the gradient is one and also has gradient norm one over all the metric space $\mathcal{X} = \mathbb{R}$.

Example 2.

Let $f(x) = x^2$ be on the interval $I = [a, b] = [0, 1]$. Then we can derive the following Lipschitz continuity by definition in equation (75) as:

$$d_Y(f(x_1), f(x_2)) = |x_1^2 - x_2^2| = |x_1 - x_2||x_1 + x_2| \leq K|x_1 - x_2|,$$

where $K = 2\max(|a|, |b|) = 2\max(0, 1) = 2$.

Hence, the function f is $2-$Lipschitz continuous in interval $[0, 1]$. Note that the gradient for this function is $f'(x) = 2x$ and the gradient has maximal norm of the value two in the defined interval, where it is $2-$Lipschitz continuous. Also recall that if a secant between $f(x_1)$ and $f(x_2)$, where $x_1, x_2 \in (0, 1)$ is drawn, the slope of the secant is bounded by the maximal value of two, which follows from the Lipschitz continuity.

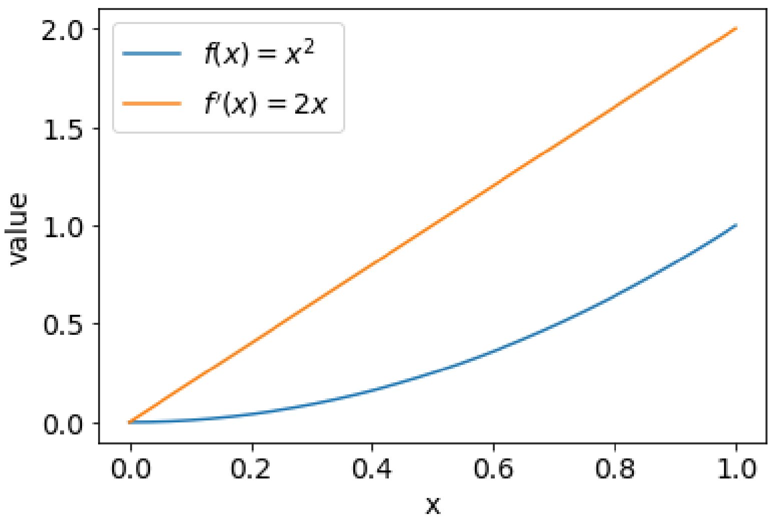


Figure 57: The function $f(x) = x^2$ is globally $2-$Lipschitz continuous in the interval $I = [0, 1]$. Furthermore its derivative is bounded with maximal norm two in the interval. If a secant is drawn for any points $f(x_1)$ and $f(x_2)$, the slope is also bounded with maximal value of two.

Also note that $f(x) = x^2$ would be $1-$Lipschitz continuous in I if we divide f by $K = 2$.

A.2 Definition Wasserstein-p Distance

Let $\mathcal{X}, \mathcal{Y}$ be two compact spaces on $\mathbb{R}^n$ and p_θ and p_r be two probability distributions on the spaces $\mathcal{X}$ and $\mathcal{Y}$. We will denote $d_p(x,y) : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^+$ with its definition $d_p(x,y) := ||x-y||_p = (\sum_{i=1}^n |x_i - y_i|^p)^{\frac{1}{p}}$ as l_p distance norm for samples coming from p_θ and p_r. Since we usually deal with continuous well defined probability densities, $\mathcal{X} = \mathcal{Y} = \mathbb{R}^n$. The Wasserstein-p distance between two probability measures p_θ and p_r on the joint space $M = (\mathcal{X} \times \mathcal{Y})$ is defined as

$$\begin{aligned} W_p(p_r, p_\theta) &:= \left(\inf_{\gamma \in \prod(p_r, p_\theta)} \int_{\mathcal{X} \times Y} d_p(x,y)^p d\gamma(x,y)) \right)^{\frac{1}{p}} \\ &= \left(\inf_{\gamma \in \prod(p_r, p_\theta)} \mathbb{E}_{(x,y)\sim\gamma}[d_p(x,y)^p] \right)^{\frac{1}{p}}, \end{aligned} \tag{76}$$

where $\prod(p_r, p_\theta)$ denotes the set of marginal distributions such that the following two constraints are satisfied

$$\int_x \gamma(x,y)dx = p_r(y) \text{ and } \int_y \gamma(x,y)dy = p_\theta(x). \tag{77}$$

The Wasserstein-p distance can be interpreted as the minimum cost to transport the model distribution p_θ to the real distribution p_r.

Example 1: Wasserstein-1 distance.

$$W_1(p_r, p_\theta) := \inf_{\gamma \in \prod(p_r, p_\theta)} \left(\mathop{\mathbb{E}}_{(x,y)\sim\gamma} [||x-y||_1^1] \right)^{\frac{1}{1}} = \inf_{\gamma \in \prod(p_r, p_\theta)} \mathop{\mathbb{E}}_{(x,y)\sim\gamma} [||x-y||_1]. \tag{78}$$

Example 2: Wasserstein-2 distance.

$$W_2(p_r, p_\theta) := \left(\inf_{\gamma \in \prod(p_r, p_\theta)} \mathop{\mathbb{E}}_{(x,y)\sim\gamma} [||x-y||_2^2] \right)^{\frac{1}{2}}. \tag{79}$$

A.3 Derivation Sketch Dual Problem of Wasserstein-1 Distance

For the upcoming sketch, $|| \cdot ||$ denotes the $p = 1$ norm, hence $|| \cdot || = || \cdot ||_1$.
Recall that the Wasserstein-1 distance has to find the optimal transport plan/-coupling γ^* that minimizes equation (78). In the following part, we will derive a formulation how we can rewrite the constraint of any transport plan $\gamma \in \prod(p_r, p_\theta)$ to satisfy the marginalization property in equation (77).

Consider following optimization problem

$$\begin{aligned}
&\sup_f \left[\int f(y')p_r(y')dy' - \int\int f(y)\gamma(x,y)dxdy\right] \\
&= \sup_f \left[\mathop{\mathbb{E}}_{y'\sim p_r}[f(y')] - \mathop{\mathbb{E}}_{(x,y)\sim\gamma}[f(y)]\right],
\end{aligned} \tag{80}$$

where f can be any function on $\mathbb{R}^n$. We observe the following points.

- Left-hand side: Expectation of f under p_r.
- Right-hand side: Expectation of f under the marginal distribution $\int \gamma(x,y)dx$.
- This expression is clearly zero **for all** possible f, if the marginal constraint over p_r is met since the two terms will then be identical. Hence, it must satisfy the marginalization constraint $\int \gamma(x,y)dx \equiv p_r(y)$. If the constraint is not satisfied, the supremum is ∞.

In order to prove the duality, we need to reformulate the constrained optimization (which has the condition on the transport plan / coupling γ) in the primal problem in equation (78) as a less constrained optimization that does not include γ but rather the 1−Lipschitz continuity on f. We will achieve this by adding suitable terms regarding the set of all possible transport plans $\gamma \in \prod(p_r, p_\theta) := \pi$, to reformulate the condition and rule out those transport plans that do not follow $\prod(p_r, p_\theta)$ and violate the marginalization property. Adding the term from equation (80) into the Wasserstein-1 distance from equation (78) leads to

$$\begin{aligned}
W_1(p_r, p_\theta) &= \inf_{\gamma\in\pi} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||] \\
&= \inf_{\gamma} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||] + 0_{\gamma\in\pi} + \infty_{\gamma\notin\pi} \\
&= \inf_{\gamma} \mathbb{E}_{x,y\sim\gamma}[||x-y||] \\
&\quad \underbrace{+ \sup_f \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)] - \mathbb{E}_{(x,y)\sim\gamma}[(f(y)-f(x))]}_{=0,\text{ if } \gamma\in\pi, \text{ else } \infty.} \\
&= \inf_{\gamma} \sup_f \mathop{\mathbb{E}}_{x,y\sim\gamma}[||x-y||] + \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)] \\
&\quad - \mathbb{E}_{(x,y)\sim\gamma}[(f(y)-f(x))],
\end{aligned} \tag{81}$$

creating a bilevel optimization similar to the minmax game for any GAN.
Note that in line four when adding the term (two times), we made use of the bilinearity of the expectation, where $\mathbb{E}_{(x,y)\sim\gamma}[(f(y)-f(x))] = \mathbb{E}_{(x,y)\sim\gamma}[f(y)] - \mathbb{E}_{(x,y)\sim\gamma}[f(x)]$. The next step is to make use of the **minmax**-principle that states that in certain cases (which we will not prove here) the order of inf and sup can be reverted without changing the solution using Sion's minimax theorem.

Applying Sion‘s minimax theorem on equation (81) leads to

$$\begin{aligned} W_1(p_r, p_\theta) &= \sup_f \inf_\gamma \mathop{\mathbb{E}}_{(x,y)\sim\gamma}[||x-y|| - (f(y)-f(x))] + \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)] \\ &= \sup_f \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)] + \inf_\gamma \mathop{\mathbb{E}}_{(x,y)\sim\gamma}[||x-y|| - (f(y)-f(x))]. \end{aligned} \tag{82}$$

The term $||x-y||$ can be rewritten since the 1-norm is a proper norm satisfying the homogeneity property, $||x-y|| = ||(-1)(-x+y)|| = |(-1)| \cdot ||y-x|| = ||y-x||$ and we can see the term of $1-$Lipschitz continuity in the infimum over the expectation.

$$\begin{aligned} ||f(y)-f(x)|| &\leq ||y-x|| \\ ||f(y)-f(x)|| - ||y-x|| &\leq 0 \\ \underbrace{||y-x|| - ||f(y)-f(x)||}_{=:l(x,y)} &\geqslant 0. \end{aligned} \tag{83}$$

Note that the bilevel optimization is $\sup_f \inf_\gamma$, where the terms related to γ are coupled to the function f. Hence, it is desired to have as solution for the inner optimization (inf states the greatest lower bound) 0 rather than $-\infty$, which could be the case if f is not $1-$Lipschitz continuous. We constrain $l(x,y) \geqslant 0$ for all x, y then the infimum is 0 and reached by assigning the whole probability density $\gamma(x,y)$ on the $x = y$ subspace. Conversely, if there is a region, where $l(x,y) < 0$, the cost can become arbitrarily large by assigning an arbitrarily large amount of density to that region, leading to $(-1)\cdot\infty$. Finally, we obtain the following optimization problem, where we can change the condition into a constraint.

$$\begin{aligned} W_1(p_r, p_\theta) &= \sup_f \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)] + \underbrace{\inf_\gamma \mathop{\mathbb{E}}_{x,y\sim\gamma}[||y-x|| - (f(y)-f(x))]}_{=0,\text{ if } ||f||_L \leq 1, \text{ else } -\infty} \\ &= \sup_{||f||\leq 1} \mathop{\mathbb{E}}_{s\sim p_r}[f(s)] - \mathop{\mathbb{E}}_{t\sim p_\theta}[f(t)], \end{aligned} \tag{84}$$

finishing the sketch proof, where $||f||_L \leq 1$ means that the function f is a $1-$Lipschitz function.

B Maximum Likelihood Optimization and Kullback-Leibler Divergence Minimization

Proof of equation (35) that solving the maximum likelihood problem is equivalent to minimizing the KL-divergence as $N \to \infty$.
Recall from equation (36) that the Kullback-Leibler divergence is formulated as

$$\varphi_{KL}(p_r || p_\theta) = \int_x p_r(x) \log \frac{p_r(x)}{p_\theta(x)} dx.$$

The law of the unconscious statistician (LOTUS) [Ringnér (2009)] states that if X is a random variable, so is the transformation $g(X)$.
One does not need to find the probability distribution $f_{g(X)}(x)$ in order to compute $\mathbb{E}[g(X)]$. With LOTUS one can compute the mean of the transformed (unknown) probability distribution $f_{g(X)}(x)$ via

$$\mathbb{E}[g(X)] = \int_{\mathbb{R}} g(x) f_X(x) dx,$$

where $f_X(x)$ is the known probability distribution of the random variable X. With the strong law of large numbers as the sample size increases, it holds that the sample average of an i.i.d sample from $g(X) = \log(X)$ will converge almost surely to $\mathbb{E}[g(X)]$ leading to the proof:

$$\begin{aligned}
\lim_{N \to \infty} \arg\max_{\theta \in \mathbb{R}^d} \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x^{(i)}) &= \arg\max_{\theta \in \mathbb{R}^d} \mathbb{E}[\log p_\theta(X)] \\
&= \arg\max_{\theta \in \mathbb{R}^d} \int_x p_r(x) \log p_\theta(x) dx \\
&= \arg\min_{\theta \in \mathbb{R}^d} - \int_x p_r(x) \log p_\theta(x) dx \\
&= \arg\min_{\theta \in \mathbb{R}^d} \int_x p_r(x) \log p_r(x) dx - \int_x p_r(x) \log p_\theta(x) dx \\
&= \arg\min_{\theta \in \mathbb{R}^d} \int_x p_r(x) \log \frac{p_r(x)}{p_\theta(x)} dx \\
&= \arg\min_{\theta \in \mathbb{R}^d} \varphi_{KL}(p_r || p_\theta),
\end{aligned}$$

where in the third line we switch from maximization to a minimization by multiplying the objective with -1. In the fourth line, since the minimization is over $\theta \in \mathbb{R}^d$ we can add terms that are not dependent on θ, as this does not affect the minimization. The last two lines are showing that the maximum likelihood optimization problem is equal to the KL-divergence minimization.

C Distribution-Learning

This Appendix paragraph shows additional results for the distribution learning.

C.1 Exploring different Architectures and Settings

In this experiment, several neural network architectures were tested and trained following the Improved WGAN Algorithm 3 over 50 epochs. It turns out that the GAN model is learning to generate molecules resembling the training set, i.e. ChEMBL space, if the generator network includes batch normalization [Ioffe & Szegedy (2015)] in each hidden layer and the critic network does not incorporate batch nor layer normalization (see Figure 58). If the generator network does not support batch normalization or contains layer normalization [Ba et al. (2016)], the generator network is not able to generate good samples that minimize the FCD measure as illustrated in Figure 59. Furthermore, when applying batch normalization in the generator network as well as in the critic network, the generated molecules are also not minimizing the FCD measure, see `param10` in Figure 59.
To conclude, batch normalization should be applied in the generator network batch normalization and the critic should not contain any normalization within the hidden layers. The hidden layers of the critic should compose only the activation of an affine transformation, i.e. weighted sum shifted with a bias.
Regarding the legend in the plots, the normalization can contain following attributes: {batch normalization '`batch`', layer normalization '`layer`', no normalization '`no`'}.

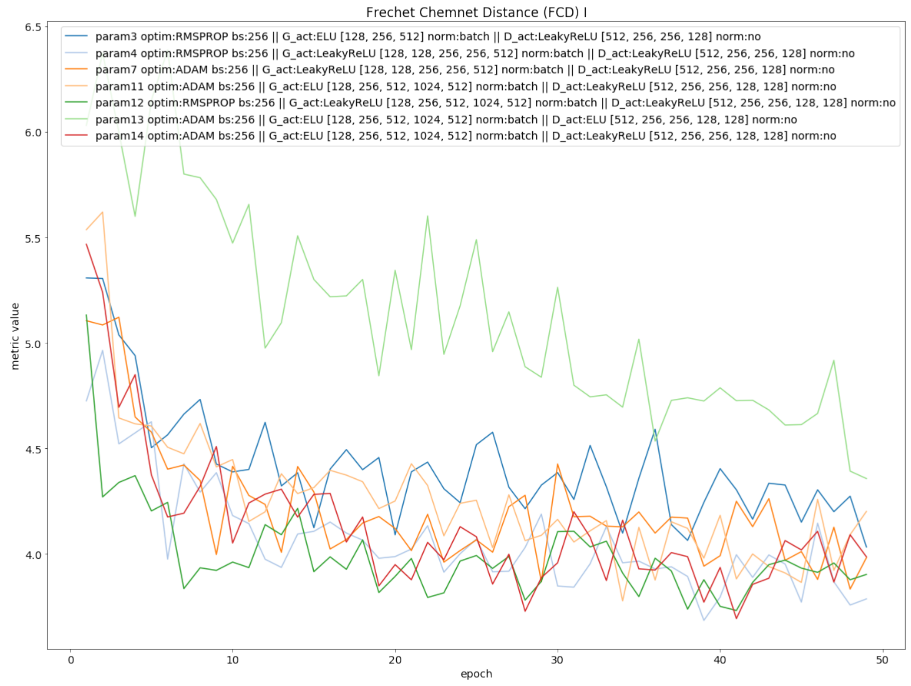


Figure 58: Including batch normalization in the generator network enables the generator to sample compounds that minimize the FCD measure. The critic network should not support any normalization.

The legend for each plot shows the optimizer and batch-size used for each parameter-run. Additionally, the generator network structure with the number of hidden layers and their respective neurons within each layer are displayed followed by the normalization attribute. The same logic is included for the critic network.

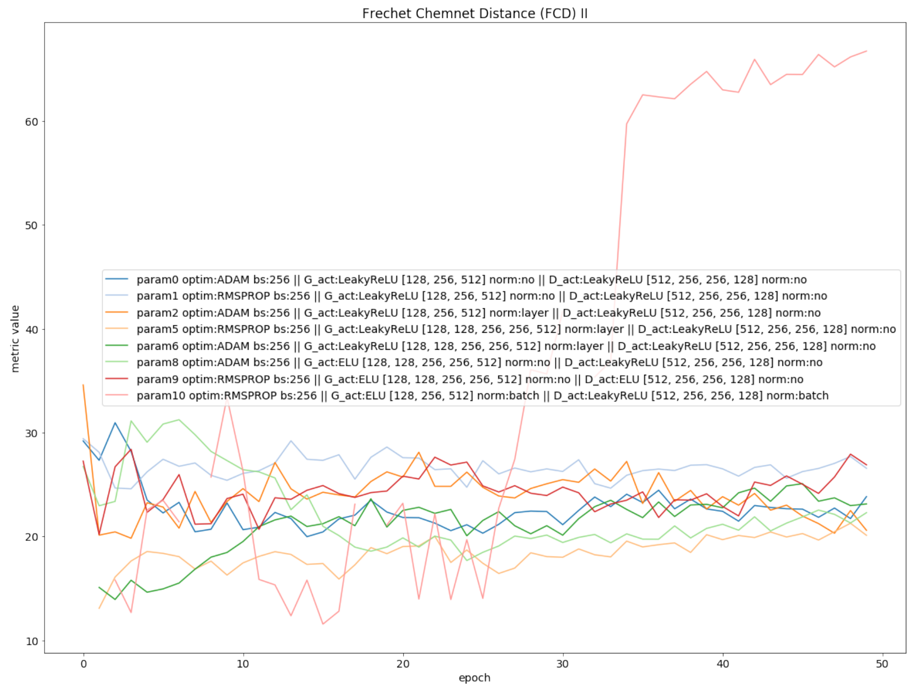


Figure 59: Including layer- or no normalization in the generator network leads to samples that do not minimize the FCD measure. `param10` deteriorates over the training.

For the validity, uniqueness and novelty the presence of batch normalization in generator- and no normalization in critic network is best as shown in Figure 60, 61 and 62.

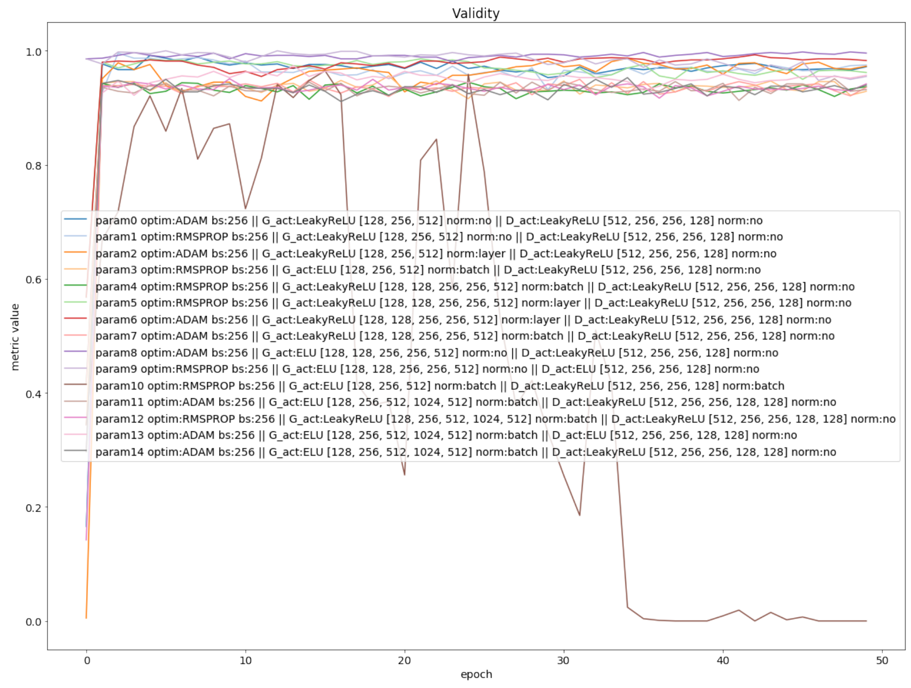


Figure 60: Validity is overall good except for `param10`. Surprisingly, if the generator network has no batch normalization, validity converges to 100% but with the disadvantage of high FCD scores.

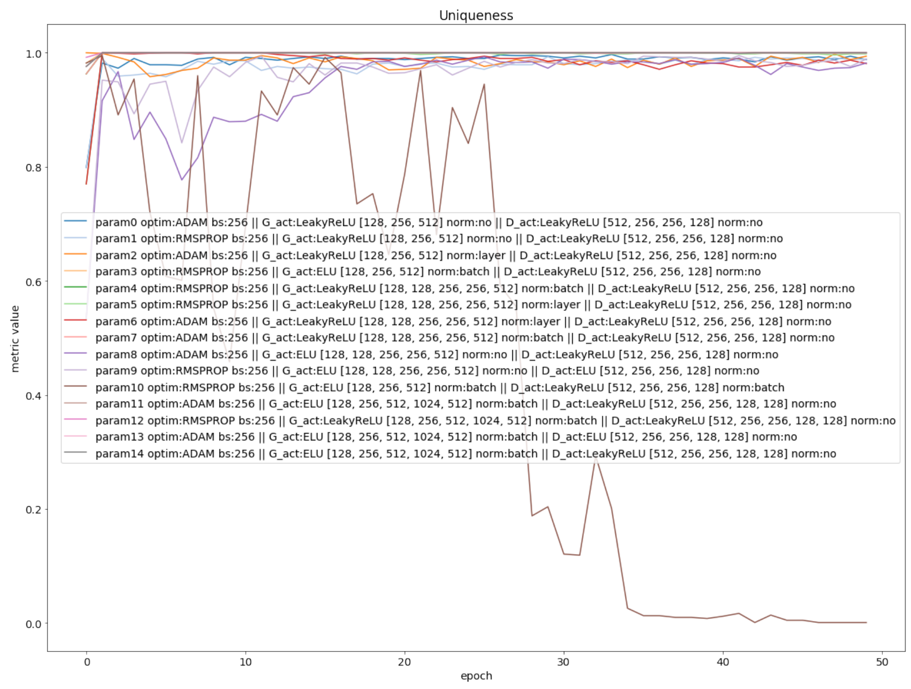


Figure 61: Uniqueness metrics. For `param10` the phenomenon of *mode collapse* happened. Apart from the mode collapse issue, `param10` is not able to create valid SMILES as seen in Figure 60.

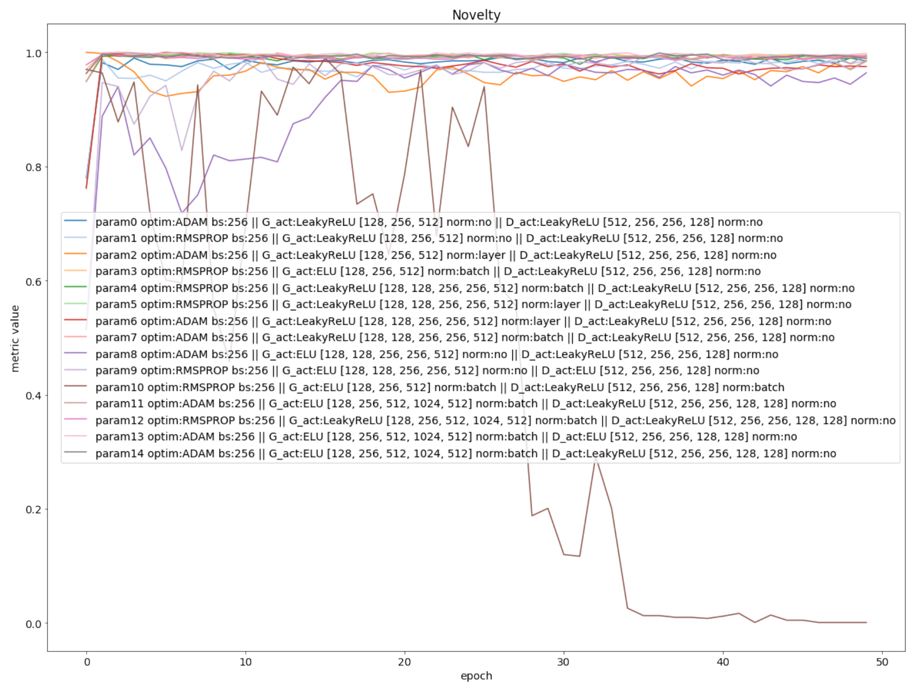


Figure 62: Novelty metrics. The model from `param10` performs worst in novelty as well.

This experiment was extensively conducted and the parameter settings were arbitrarily chosen. Additionally, sampling from a uniform distribution $Z \sim U(-1, 1)$ was tested. There was almost no difference when sampling from the uniform distribution compared to the standard Gaussian. Including batch normalization in the critic network in `param10` led to poor evaluation metrics. A possible reason why batch normalization degraded the Improved WGAN Algorithm 3 is the Lipschitz continuity, which is enforced through the gradient penalty term. By enabling batch normalization the gradient penalty could have affected the critic weights in such way that the 1−Lipschitz constraint is strongly violated. After an analysis of the Wasserstein GP loss curve for `param10` this hypothesis is confirmed.

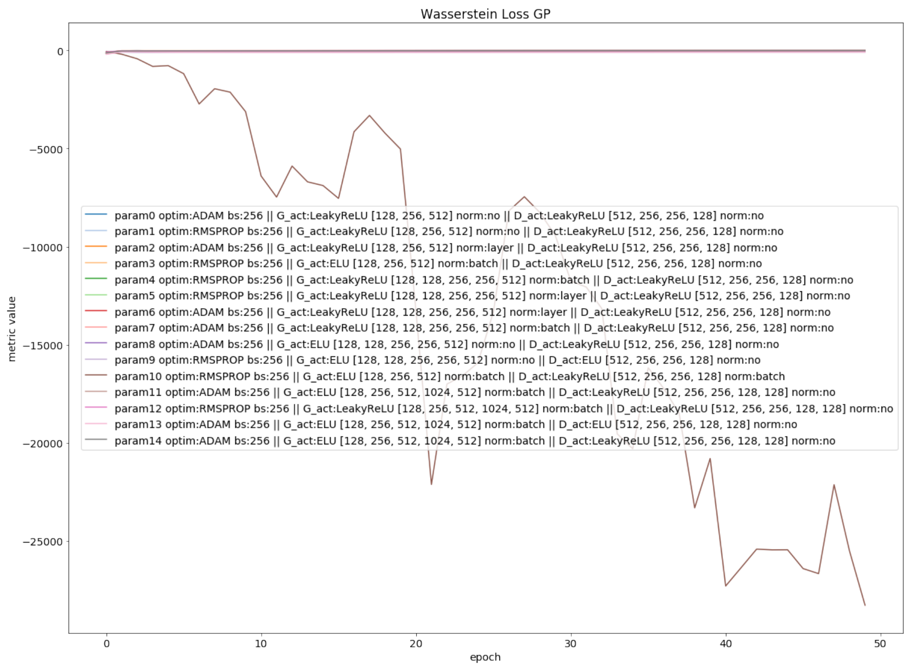


Figure 63: The Wasserstein GP loss is deteriorating for the run `param10`. The loss decreases up to -25000. Recall that the minmax WGAN optimization problem has the optimal value of zero. Batch normalization in the critic network seems to disturb the 1−Lipschitz continuity and hence should not be included in the critic network.

Figure 64 shows the Wasserstein GP loss of the parameter runs except for `param10`.

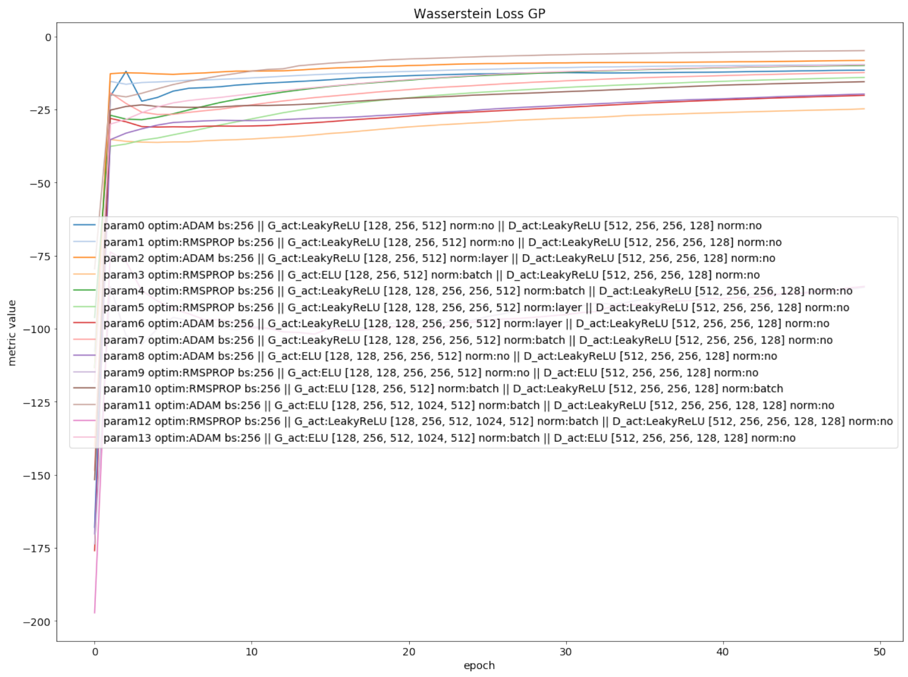


Figure 64: The Wasserstein GP loss is improving for all parameter runs. All those parameter runs include a critic network that does not support batch normalization.

After an analysis of the evaluation, the parameter settings from `param12` were selected as model architecture for generator and critic network for the final best model, i.e. *cdddGAN*. The deciding criterion was the FCD metric, shown in Figure 58.

C.2 Comparison Baseline Model and Best Model

In the following Table 11 the four evaluation metrics for *validity, uniqueness, novelty* and *Fréchet Chemnet Distance (FCD)* for the baseline model (Improved WGAN with baseline network architectures from Table 5 and 6) and the selected best model (Improved WGAN with best network architectures from Table 7 and 8) for the epochs 50,100,150 and 200 are shown.
Since the experiments were conducted five times with different seeds, the mean value for each evaluation metric and its standard deviation are shown. Those metrics were computed w.r.t. the provided training set from GuacaMol.

Metric	**Epoch**	**Best model**	**Baseline**
validity	50	$0.9321 \pm (0.0040)$	$0.9356 \pm (0.0011)$
	100	$0.9325 \pm (0.0032)$	$0.9354 \pm (0.0029)$
	150	$0.9350 \pm (0.0048)$	$0.9362 \pm (0.0017)$
	200	$0.9366 \pm (0.0029)$	$0.9376 \pm (0.0015)$
uniqueness	50	$1.0000 \pm (0.0000)$	$1.0000 \pm (0.0000)$
	100	$1.0000 \pm (0.0000)$	$1.0000 \pm (0.0000)$
	150	$0.9998 \pm (0.0002)$	$0.9999 \pm (0.0001)$
	200	$1.0000 \pm (0.0000)$	$1.0000 \pm (0.0000)$
novelty	50	$0.9898 \pm (0.0013)$	$0.9941 \pm (0.0010)$
	100	$0.9903 \pm (0.0011)$	$0.9939 \pm (0.0012)$
	150	$0.9888 \pm (0.0022)$	$0.9944 \pm (0.0003)$
	200	$0.9878 \pm (0.0014)$	$0.9943 \pm (0.0007)$
FCD	50	$1.2178 \pm (0.0422)$	$1.8850 \pm (0.0568)$
	100	$1.1516 \pm (0.0278)$	$1.7868 \pm (0.0529)$
	150	$1.1146 \pm (0.0162)$	$1.7688 \pm (0.0511)$
	200	$1.0813 \pm (0.0204)$	$1.7182 \pm (0.0684)$

Table 11: Mean criteria value c and its standard deviation σ. The displayed value is $c \pm (\sigma)$

In addition, the results for a 'random baseline' were computed. The random baseline is method, where 5000 samples are drawn from $U \sim (-1, 1)$ and then translated back to the SMILES representation. By adding this, the most naive way of learning the `cddd` space $\mathcal{C} = (-1, 1)^{512}$ is illustrated.
It turned out that out of five runs with different seeds, the random baseline generated samples with $0.3011 \pm (0.0064)$ validity.
Uniqueness and novelty metric were almost perfect having value of one but not useful for application since the generated samples were not valid SMILES. The FCD metric was not computable due to the invalid SMILES. Even after filtering out the valid SMILES representations, the FCD metric could not be calculated.

List of Figures

List of Tables

List of Algorithms

Bibliography

Adiga, S., Attia, M., Chang, W.-T. & Tandon, R. (2018), On the tradeoff between mode collapse and sample quality in generative adversarial networks, pp. 1184–1188.

Aggarwal, C. C. (2018), *Neural Networks and Deep Learning - A Textbook*, Springer.
URL: *https://doi.org/10.1007/978-3-319-94463-0*

Anil, C., Lucas, J. & Grosse, R. (2019), Sorting out Lipschitz function approximation, *in* K. Chaudhuri & R. Salakhutdinov, eds, 'Proceedings of the 36th International Conference on Machine Learning', Vol. 97 of *Proceedings of Machine Learning Research*, PMLR, Long Beach, California, USA, pp. 291–301.
URL: *http://proceedings.mlr.press/v97/anil19a.html*

Arjovsky, M. & Bottou, L. (2017), 'Towards principled methods for training generative adversarial networks', *ArXiv* **abs/1701.04862**.

Arjovsky, M., Chintala, S. & Bottou, L. (2017), 'Wasserstein gan', *ArXiv* **abs/1701.07875**.

Ba, J., Kiros, J. R. & Hinton, G. E. (2016), 'Layer normalization', *ArXiv* **abs/1607.06450**.

Bickerton, R., Paolini, G., Besnard, J., Muresan, S. & Hopkins, A. (2012), 'Quantifying the chemical beauty of drugs', *Nature chemistry* **4**, 90–8.

Bischl, B. (2018*a*), 'Lecture notes in 'deep learning' chapter 1: Introduction to dl'. `https://moodle.lmu.de/course/view.php?id=4192`.

Bischl, B. (2018*b*), 'Lecture notes in 'deep learning' chapter 2: Optimization i'. `https://moodle.lmu.de/course/view.php?id=4192`.

Bischl, B. (2018*c*), 'Lecture notes in 'deep learning' chapter 7: Recurrent neural networks'. `https://moodle.lmu.de/course/view.php?id=4192`.

Bischl, B. (2019*a*), 'Lecture notes in 'cim 1 - statistical computing', lecture 11 - multivariate unrestringierte optimierung'. `https://moodle.lmu.de/course/view.php?id=3927`.

Bischl, B. (2019*b*), 'Lecture notes in 'predictive modelling' chapter 1: Introduction and formalization'. `https://moodle.lmu.de/course/view.php?id=4769`.

Brown, N. (2009), 'Chemoinformatics&mdash;an introduction for computer scientists', *ACM Comput. Surv.* **41**(2), 8:1–8:38.
URL: *http://doi.acm.org/10.1145/1459352.1459353*

Brown, N., Fiscato, M., Segler, M. H. & Vaucher, A. C. (2019), 'Guacamol: Benchmarking models for de novo molecular design', *Journal of Chemical Information and Modeling* **59**(3), 1096–1108.
URL: *https://doi.org/10.1021/acs.jcim.8b00839*

Cao, N. D. & Kipf, T. (2018), 'Molgan: An implicit generative model for small molecular graphs', *ArXiv* **abs/1805.11973**.

Cereto-Massagué, A., Montes, M., Valls, C., Mulero, M., Garcia-Vallve, S. & Pujadas, G. (2014), 'Molecular fingerprint similarity search in virtual screening', *Methods (San Diego, Calif.)* **71**.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. & Shelhamer, E. (2014), 'cudnn: Efficient primitives for deep learning.', *CoRR* **abs/1410.0759**.
URL: *http://dblp.uni-trier.de/db/journals/corr/corr1410.htmlChetlurWVCTCS14*

Cho, K., van Merriënboer, B., Bahdanau, D. & Bengio, Y. (2014), On the properties of neural machine translation: Encoder–decoder approaches, *in* 'Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation', Association for Computational Linguistics, Doha, Qatar, pp. 103–111.
URL: *https://www.aclweb.org/anthology/W14-4012*

Clevert, D.-A., Unterthiner, T. & Hochreiter, S. (2015), 'Fast and accurate deep network learning by exponential linear units (elus)', *CoRR* **abs/1511.07289**.

Dabbura, I. (2017), 'Gradient descent algorithm and its variants'.
URL: *https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3*

Dayan, P. & Abbott, L. F. (2005), *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, The MIT Press.

Do, C. B. (2008), 'More on multivariate gaussians'. `http://cs229.stanford.edu/section/more_on_gaussians.pdf`.

Domingos, P. (2012), 'A few useful things to know about machine learning', *Commun. ACM* **55**(10), 78–87.
URL: *http://doi.acm.org/10.1145/2347736.2347755*

Donahue, C., McAuley, J. & Puckette, M. (2018), Adversarial audio synthesis, *in* 'ICLR 2019'.

Elton, D., Boukouvalas, Z., D. Fuge, M. & W. Chung, P. (2019), 'Deep learning for molecular design - a review of the state of the art', *Molecular Systems Design Engineering* .

Ertl, P. & Schuffenhauer, A. (2009), 'Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions', *Journal of cheminformatics* **1**, 8.

Fedus, W., Goodfellow, I. J. & Dai, A. M. (2018), 'Maskgan: Better text generation via filling in the
', *ArXiv* **abs/1801.07736**.

Gillet, V. (2013), 'Ligand-based and structure-based virtual screening', University Lecture.

Glorot, X. & Bengio, Y. (2010), 'Understanding the difficulty of training deep feedforward neural networks', *Journal of Machine Learning Research - Proceedings Track* **9**, 249–256.

Gong, M., Xu, Y., Li, C., Zhang, K. & Batmanghelich, K. (2019), 'Twin auxiliary classifiers gan'.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Goodfellow, I. J. (2016), 'Nips 2016 tutorial: Generative adversarial networks', *ArXiv* **abs/1701.00160**.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. (2014), Generative adversarial nets, *in* Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence & K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems 27', Curran Associates, Inc., pp. 2672–2680.
URL: *http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf*

Graves, A. (2008), Supervised sequence labelling with recurrent neural networks, *in* 'Studies in Computational Intelligence'.

Graves, A. (2013), 'Generating sequences with recurrent neural networks', *ArXiv* **abs/1308.0850**.

Guimaraes, G. L., Sanchez-Lengeling, B., Farias, P. L. C. & Aspuru-Guzik, A. (2017), 'Objective-reinforced generative adversarial networks (organ) for sequence generation models', *ArXiv* **abs/1705.10843**.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. & Courville, A. (2017), Improved training of wasserstein gans, *in* 'Proceedings of the 31st International Conference on Neural Information Processing Systems', NIPS'17, Curran Associates Inc., USA,

pp. 5769–5779.
URL: *http://dl.acm.org/citation.cfm?id=3295222.3295327*

Gupta, A. & Zou, J. (2019), 'Feedback gan for dna optimizes protein functions', *Nature Machine Intelligence* **1**, 105–111.

Guzel Turhan, C. & Bilge, H. (2018), Recent trends in deep generative models: a review.

Gómez-Bombarelli, R., Duvenaud, D., Miguel Hernández-Lobato, J., Aguilera-Iparraguirre, J., D. Hirzel, T., P. Adams, R. & Aspuru-Guzik, A. (2016), 'Automatic chemical design using a data-driven continuous representation of molecules', *ACS Central Science* **4**.

H. S. Segler, M., Kogej, T., Tyrchan, C. & P. Waller, M. (2017), 'Generating focused molecule libraries for drug discovery with recurrent neural networks', *ACS Central Science* **4**.

Hastie, T., Tibshirani, R. & Friedman, J. (2001), *The Elements of Statistical Learning*, Springer Series in Statistics, Springer New York Inc., New York, NY, USA.

Heller, S. R., McNaught, A., Pletnev, I., Stein, S. & Tchekhovskoi, D. (2015), 'InChI, the IUPAC International Chemical Identifier', *Journal of Cheminformatics* **7**(1), 23+.
URL: *http://dx.doi.org/10.1186/s13321-015-0068-4*

Herrmann, V. (2017), 'Wasserstein gan and the kantorovich-rubinstein duality'.
URL: *https://vincentherrmann.github.io/blog/wasserstein/*

Hinton, G. (2012), 'Neural networks for machine learning: Lecture 6e - rmsprop, divide the gradient by a running average of its recent magnitude'.
URL: *https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture$_s$lides$_l$ec6.pdf*

Hochreiter, S. & Schmidhuber, J. (1997), 'Long short-term memory', *Neural computation* **9**, 1735–80.

Huang, T.-W. (2017), 'tensorboardX: A module for visualization with tensorboard for Pytorch'. [Online; accessed 04.09.2019].
URL: *https://github.com/lanpa/tensorboardX*

Hui, J. (2018), 'Gan - wasserstein gan wgan-gp'.
URL: $https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490$

Hunter, J. D. (2007), 'Matplotlib: A 2d graphics environment', *Computing in Science & Engineering* **9**(3), 90–95.

Ioffe, S. & Szegedy, C. (2015), Batch normalization: Accelerating deep network training by reducing internal covariate shift, *in* 'Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37', ICML'15, JMLR.org, pp. 448–456.
URL: *http://dl.acm.org/citation.cfm?id=3045118.3045167*

Irwin, J. & Shoichet, B. (2005), 'Zinc a free database of commercially available compounds for virtual screening', *Journal of chemical information and modeling* **45**, 177–82.

Jensen, J. (2019), 'Graph-based genetic algorithm and generative model/monte carlo tree search for the exploration of chemical space', *Chemical Science* **10**.

Kadurin, A., Nikolenko, S. I., Khrabrov, K., Aliper, A. & Zhavoronkov, A. (2017), 'drugan: An advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico.', *Molecular pharmaceutics* **14 9**, 3098–3104.

Karpathy, A. (2015), 'The unreasonable effectiveness of recurrent neural networks'.
URL: *http://karpathy.github.io/2015/05/21/rnn-effectiveness/*

Khapra, M. (2019), 'Cs7015 (deep learning) : Lecture 7 autoencoders and relation to pca, regularization in autoencoders, denoising autoencoders, sparse autoencoders, contractive autoencoders'.
URL: *https://www.cse.iitm.ac.in/ miteshk/CS7015/Slides/Handout/Lecture7.pdf*

Kingma, D. P. & Ba, J. (2014), 'Adam: A method for stochastic optimization', *CoRR* **abs/1412.6980**.

Kingma, D. P. & Welling, M. (2013), 'Auto-encoding variational bayes'. cite arxiv:1312.6114.
URL: *http://arxiv.org/abs/1312.6114*

Kodali, N., Abernethy, J. D., Hays, J. & Kira, Z. (2018), On convergence and stability of gans.

Landrum, G. (2006), 'Rdkit: Open-source cheminformatics'.

LeCun, Y., Bengio, Y. & Hinton, G. (2015), 'Deep learning', *Nature* **521**(7553), 436–444.

Lin, J. (1991), 'Divergence measures based on the shannon entropy', *IEEE Transactions on Information Theory* **37**(1), 145–151.

Makarychev, Y. (2015), 'Basic properties of metric and normed spaces'.
URL: *https://ttic.uchicago.edu/ yury/courses/geom2015/notes/metric.pdf*

Mendez, D., Gaulton, A., Bento, A. P., Chambers, J., De Veij, M., Félix, E., Magariños, M., Mosquera, J., Mutowo, P., Nowotka, M., Gordillo-Marañón, M., Hunter, F., Junco, L., Mugumbate, G., Rodriguez-Lopez, M., Atkinson, F., Bosc, N., Radoux, C., Segura-Cabrera, A., Hersey, A. & Leach, A. (2018), 'ChEMBL: towards direct deposition of bioassay data', *Nucleic Acids Research* **47**(D1), D930–D940.
URL: *https://doi.org/10.1093/nar/gky1075*

Mirza, M. & Osindero, S. (2014), 'Conditional generative adversarial nets', *ArXiv* **abs/1411.1784**.

Miyato, T. & Koyama, M. (2018), 'cgans with projection discriminator'.

Mogren, O. (2016), 'C-rnn-gan: Continuous recurrent neural networks with adversarial training', *ArXiv* **abs/1611.09904**.

Morgan, H. L. (1965), 'The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service.', *Journal of Chemical Documentation* **5**(2), 107–113.
URL: *https://doi.org/10.1021/c160017a018*

Nair, V. & Hinton, G. E. (2010), Rectified linear units improve restricted boltzmann machines, *in* 'Proceedings of the 27th International Conference on International Conference on Machine Learning', ICML'10, Omnipress, USA, pp. 807–814.
URL: *http://dl.acm.org/citation.cfm?id=3104322.3104425*

Nielsen, F. (2010), 'A family of statistical symmetric divergences based on jensen's inequality', *CoRR* **abs/1009.4004**.
URL: *http://dblp.uni-trier.de/db/journals/corr/corr1009.htmlabs-1009-4004*

Nielsen, M. A. (2018), 'Neural networks and deep learning'.
URL: *http://neuralnetworksanddeeplearning.com/chap2.html*

Odena, A., Olah, C. & Shlens, J. (2017), Conditional image synthesis with auxiliary classifier gans, *in* 'Proceedings of the 34th International Conference on Machine Learning - Volume 70', ICML'17, JMLR.org, pp. 2642–2651.
URL: *http://dl.acm.org/citation.cfm?id=3305890.3305954*

Olah, C. (2015), 'Understanding lstm networks'.
URL: *http://colah.github.io/posts/2015-08-Understanding-LSTMs/*

Olivecrona, M., Blaschke, T., Engkvist, O. & Chen, H. (2017), 'Molecular de-novo design through deep reinforcement learning', *Journal of Cheminformatics* **9**(1), 48.
URL: *https://doi.org/10.1186/s13321-017-0235-x*

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017), 'Automatic differentiation in pytorch'.

Pearlman, R. (1987), 'Rapid generation of high quality approximate 3d molecular structures', *Chem. Des. Automa* pp. 5–7.

Polishchuk, P., Madzhidov, T. & Varnek, A. (2013), 'Estimation of the size of drug-like chemical space based on gdb-17 data', *Journal of computer-aided molecular design* **27**.

Polykovskiy, D., Zhebrak, A., Sanchez-Lengeling, B., Golovanov, S., Tatanov, O., Belyaev, S., Kurbanov, R., Artamonov, A., Aladinskiy, V., Veselov, M., Kadurin, A., Nikolenko, S., Aspuru-Guzik, A. & Zhavoronkov, A. (2018), 'Molecular Sets (MOSES): A Benchmarking Platform for Molecular Generation Models', *arXiv preprint arXiv:1811.12823* .

Popova, M., Isayev, O. & Tropsha, A. (2018), 'Deep reinforcement learning for de novo drug design', *Science Advances* **4**(7), eaap7885.
URL: *http://dx.doi.org/10.1126/sciadv.aap7885*

Preuer, K., Renz, P., Unterthiner, T., Hochreiter, S. & Klambauer, G. (2018), 'Fréchet chemnet distance: A metric for generative models for molecules in drug discovery.', *Journal of chemical information and modeling* **58 9**, 1736–1741.

Prykhodko, O., Johansson, S., Kotsias, P.-C., Bjerrum, E., Engkvist, O. & Chen, H. (2019), 'A de novo molecular generation method using latent vector based generative adversarial network'.

Ringnér, B. (2009), 'The law of the unconscious statistician'. URL: `http://www.maths.lth.se/matstat/staff/bengtr/mathprob/unconscious.pdf`.

Rogers, D. & Hahn, M. (2010), 'Extended-connectivity fingerprints', *Journal of chemical information and modeling* **50 5**, 742–54.

Ruder, S. (2016), 'An overview of gradient descent optimization algorithms.'. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam.
URL: *http://arxiv.org/abs/1609.04747*

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning Representations by Back-propagating Errors', *Nature* **323**(6088), 533–536.
URL: *http://www.nature.com/articles/323533a0*

S, S. & Thilak Chaminda, H. (2017), Generate bioinformatics data using generative adversarial network: A review.

Sanchez, B., Outeiral, C., L Guimaraes, G. & Aspuru-Guzik, A. (2017), 'Optimizing distributions over molecular space. an objective-reinforced generative adversarial network for inverse-design chemistry (organic)'.

Santambrogio, F. (2015), *Wasserstein distances and curves in the Wasserstein spaces*, Springer International Publishing, Cham, pp. 177–218.
URL: *https://doi.org/10.1007/978-3-319-20828-2*$_5$

Schmidhuber, J. (2014), ‘Deep learning in neural networks: An overview’, *CoRR* **abs/1404.7828**.
URL: *http://arxiv.org/abs/1404.7828*

Schneider, G. (2019), ‘Mind and machine in drug design’, *Nature Machine Intelligence* **1**.

Schwalbe-Koda, D. & Gómez-Bombarelli, R. (2019), ‘Generative models for automatic chemical design’, *arXiv preprint arXiv:1907.01632* .

Sriperumbudur, B., Fukumizu, K., Gretton, A., Schölkopf, B. & Lanckriet, G. (2012), ‘On the empirical estimation of integral probability metrics’, *Electronic Journal of Statistics* **6**, 1550–1599.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), ‘Dropout: A simple way to prevent neural networks from overfitting’, *J. Mach. Learn. Res.* **15**(1), 1929–1958.
URL: *http://dl.acm.org/citation.cfm?id=2627435.2670313*

Suki (2017), ‘Learning rate schedules and adaptive learning rate methods for deep learning’.
URL: *https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1*

Sutskever, I., Vinyals, O. & Le, Q. V. (2014), Sequence to sequence learning with neural networks, *in* ‘Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2’, NIPS’14, MIT Press, Cambridge, MA, USA, pp. 3104–3112.
URL: *http://dl.acm.org/citation.cfm?id=2969033.2969173*

Sutton, R. & Barton, A. (1998), *Reinforcement learning: an introduction*, MIT Press, Cambridge.

Theis, L., van den Oord, A. & Bethge, M. (2015), ‘A note on the evaluation of generative models’.

Van Rossum, G. & Drake Jr, F. L. (1995), *Python tutorial*, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Viehmann, T. (2017), ‘More improved training of wasserstein gans and dragan’.
URL: *https://lernapparat.de/more-improved-wgan/*

Villani, C. (2008), *Optimal transport – Old and new*, Vol. 338, pp. xxii+973.

Vondrick, C., Pirsiavash, H. & Torralba, A. (2016), 'Generating videos with scene dynamics', *ArXiv* **abs/1609.02612**.

Wang, Y., Bryant, S. H., Cheng, T., Wang, J., Gindulyte, A., Shoemaker, B. A., Thiessen, P. A., He, S. & Zhang, J. (2017), Pubchem bioassay: 2017 update, *in* 'Nucleic Acids Research'.

Wei, X., Gong, B., Liu, Z., Lu, W. & Wang, L. (2018), Improving the improved training of wasserstein gans: A consistency term and its dual effect., *in* 'ICLR (Poster)', OpenReview.net.
URL: *http://dblp.uni-trier.de/db/conf/iclr/iclr2018.htmlWeiGL0W18*

Weininger, D. (1988), 'Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules', *Journal of Chemical Information and Computer Sciences* **28**(1), 31–36.
URL: *https://pubs.acs.org/doi/abs/10.1021/ci00057a005*

Weiss, K., Khoshgoftaar, T. & Wang, D. (2016), 'A survey of transfer learning', *Journal of Big Data* **3**.

Weng, L. (2018), 'From autoencoder to beta-vae', *lilianweng.github.io/lil-log* .
URL: *http://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html*

Willett, P., Barnard, J. M. & Downs, G. M. (1998), 'Chemical similarity searching', *Journal of Chemical Information and Computer Sciences* **38**(6), 983–996.
URL: *https://doi.org/10.1021/ci9800211*

Winter, R., Montanari, F., Noé, F. & Clevert, D.-A. (2018), 'Learning continuous and data-driven molecular descriptors by translating equivalent chemical representations'.

Winter, R., Montanari, F., Steffen, A., Briem, H., Noé, F. & Clevert, D.-A. (2019), 'Efficient multi-objective molecular optimization in a continuous latent space', *Chemical Science* .

Xiong, W., Luo, W., Ma, L., Liu, W. & Luo, J. (2017), 'Learning to generate time-lapse videos using multi-stage dynamic generative adversarial networks'.

Xu, B., Wang, N., Chen, T. & Li, M. (2015), 'Empirical evaluation of rectified activations in convolutional network', *ArXiv* **abs/1505.00853**.

Yu, L., Zhang, W., Wang, J. & Yu, Y. (2017), 'Seqgan: Sequence generative adversarial nets with policy gradient', *ArXiv* **abs/1609.05473**.

Zhou, Z., Kearnes, S., Li, L., Zare, R. & Riley, P. (2019), 'Optimization of molecules via deep reinforcement learning', *Scientific Reports* **9**, 10752. 10.1038/s41598-019-47148-x.

Acknowledgement

At this point I would like to thank everyone who supported and motivated me during the preparation of this work.

First of all I would like to thank Prof. Dr. Ulrich Mansmann and Dr. Roman Hornung for the opportunity to work on this interesting topic as part of a master's thesis and for their supervision and assessment of this thesis.

I would also like to thank the Bayer AG and in particular Dr. Djork-Arné Clevert for enabling me to write my master's thesis as an accompanying research internship. The work in the Machine Learning Research Group was very instructive and the stimulating discussions on the theory of neural networks and its application in cheminformatics were very interesting, as I do not have a chemistry background.
Also, I would like to thank Santiago Villalba and Robin Winter from the Machine Learning Research Group who both supported me during the research internship with inspiring advices to efficiently implement clean programming code as well as explanations on some parts on unsupervised learning theory.

Last but not least I would like to thank my family and friends for the proofreading of my master thesis.

Satutory declaration

I hereby confirm that I composed the present thesis with the title

De novo drug design in continuous space

independently and that I have used no other sources other than those cited in the text. The text passages which are taken from other works in wording or meaning I have identified in each individual case by stating the source. This applies also to all graphics, drawings, maps and images included in the thesis. Neither this, nor a similar work, has been published or presented to an examination committee.

Tuan Le	Date