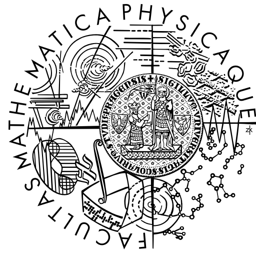

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Adam Frey

Predicting novel drug-target interactions via deep learning techniques

Department of Software Engineering

Supervisor of the master thesis: Mgr. Ladislav Peška, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

I would like to thank my thesis supervisor, Mgr. Ladislav Peška, Ph.D., for his guidance and helpful advice during the writing of this work.

Title: Predicting novel drug-target interactions via deep learning techniques

Author: Adam Frey

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., Department of Software Engineering

Abstract: Aim of this work was to develop a machine-learning model for a prediction of drug-target interactions. Inspired by previous state-of-the-art approaches, the work focuses on collaborative filtering methods and deep learning neural network models. The goal of improving upon the previous work was achieved using a series of improvements of a basic latent matrix factorization algorithm on the relevant dataset. The small amount of data currently seems like the bottleneck for utilizing more sophisticated deep learning methods. As such hybrid approaches for recommendation systems can prove to be interesting next step due to their effective utilization of multiple data sources.

Keywords: drug-target interaction, machine learning, deep learning

Contents

1 Introduction **2**

Introduction **2**

2 Drug Target Interaction **4**
2.1 Overview 4
2.2 Molecular docking 5
2.3 Dataset 9

3 Machine Learning **16**
3.1 Machine Learning Background 16
3.1.1 Empirical Error vs. Generalization Error 18
3.1.2 Overfitting and Bias 21
3.2 Neural Network and Deep Learning 23
3.2.1 Multi-layered neural network 25
3.2.2 Backpropagation 27
3.2.3 Convolutional networks 32
3.3 Model Evaluation 35
3.3.1 Methods for Classification 35
3.3.2 Methods for Regression 41
3.3.3 Cross-Validation 42
3.4 Previous work 44
3.4.1 Neighborhood Regularized Logistic Matrix Factorization for Drug-Target Interaction Prediction [Liu et al. [2016]] . 45
3.4.2 Drug repositioning by integrating target information through a heterogeneous network model [Wang et al. [2014] 47
3.4.3 NeoDTI: Neural integration of neighbor information from a heterogeneous network for discovering new drug-target interactions [Xiao et al. [2018]] 49

4 Model **53**
4.1 Considerations and influences 53
4.2 Modified logistic matrix factorization 54
4.3 Penalized drug-target error 55
4.4 Adding descriptors 58
4.5 Transformation layers 59
4.6 Final evaluation 61
4.7 Experiment details 63

5 Conclusion **65**

Conclusion **65**

Bibliography **66**

List of Figures **72**

1. Introduction

The price of drug research has been steadily increasing in the past. As such, it is often not financially viable to develop a brand new drug for each particular disease. In these cases it can be possible to use existing tested drugs with additional desirable effects. Discovery of these agents is not trivial, although there are several possible approaches to this issue.

One group of these methods employs machine learning. This approach is less accurate than some of its alternatives, but it is arguably less resource-intensive, and can provide probable candidates for focused search using more accurate, albeit more expensive methods. The aim of this thesis is to create a machine learning solution for discovery of interacting drug-target pairs that would improve upon previously published models. Focus is mostly put on neural networks and the so-called deep learning.

Chemical compounds used as drugs in medicine can have unforeseen side effects. It is not only due to negative cases like thalidomide, which has caused birth defects [Kim and Scialli [2011]], that drug development has grown ever more expensive. Generally the efficiency of drug discovery has been in steady decline for the last 60 years, and one could say that the low-hanging fruit was already picked [Scannell et al. [2012]].

Some studies estimate the price of a new drug development to be around 2.6 billion dollars [Dimasi et al. [2016]]. Such costs limit the probability of drug detection for less frequent or less severe diseases, and in these cases a different approach needs to be taken. For example an already developed drug might be used that positively influences given condition. This process is called *drug re-positioning*. Unfortunately, as stated before, side-effects for a given drug are not so easily found.

There are three basic approaches used for drug re-positioning, each with its own advantages and disadvantages. The first possibility is to perform in-vivo or in-vitro experiments, and search for desirable effects. While this is certainly the most accurate method, it is also the most time-consuming and the most financially demanding one. It is worth to say that while toxicity is a factor when testing new drugs, this issue is already eliminated before drug re-positioning. Originally both in-vivo and in-vitro methods were indirectly the only methods used as the discovery of drug's secondary effects was almost always accidental. For example minoxidil was originally aimed to treat ulcers, but was subsequently also found to decrease blood pressure and limit hair loss [Talevi [2018]].

The second approach consists of simulating the interactions between the structure of a drug molecule and that of the target protein. It is called *molecular docking* [Meng et al. [2011]], and it achieves accurate results without the need to perform direct experiments. Its downside is the need for structural description of both actors. This is not always known, and as such the utility of docking is limited.

Last but not least is the family of machine learning approaches. These use descriptors of both proteins and drugs as well as already existing information about interacting pairs. In contrast to structural information in case of docking,

descriptors used by these methods are more ubiquitous. In comparison to actual experiments, it is far more time efficient as a single model can be used to predict interactions among a large group of targets and drugs. A downside is a lower accuracy than both previously noted options.

While drug repositioning as a whole rose in importance, "machine learning", a data-driven approach for computer behaviour inferred from presented patterns, also gained a large traction. The term machine learning denotes a family of approaches that build a model based on provided data. In many cases is this model then used for prediction. Previously predictors needed to be created by a human, typically a domain expert. Models created by a machine learning approach on the other hand do not need such a level of supervision. This allows for a faster development as well as creation of models that would be too complex or too laborious for people to create. Additionally, machine learning may be successful in situations where there is no thorough domain knowledge.

Neural networks are a family of machine learning techniques formerly inspired by neuron connections in a brain. Their most important successes have been relatively recent, although neural networks have been studied before. A more powerful hardware allowed for larger and more complicated networks that were then able to achieve high accuracy on previously difficult tasks. Those were for example image recognition [Krizhevsky et al. [2012a]] or machine translation [Sutskever et al. [2014]]. Notable term within the the neural network field is *deep learning*. Deep learning is concerned with neural networks that are created from multiple groups of neurons that together form a sort of a cascade. It is this general architecture that proved to be especially effective for many previously untractable problems.

Neural networks themselves have formed some of the machine learning solutions used for prediction of drug-target interactions. The relevant models are often based on approaches originally utilized for recommender systems with which they share many characterstics. Recommender systems are concerned typically with finding links between prospective buyers and relevant products utilizing previous links. There are other tasks, but here the relation with with drug-target interaction is quite clear. Drugs can be conceptualized as buyers that look for prospective targets in the form of proteins. This correspondence allows for utilization of machine learning models previously used elsewhere.

As for deep learning, some of these approaches are successfully utilized for recommender systems. For drug-target interaction specifically, the paper Xiao et al. [2018] achieves state-of-the-art results while utilizing these general principles.

2. Drug Target Interaction

2.1 Overview

The main reasoning behind the development of computational methods of interaction discovery was already presented in the introduction. The basic demands for this approach stem from the decreasing rate of new drug discovery, which is tightly linked with its increasing costs. A great effort is put into proving that newly-developed drugs are not harmful. While this demands are valid, it makes discovery even more expensive, and thus it puts more strain on re-use of drugs, that are already proved to be safe. This is the goal of drug re-positioning.

While this is a thesis mostly concerned with computer science, the underlying biological and chemical principles used in drug-repositiong cannot be completely ignored or abstracted. The following chapter concerns itself with the domain knowledge and its corresponding assumptions with which machine learning (ML) algorithm works. This chapter aims to introduce basic terminology and relationships, as well as molecular docking, which is another computational approach for discovery of drug-target interactions that aims to illustrate the interaction mechanism itself. Additionally, possible representations are discussed with special emphasis put on data used for training and testing of machine learning models.

The goal of a drug is to cause a physiological change in a body to positively affect its health. A mechanism through which this change happens concerns *receptors* [Farlex [2012]], proteins that receive and transmit signals regulating the function of a biological system. Drug's purpose is to bind to the proper set of receptors, and thus elicit a desirable response from the body. These receptor objectives are often called *targets*.

Organic molecules that bind to specific sites on a cell surface, not only receptors, are called *ligands* [Mosby [2009]]. Drugs form a subset of all ligands, and they are sometimes referenced by this term in the following chapters. Ligands change the biochemical properties of receptors by binding with them. More on this in section 2.2. As not all ligands interact with all receptors, the goal of drug re-positioning is to find actual interactions that may be utilized in medicine.

The already presented terms *in vivo* and *in vitro* methods describe experiment performed within a living body and within an artificial environment respectively. *In silico* on the other hand refers to the silicone within a CPU, and thus the term corresponds to experiments simulated on a computer. Both molecular docking and machine learning approaches are in silico.

There is one more concept that is useful to mention. Originally defined only for research paper search, *Swanson's ABC model* [Swanson [1991]] aims to describe a mechanism through which new hypotheses about connections between subjects can be made. It builds on the assumption that if there exists a known connection between concept A and B and another known connection between B and C, then that increases the probability of a connection between A and C. Together these three points A, B, and C form a triangle where two edges AB and BC are known while the existence of AC is implied. Swanson refers to this principle as "logic of suggestibility". The model is concerned with relations defined by research papers which form a knowledge base called "complementary but disjoint

structures within the literature of science" [Swanson [1991]]. Disjoint implies that these two structures of articles do not reference each other, are not co-cited, and have no overlap.

An example is a newly stated hypothesis linking *migraine* (A) to *magnesium* (C). Eleven indirect connections are defined to support this hypothesis. One of them uses as an intermediate link *stress and type A behaviour.* The corresponding connections are that stress and type A behaviour are associated with migraines, and that stress and type A behaviour lead to body loss of magnesium. Another pair of links builds on the transitional step of *vascular tone and reactivity.* Here the links are formed by statements claiming that excessive vascular tone and reactivity may increase susceptibility to migraine, and that magnesium can reduce vascular tone and reactivity. Stress and type A behaviour and vascular tone and reactivity represent the B nodes of the ABC model.

Originally, Swanson defined the concept only for scientific literature, and thus this method was dependent on text mining. But assuming that intermediate connections are already explicit and encoded in a graph, there is no more need for further data mining. ABC model applied to interactions or relations between receptors, ligands, and possibly other actors is mentioned for example in Talevi [2018].

2.2 Molecular docking

The following section is concerned with computational docking. This is a method for discovery of novel interactions between ligands and targets, typically proteins. It uses information about the 3D structure of these individual objects in order to find possible configurations in which they fit together.

This approach is quite successful, but can be used only in situations where the corresponding 3D structures are known. This is the case only in a fraction of cases [Prokop [2015]], which reduces docking utility quite a bit. Additionally when this information is known, docking generally tends to outperform methods based on machine learning. As such the choice of dedicating a section to docking in work centered around deep learning may seem peculiar. The main reason for focusing on this topic is that different variants of docking provide a good overview of the underlying mechanisms without introducing an excessive amount of notation from chemistry or biology. In other words, it presents the topic of drug and target interaction using terminology rooted in computer science.

Molecular docking relies on simulating the relationship described by theoretical models of actual biological processes. These are generally concerned with the possible states of individual actors, and the conditions under which they interact. In history, models describing molecular interactions have been evolving and getting more precise. Typically, that meant progress from simple hypotheses and methods to ones with fewer invariants and greater complexity. Similarly, algorithms for computational docking built on these models have grown more complex.

A simple description of interaction between targets and ligands presents the *lock and key* model [Fischer [1894]]. It assumes that the structure of both target and the binding ligand is constant and rigid. It uses a metaphor of a physical lock that can be opened only by a key of certain shape. Target is analogous to a

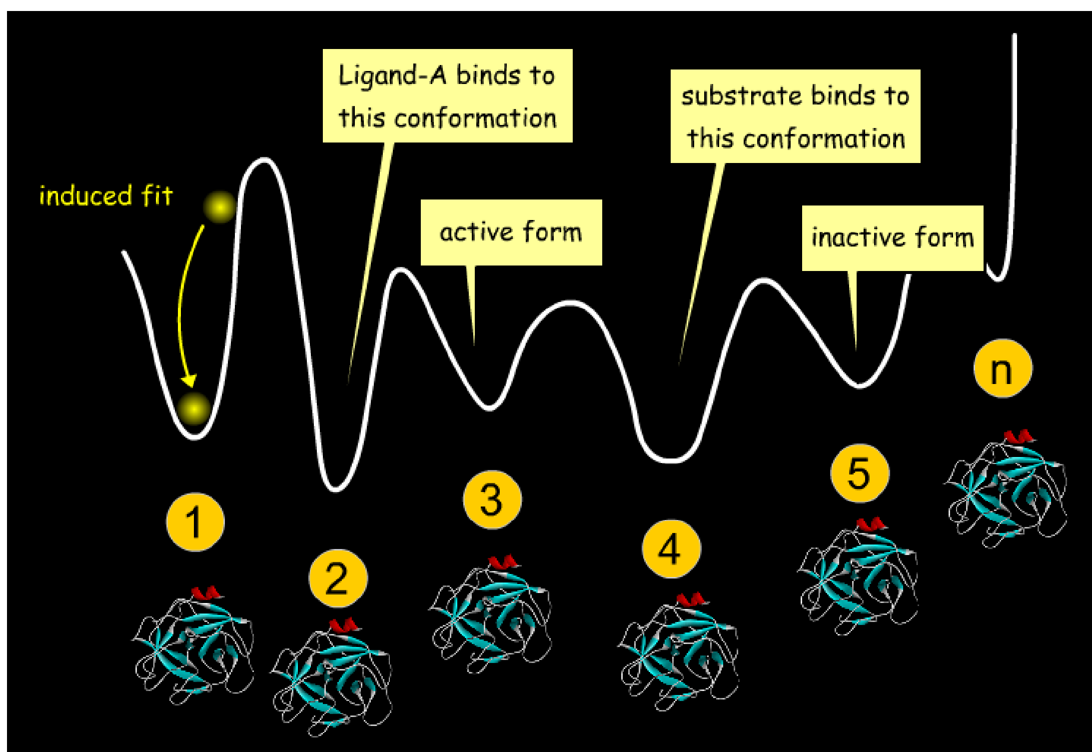

Figure 2.1: Several conformations of a protein with corresponding potential energy. Note the yellow arrow representing change caused by induced fit. Source: dti

lock, while ligand is analogous to a key. Together they interact if and only if there exists a part of surface of the protein the ligand can latch and connect onto. For a key to enter a lock, it needs to be of a correct shape. For a successful molecular interaction it is crucial to decrease the potential energy of the complex.

Alternatively, the *induced fit* model [Koshland [1958]] presents a more complex explanation of drug-target interaction. The metaphorical lock and key described by the previous model both remain rigid through the whole process. To be more precise, that means that the *conformation*, the relative spatial arrangement of atoms within one of these objects remains the same. Such a description does not account for the external forces that in reality modify the shape of a molecule. Induced fit model on the other hand assumes that atoms within a single molecule do not perfectly retain their relative positions, but instead allows for small shifts. To return to the previous metaphor, both the lock and the key are now partially flexible and can thus accommodate greater number of complementary shapes. This approximates the actual biological process more closely, and can thus yield better docking results.

The *conformational ensemble* model [Boehr et al. [2009]] allows for deeper structural changes in proteins. In addition to small induced changes, proteins can exist in multiple structural configurations. As such a protein forms an ensemble of conformational states. Each of these has its own set of possible interacting ligands due to differing conformations.

As for the molecular docking algorithm, there does not exist a single ideal approach that would always yield correct result while satisfying all desirable constraints like time complexity. There are too many possible configurations in which

can interactions occur, and no efficient way to find the correct ones. So molecular docking relies on defining individual conformation states and subsequently searching through them. For each state, one can evaluate the quality of the fit that is based mostly on a potential energy of the unified complex. The complete space of all possible solutions is typically far too large to analyze completely. That creates the need for intelligent search strategies and useful heuristics. The introduction of flexibility in both actors complicates things even further.

To successfully search through possible individual binding modes, that is complex conformations, one needs to define the representation of a molecule. These representations define the searched space. They also form the input for scoring functions that evaluate the quality of the fit.

A frequently used possibility for molecule representation is *atomic representation* [Prokop [2015]]. Individual molecules are here described by a list of atoms, each with its own type and position. Bonds between atoms are also part of this description.

Another form of molecule characterization is the *surface representation*. For this representation, the atoms are not presented directly but instead the whole molecule is covered by a 2D surface that forms its representation. This can be achieved by taking the van der Waals surfaces of all atoms in a molecule and "rolling" a ball over them to get continuous, smooth surface with convex and concave areas [Connolly [1983]]. The advantage of this method is that it makes the various ridges and protrusions explicit, thus making spatial complementarity that plays a role during an interaction assessment easier.

In a *grid-based representation*, one of the actors, typically the receptor is fixed in space. This space is then overlaid with a fine grid. For each vertex of this grid, forces stemming from the position of the receptor are computed. This in the case of a rigid receptor position is done only once, and is thus arguably computationally efficient compared to the previously described representations [Atassi and Appella [1995]]. During actual docking simulation, forces acting on individual atoms of the ligand are then approximated from the nearest points of the grid.

A crucial part of docking is a mechanism, that decides, if a ligand and a receptor in a given configuration interact. More precisely, a mechanism that measures the quality of the fit. In molecular docking, it is called a *scoring function*. This function typically takes into account multiple relations and types of force. These individually can be classified into several groups. Among others these are *force field scoring functions* and *empirical scoring functions* [Jhoti and Leach [2007]]. Notable exceptions are methods based on machine learning [Ballester and Mitchell [2010]] that predict results only from presented data.

Scoring functions based on approximation of force fields use combination of van der Waals interactions, electrostatic interactions, and desolvation energy. To alleviate the potentially high computational costs, energies are often precomputed in the previously introduced grid-based representation of a receptor.

Empirical scoring function is based on the wrong [Mark and van Gunsteren [1994]], but useful assumption that binding energy ΔG can be decomposed into a series of distinct terms each with its own underlying chemical phenomena [Jhoti and Leach [2007]]. ΔG of the complex is defined as an energy needed to split the complex again into its two original components. The higher it is, the stronger

the connection, and thus it is a good measure of the strength of an interaction between a ligand and a protein. In Prokop [2015] it is defined in the following form although its computation using the previously presented scoring functions is typically more complex as one does not know the potential energies of all participants. Energies of ligand and receptor are taken in their unbound state.

$$\Delta G = E_{complex} - E_{ligand} - E_{receptor} \tag{2.1}$$

Hydrogen bonds, hydrophobic, and ionic interactions all belong among the phenomena that influence ΔG, and the weighted sum of all these factors determines the ΔG prediction.

Another crucial component of molecular docking is state space search. Considering at first that both ligand and receptor are rigid, it is possible to fix the position of one of them, typically the receptor. Then the state is defined by the position of the other in space In 3D space that means 3 coordinates for translation and 3 angles for rotation. As search through all possible states is for larger complexes too computationally intensive, a more sophisticated algorithm needs to be chosen. Several possible choices are valid. Simulated Annealing is one of them, so are Monte Carlo methods or approaches based on Genetic Algorithms.

Additionally, *Tabu search* [Glover and Laguna [1998]], a search algorithm restricting usage of previously visited nodes, can be used. There are multiple versions of this algorithm, but the core idea is some form of search for a local optimum in discrete space. This search strategy may be stochastic or not, but the main feature of tabu search is an additional memory designed to store already visited nodes. These are the nodes that are taboo for the algorithm, i.e. they should be avoided. There are again several modifications to the way the taboo memory can be approached. It can be for example finite, or the ban can be just probabilistic, allowing older memorized nodes with higher probability than new ones.

In the simplest form, tabu search keeps a taboo list, a record of the last k nodes that were visited. Whenever a choice is made where to go from the current state s, these nodes are removed from the list of possibilities. Then the oldest node in the taboo list is removed and replaced by s.

In contrast to the iterative state space search, another method used for finding possible interactions is *feature-based matching* [Prokop [2015]]. At first a set of features in a ligand is matched with a complementary set of features of a receptor. Quality of these matchings are then assessed individually. Then both actors are assembled based on the filtered feature pairs, and the overall fit is evaluated.

So far methods were implicitly using rigid forms of molecules. The state of the complex was defined by translation and rotation of the ligand, but no structural changes within individual actors was allowed. This corresponds to the lock and key model, but can be a too severe simplification. On the other hand, adding a parameter for all flexible bonds can make the model computationally intractable. An approximative approach is needed.

Firstly, a distinction has to be made between introducing flexibility to ligands and introducing it to receptors. Ligands are typically smaller, and are thus treated differently. Arguably, one of the easier methods of introducing flexibility on a ligand is introducing a set of rigid forms for each ligand molecule. In this approach, an ensemble of ligand conformations is generated, and then its members

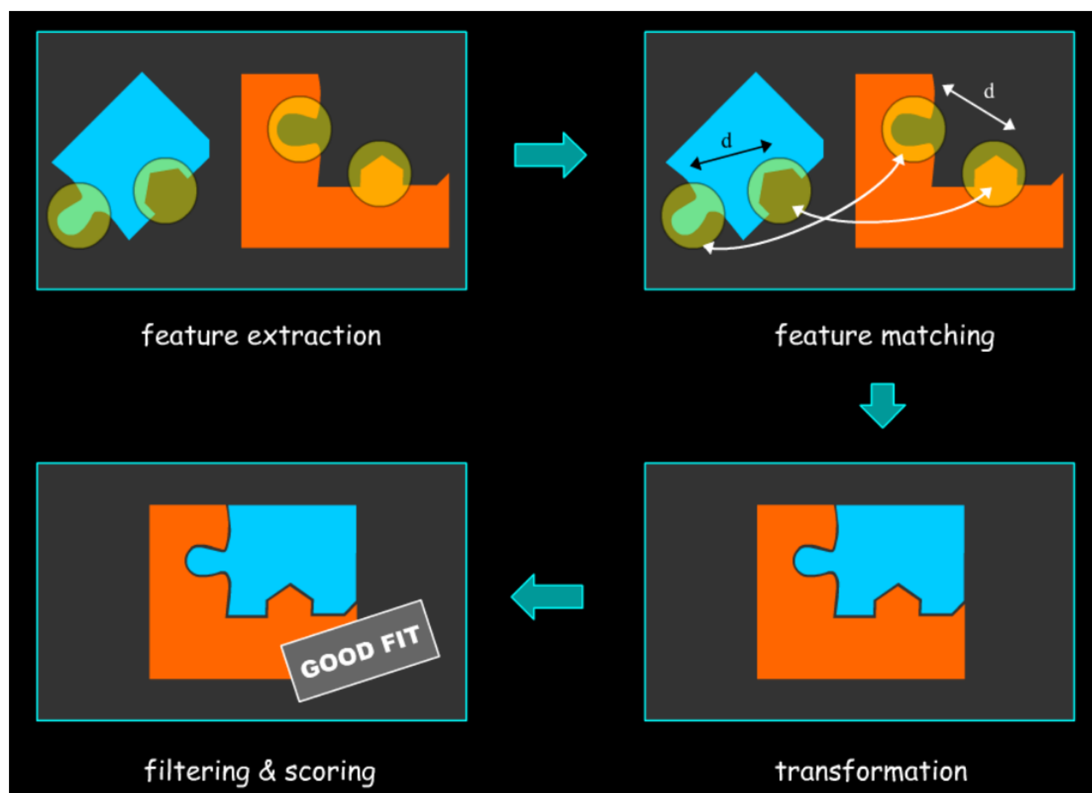

Figure 2.2: Illustration of the feature-based matching process. Source: Prokop [2015]

are used individually in a rigid docking algorithm.

A different set of approaches is based on breaking the original ligand into several pieces. One method tries to fit the distinct fragments individually onto the receptor. If that happens, it then tries to assert if this combination is possible based on the position of all fragments. Another method starts with connecting a single fragment onto the ligand and then successively adds additional pieces, so that they connect to their already placed predecessors.

As for flexible receptors, here is a situation different due to a larger molecular size. One of the approaches is to keep the structure rigid, but introduce more permissive, softer scoring functions. This approach is called *soft docking* [Ferrari et al. [2004]], and its underlying concept is that with certain repulsive functions being inhibited, conformations do in fact interact thanks to receptor flexibility. Van der Waals repulsive force is an example of a force that is here being attenuated.

As a side-note, the previous section omitted one important part of molecular docking. The preparation of molecular models themselves is no trivial matter. Among other activities, hydrogen bonds need to be optimized, atomic clashes need to be removed, and tautomer and ionization states need to be generated [Sastry et al. [2013]]. This adds to the complexity of the process, but at the same time it is beyond the scope of this text.

2.3 Dataset

While molecular docking relies on accurate 3D models, machine learning approaches are less demanding and can utilize greater range of information. Arguably, machine learning achieved considerable achievements even as a component of docking techniques [Khamis et al. [2015]].

Broadly speaking, information used by ML techniques can be classified into two classes. The first one composes of information relating to discrete actors, most notably drugs and receptors. An example is the number of atoms or bonds for a molecule. While this may present useful information, it does not actually provide necessary data for supervised[1] machine learning approaches that are typically used for drug-target interaction. Into second class one could group all data that describe relations between individual actors. The crucial associations are the already known drug-target interactions.

As for the first group, information about molecules are aggregated into so-called *molecular descriptors*. A molecular descriptor "is the final result of a logic and mathematical procedure which transforms chemical information encoded within a symbolic representation of a molecule into a useful number or the result of some standardized experiment" [Todeschini and Consonni [2008]]. There is a range of descriptors computed. An example is a molecular surface area, or number of atoms. Broadly they can be classified into 1D, 2D, and 3D descriptors. Each descriptor can consist of a single property, or it can form an aggregation of multiple properties like 3D-MoRSE [Yap [2011]].

A distinct set of molecular descriptors that does not fit neither the 1D, 2D, or 3D group is the collection of *fingerprints*. A fingerprint is a sequence of typically boolean values that encodes structure and properties of a molecule [Bajorath [2008]]. Its main purpose is to detect pairs of molecules that are not structurally similar. This is important as the general algorithm for substructure search is NP-complete, and is thus too slow for searching through large databases [day [2011]].

There are multiple kinds of software projects used for generating molecular descriptors. They differ in the set of features they generate, their license, or the platform on which they run. *PaDEL-Descriptor* [Yap [2011]] is one of them. It is an open source project written in Java and thus running on all major platforms. It computes 797 descriptors and 10 fingerprints. A software with even greater number of descriptors (5270) is Dragon [Mauri et al. [2006]]. Unfortunately the fact that it is a commercial software is a downside compared to the free PaDEL.

In contrast to information relating to individual molecules or other discrete objects, there are several types of relations that may be used during machine learning. Given the assumption that drug-target interaction is being performed, the most important information are the already discovered interactions. Data about pairs drug-target that are known not to interact are of course also useful, if they are provided. Other possible types of relations are protein-protein interactions among receptors or known drug-disease associations.

For evaluation of introduced machine learning methods a heterogenous dataset was taken from Xiao et al. [2018]. It consists of nodes and edges between them. Nodes here represent discrete entities grouped into 4 classes or types, drugs, pro-

[1] For definition of supervised ML, see 3.1.

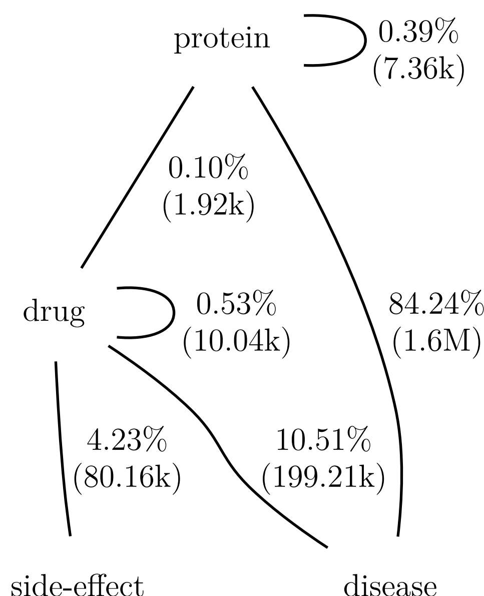

Figure 2.3: Types of nodes in graph each represented as a vertex. The percentages denote the portion of edges of given kind. Number in brackets is the total number of edges of this kind.

teins, diseases and side-effects. Edges then represent known interactions between these objects. No further descriptive informations are provided. This dataset was used to allow comparison between models, namely between the model presented in this work and the already mentioned Xiao et al. [2018].

Altogether data was compiled from 4 sources. As for the nodes, drugs were taken from DrugBank 3.0 [Knox et al. [2011]], proteins from ninth release of HPRD database [Keshava Prasad et al. [2008]], diseases from the 2013 update of Comparative Toxicogenomics Database [Davis et al. [2012]], and side-effects from the second version of SIDER database [Kuhn et al. [2010]]. Concerning the edges, drug-drug interactions are taken from DrugBank, protein-protein interactions from HPRD, disease-related interactions from the Comparative Toxicogenomics Database, and interactions between side-effects and drugs from SIDER. Finally protein-drug connections were extracted also from DrugBank.

All of these sources are publicly and freely available online for non-commercial uses. As an example, DrugBank is a database created in 2006 “containing information on drugs and drug targets” dru. Release 5.1.4 contained 13370 individual drug entries.

Below are several graphs and visualizations meant to familiarize the reader with the used dataset. These focus mostly on the dataset as a graph of interactions or connections. Most notably 2.3 shows the total amounts of edges and their quantity for each type of connection. It can be seen that connections utilizing disease information present a majority of all edges. But not all diseases contribute equally to this number The same applies to side-effects.

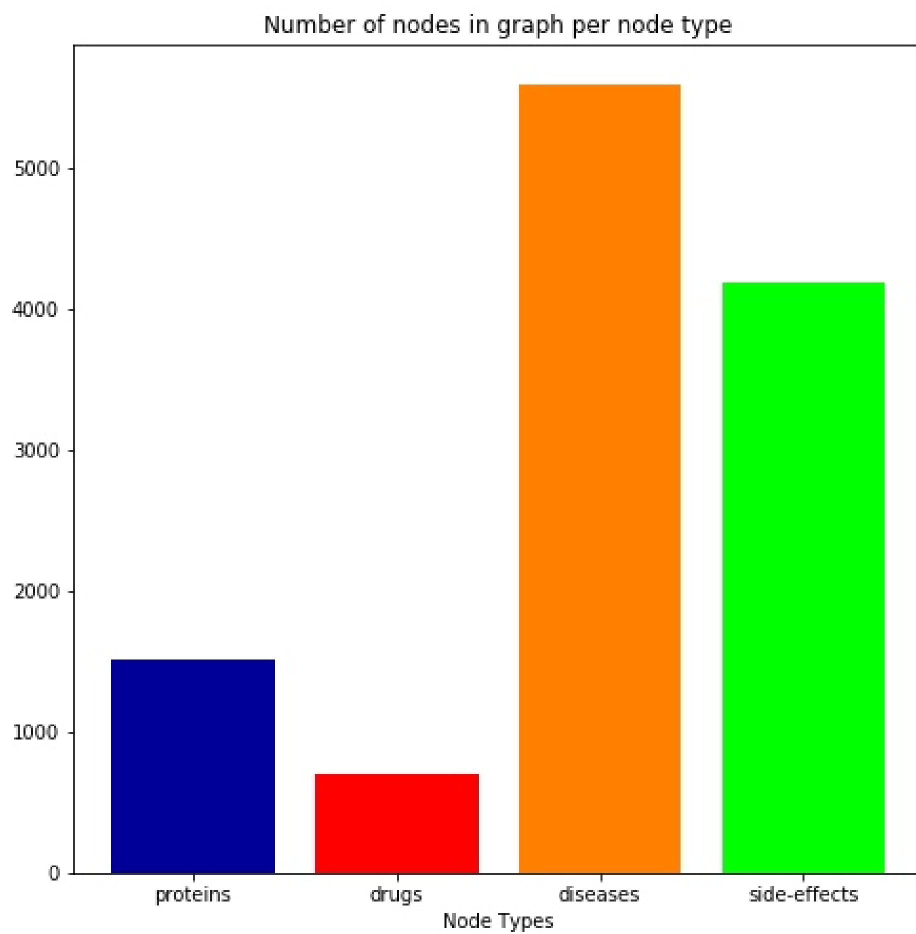

Figure 2.4: Proportion of different types of nodes in the graph.

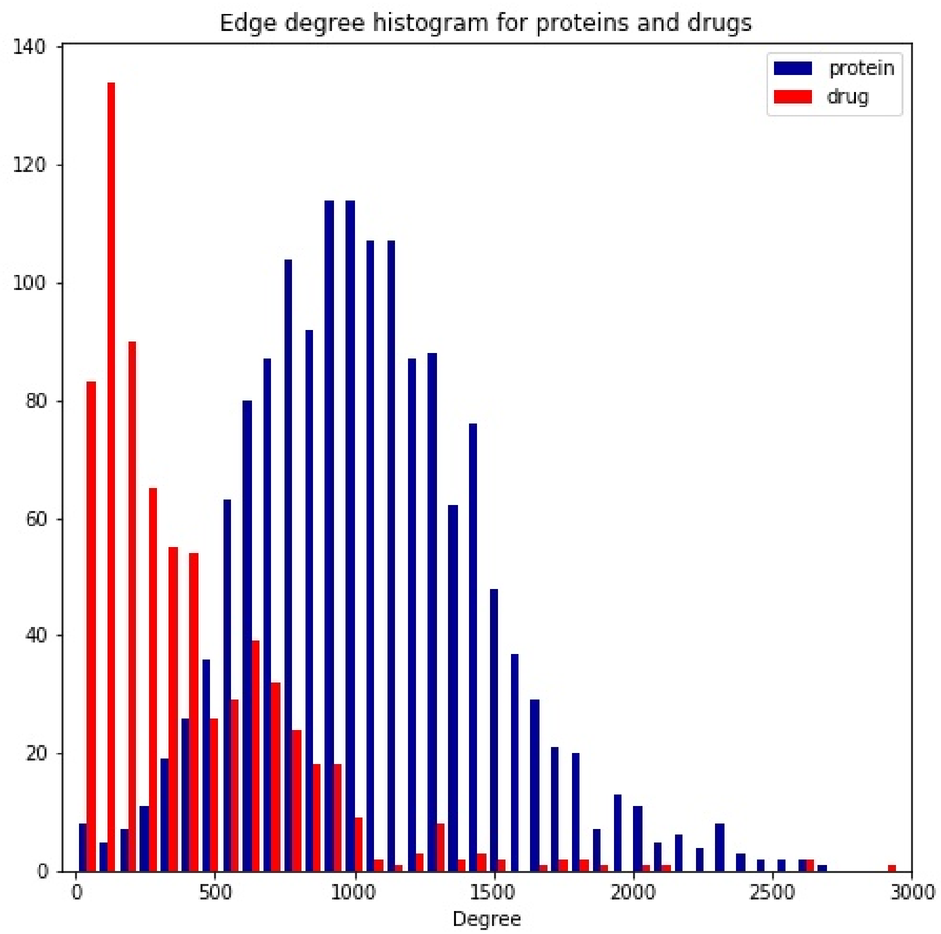

Figure 2.5: Histogram of edge degrees for proteins and drugs.

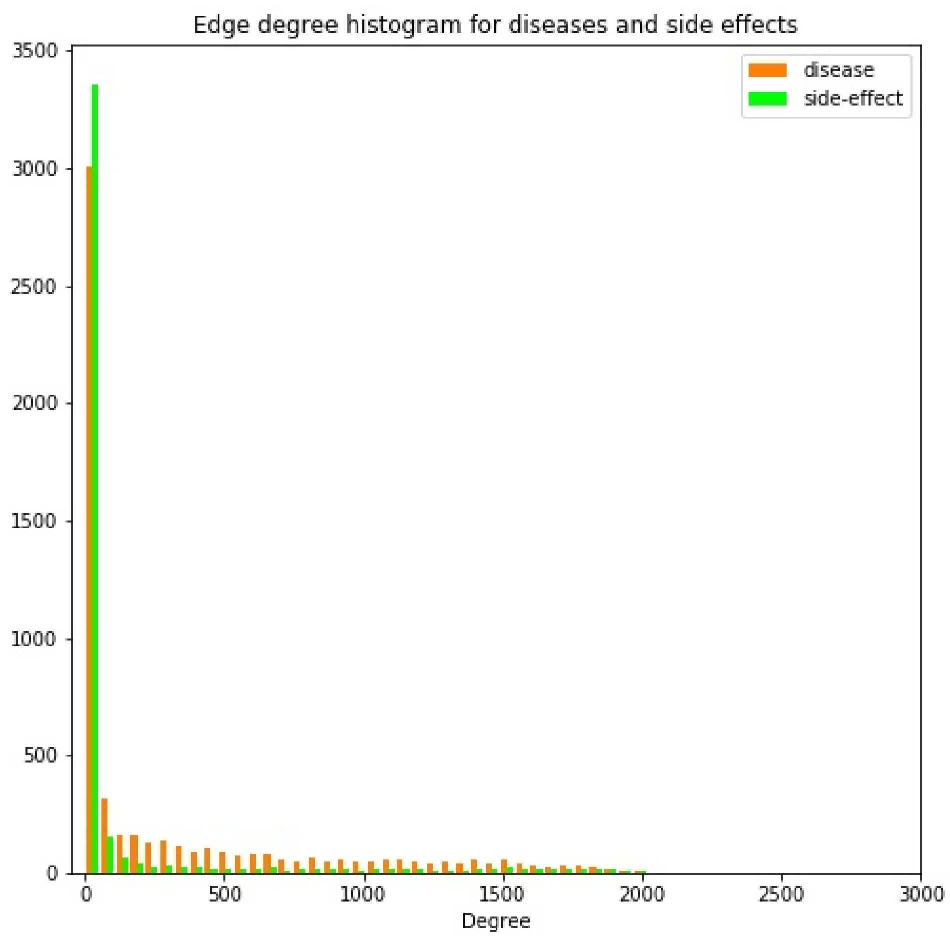

Figure 2.6: Histogram of edge degrees for drugs and side-effects.

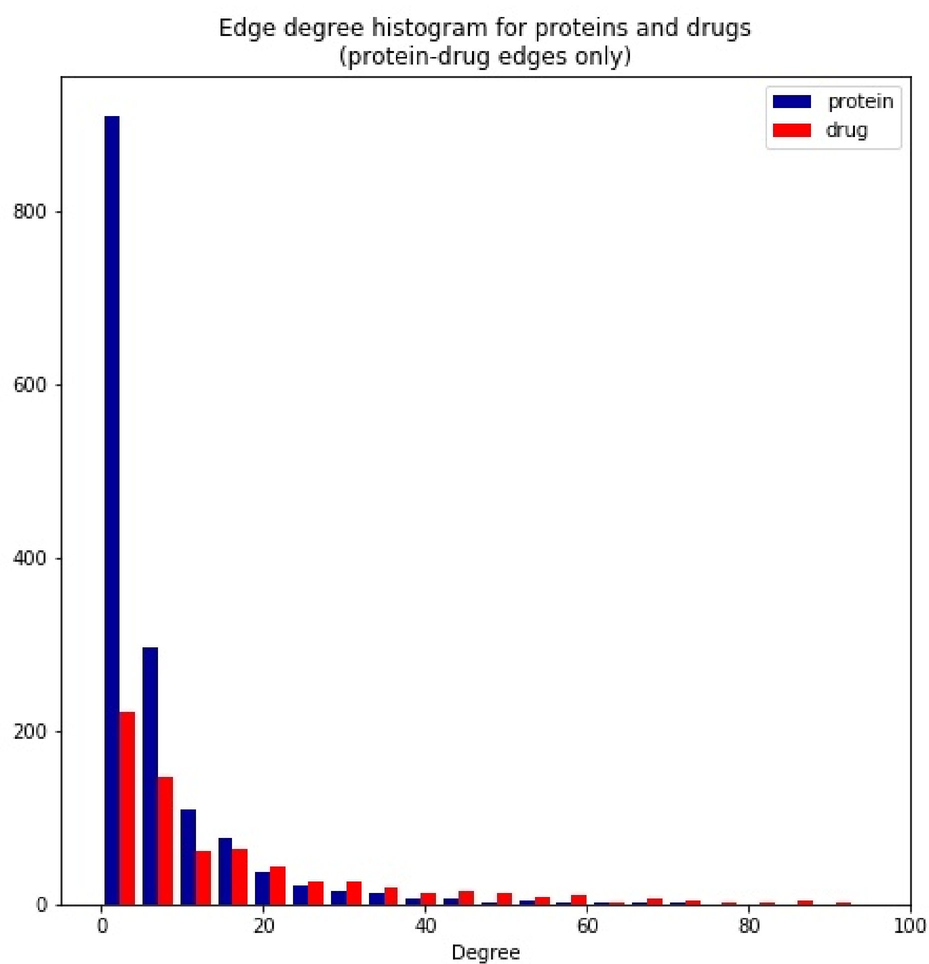

Figure 2.7: Histogram of edge degrees for proteins and drugs on a subgraph with only proteins and drugs.

3. Machine Learning

The following chapter is about machine learning. It aims to introduce the theoretical background centered on machine learning in general and neural networks in more detail. For information about the developed model and corresponding experiments, refer to a chapter 4.

The first section is concerned with a very high-level view of machine learning where individual algorithms are not discussed. It should provide an overview of the terminology used, provide unifying framework for following practical methods as well as assert their limits.

In second section, neural networks, a family of machine learning algorithms, are presented in more detail. Apart from the more basic models, emphasis is put on the so-called *deep learning* models or approaches that were developed relatively recently. Some of the most successful models are discussed together with those that are relevant to the following experiments. The problem of finding an optimal model configuration given model's neuron weights is considered as well.

In addition, a section on model evaluation is also included. It is concerned with the basic methods designed to specify model's performance as well as with the specialized approaches that are more relevant to given problem area.

Several notable machine learning algorithms that have been successfully employed to previously advance the field of drug-target interaction prediction are briefly touched on in the last section of this chapter.

3.1 Machine Learning Background

Machine learning is generally defined as a collection of computational methods geared toward optimization or prediction based on past experiences or previous information [Mohri et al. [2012]]. While this definition is quite abstract, some examples are object recognition, speech-to-text transcription, clustering, virtual agent learning, artificial intelligence for games like chess or go, and recommender systems. This thesis is concerned with prediction of drug-target interactions. This is in itself a quite distinct problem compared to the full breadth of machine learning approaches. For that reason following chapter will be centered on a section of Machine Learning called *supervised learning*. Its goal is to provide some, possibly new information about an object given the object's description.

An algorithm is in the beginning supplied with a sequence of pairs $((x_1, y_1), (x_2, y_2), \dots (x_n, y_n)) \in (X \times Y)^n$. The first element in each pair is typically called a *feature* and provides a characterization of an object often in a form of a vector. The second element the output algorithm should provide given the aforementioned feature is typically called a *label*[1] [Mohri et al. [2012]]. Another way to look at it is that ML algorithm, sometimes also called a *learner*, is provided with questions, and when it produces an answer, the answer is checked and corrected. The learning process is therefore supervised, hence the name.

[1]The term label is here used in its broader sense. Yet often it is defined as a value from a discrete set. Like *dog* or *cat* for image classification. That is in contrast to continuous values for which the term *label* is not used as often.

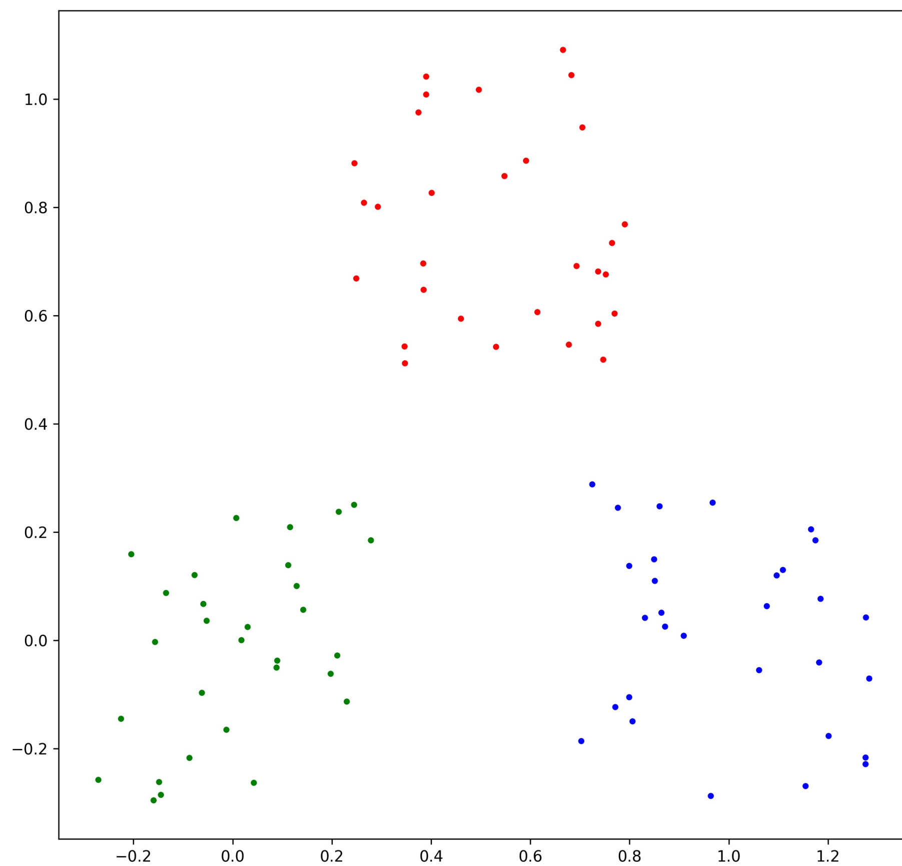

Figure 3.1: A simple example of clustering. An algorithm is provided with a set of unannotated points in $\mathbb{R}^2$. Its task is to divide those points into a set of groups so that members of each group lie close together while points from distinct groups are relatively far apart. Resulting groups are here denoted with colours.

There are other classes of machine learning algorithms, one of them being *unsupervised learning*. In such case, a learner is provided only with features. The goal then typically isn't to provide a response to each object, but rather to provide information about the set of features itself. An example of this approach is *clustering* where elements in a feature space are returned to represent distinct groups in the original dataset. A basic case is shown in figure 3.1.

A bit more exotic group of algorithms is covered by the term *reinforcement learning*. In this scenario, an agent is embedded within an environment with which it interacts. It is not provided with a direct plan of action, but instead it is given a function rewarding it for achieving desirable states of the environment, and punishing it for entering undesirable ones. In this way, its goal is to act on its own to maximize its gain. A concrete example is the artificial intelligence for the game of Go [Silver et al. [2016]].

Returning to supervised learning, a learning algorithm is defined as a function $\mathcal{F}$ that takes a *training set*, a sequence[2] of pairs (feature, label) from a set $X \times Y$. The learning algorithm itself returns a function $f : X \to Y$. This new function is called a *hypothesis*. The range of a learning algorithm $rng(\mathcal{F})$, the set of all possible classifiers, is called a *hypothesis set*. The process of acquiring a hypothesis for given input data is called *training*.

Machine learning tasks are sometimes split into two categories based on the range of their hypotheses, i.e. Y. A task with a finite, discrete range is called *classification*, while task with a continuous range is called *regression*.

3.1.1 Empirical Error vs. Generalization Error

As there is a broad range of classifying functions, it is important to have a mechanism for discriminating between successful and unsuccessful ones. Let's assume an underlying probability distribution $\mathcal{D}$ on set $X \times Y$ and a training sequence $S = ((x_1, y_1), (x_2, y_2), \ldots (x_n, y_n))$ drawn from $\mathcal{D}$ where individual samples are independent and identically distributed. For both of these cases an evaluation function is needed. The resulting classifier should be able to perform well on the whole distribution. In other words it should generalize well. Meanwhile the only information provided to the learner is the sequence S, so performance can be only evaluated empirically using those pairs. The corresponding functions are thus called *generalization loss* and *empirical loss*. Alternativelly they are also called *generalization risk* and *empirical risk*.

Definition 1 (Generalization error)**.** *Given a hypothesis $h : X \to Y$, a loss function $L : Y \times Y \to \mathbb{R}^+$, and a distribution $\mathcal{D}$ on $X \times Y$, generalization error is defined as*

$$R(h) = E_{(x,y)\sim D}[L(h(x), y)] \tag{3.1}$$

Definition 2 (Empirical error)**.** *Given a hypothesis $h : X \to Y$, a loss function $L : Y \times Y \to \mathbb{R}^+$ and a sequence $((x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)) \in (X \times Y)^n$,*

[2] Note that the input is a sequence rather than a set. While allowing duplicities may seem redundant, they can provide information about the training dataset itself and its underlying distribution. More frequent features are bound to occur more often if the samples are independent and identically distributed.

empirical error is defined as

$$\hat{R}(h) = \frac{1}{n}\sum_{i=1}^{n} L(h(x_i), y_i) \tag{3.2}$$

The previous definitions use the term *loss function*, and define it as $Y \times Y \to \mathbb{R}^+$ where Y is the hypothesis range. While this definition is sufficient in theory, defining a loss function is done with certain goals in mind. The actual loss function from the set $Y \times Y \to \mathbb{R}^+$ should fulfill those goals. Generally an error is being minimized so a loss function is also being minimized. A loss for an output of a hypothesis $y_h \in Y$ should decrease as the output gets in some sense closer to the true label $y \in Y$. an extreme case is a situation where $y_h = y$. Typically $L(y, y) = 0$ for every $y \in Y$.

A simple example of a loss function is $L_1 : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$ for which $L_1(x, y) = |x-y|$. This function fulfills all conditions for a metric and the assumptions about a loss function mentioned previously. It can also be generalized for $\mathbb{R}^d$ ($d \in \mathbb{N}$) as $L_1(x, y) = \sum_{i=1}^{d} |x_i - y_i|$. This generalized version is in different contexts also called L_1-*distance* or *Manhattan distance*.

Another example is the *0-1 loss* for which $L(x, y) = 0 \iff x = y$ and $L(x, y) = 1 \iff x \neq y$. This function is typically used for discrete Y.

Now assume a learning dataset d is provided. Sampled features are elements of set $\mathbb{R}^2$ while labels are boolean, i.e. from $\{0, 1\}$. The set is visualized in figure 3.2. Considering the 0-1 loss, can it be assumed that a hypothesis with empirical error equal to 0 is in some sense good? Or can it at least be assumed that it is better than a hypothesis with empirical error equal to some given $l > 0$?

As an example a *memorizing learner* M is introduced. Learned features are elements from $\mathbb{R}^2$, each labeled with 0 or 1. A memorizing learner returns a hypothesis h. h returns a label 1 for feature x, if $(x, 1)$ was a part of the training dataset, and 0 otherwise. As there doesn't exist $x \in \mathbb{R}^2$ such that $(x, 0) \in d \wedge (x, 1) \in d$, learner does indeed return a hypothesis with 0 empirical error. But simply looking at the dataset makes this hypothesis suspect. The reader probably has a hypothesis of his own that seems more plausible. Yet this learner always returns a hypothesis with zero loss for any dataset, if there are no two samples with identical features and different labels. Fortunately this does not imply that using empirical error to evaluate hypotheses is a dead end. This approach just needs to be modified to be useful.[3]

One would need to know the underlying distribution D to precisely compute the generalization error of a hypothesis. While D isn't known in most cases, the generalization error can be at least approximated. Assuming that there is a sequence $S' = ((x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)) \subseteq (X \times Y)^n$ where individual samples are independent and identically distributed, $E_{S' \sim D^n}[\hat{R}_{S'}(h) = R(h)]$.

Theorem 1. *If all elements of* $S = ((x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)) \subseteq (X \times Y)^n$ *are sampled independently from distribution* D*, then* $E_{S \sim D^n}[\hat{R}_S(h)] = R(h)$.

[3]The previous example is inspired by, and to a large extent identical as the one in subsection 2.2.1 in Shalev-Shwartz Shai [2014].

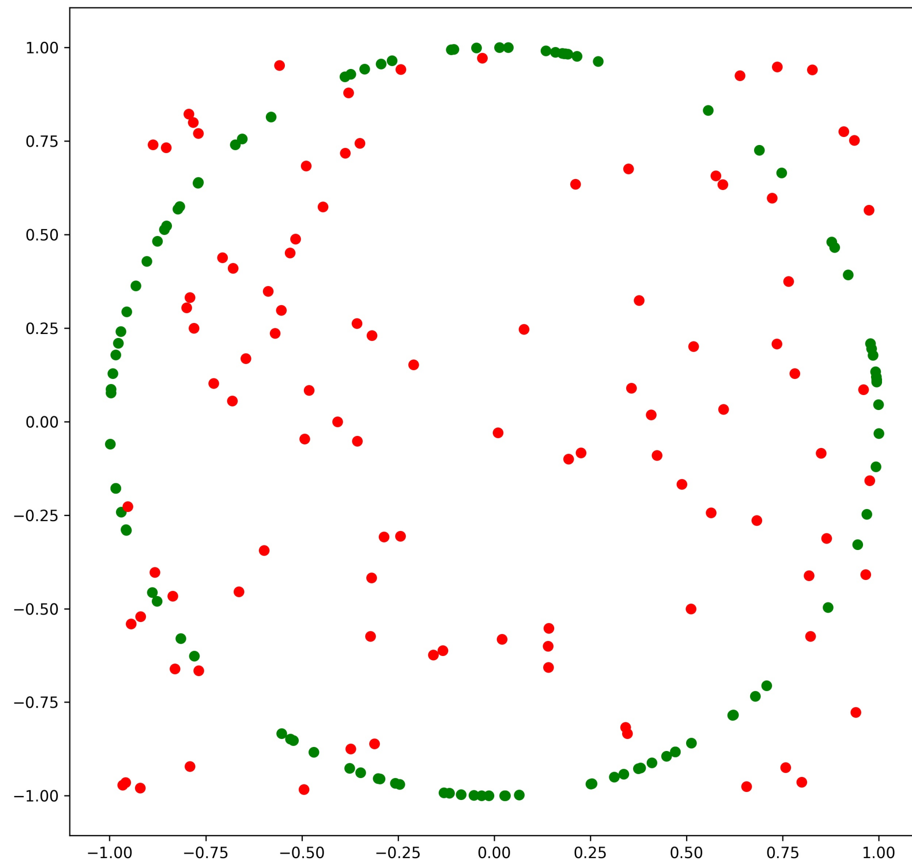

Figure 3.2: A set of features from dataset d in R^2. Points with label 1 are marked with green colour while points with label 0 are marked with red colour.

Proof.

$$\begin{aligned} E_{S\sim D^n}[\hat{R}_S(h)] &= E_{S\sim D^n}[\frac{1}{n}\sum_{i=1}^{n} L(h(x_i), y_i)] = \\ &= \frac{1}{n}\sum_{i=1}^{n} E_{(x_i,y_i)\sim D}[L(h(x_i), y_i)] = \\ &= \frac{1}{n}\sum_{i=1}^{n} R(h) = \\ &= R(h) \end{aligned} \tag{3.3}$$

□

This sequence S', called *validation set* [Shalev-Shwartz Shai [2014]], is not provided separately from the previously introduced training set S. Instead all available samples need to be split into those two sequences. This division introduces a dilemma. Providing more samples for training increases the probability of the learner returning a better hypothesis. At the same time, a small validation set is less likely to approximate the generalization error well. The question of how to set the relative sizes is in more detail investigated in subsection 3.3.3 and in Kohavi [1995].

Another situation arises when an algorithm or even a human tries to train several learners on the same dataset. It trains and then evaluates each with a training set and a validation set respectively. Yet as it discriminates among their results, it also becomes a part of the training process. This situation may not seem as important when talking about hypothetical learners, but becomes crucial when not only different types of models are tested, but also when a single model is trained several times with different configurations, also called *hyperparameters*. Each of these hyperparameters influences the choice of the final hypothesis. Such a process introduces bias that then needs to be accounted for and so additional sequence is needed. This one is called *test set* and it is independent of both training and validation sets. It is used only after model and its hyperparameters were chosen, and the model was trained so as not to prejudice the final result.

3.1.2 Overfitting and Bias

Consider the memorizing learner again. Its hypothesis space is too large, and so it can always return a hypothesis fitting the training data perfectly without expressing any underlying pattern. Such a case is called *overfitting* [Shalev-Shwartz Shai [2014]]. It usually happens when the hypothesis set is too large in comparison to the training set. A clear solution to this phenomenon is collecting more samples to train on. But the dataset is often fixed and cannot be expanded. Also for cases like the memorizing learner, no finite dataset can fit the correct hypothesis perfectly.

The other option is to decrease the hypothesis size. Rather than using the memorizing learner again to showcase this approach, it is clearer when using polynomial learners with differing maximal polynomial degrees. So let a learner L_p^k returns a hypothesis $p_k : \mathbb{R} \to \mathbb{R}$ from the set $P_k = \{\{(x, \sum_{i=1}^{k} \alpha_i x^i) \mid x \in \mathbb{R}\} \mid \forall i \in \{0, 1, \ldots k\} : \alpha_i \in \mathbb{R}\}$. As α_i can be 0, $P_0 \subseteq P_1$,$P_1 \subseteq P_2$ and so

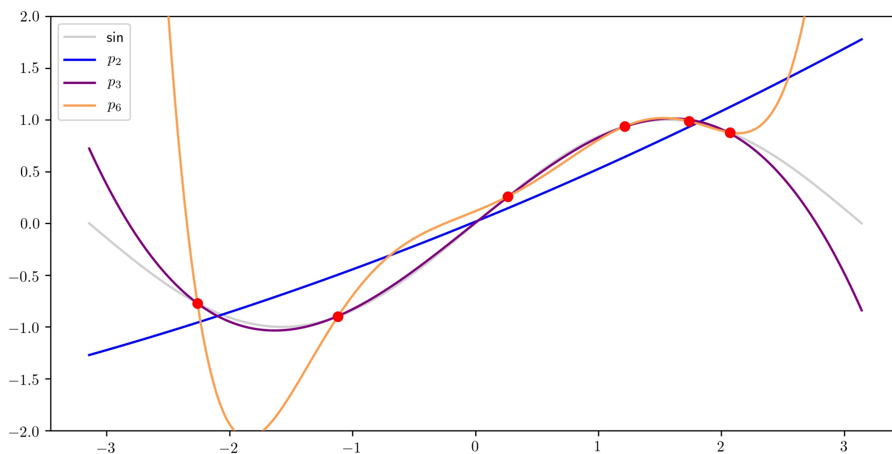

Figure 3.3: An example of overfitting showing an approximation of a sine curve with a series of polynomials. p_2 is a polynomial with a maximal degree 2, p_3 is a polynomial with a maximal degree 3, and p_6 is a polynomial with a maximal degree 6.

forth. Yet that does not always imply a better result for the more sophisticated method. Consider the approximation of a sine wave by several polynomials of various degree in figure 3.3. While only p_6 fits all points perfectly, p_3 achieves a lower generalization error. This effect is even more pronounced if the desired function is noisy. Then too complex hypothesis will fit this noise while a simpler approach is more likely to only follow the underlying pattern.

At the same time, decreasing the size of the hypothesis set tends to lower the probability that the desired function is actually learnable. Additionally, it puts more responsibility onto the designer of the learning algorithm and its user respectively. Simpler learning algorithms demand more information about the possible desired hypothesis. These limitations bias the learner by constricting the range of possible solutions. For that reason these restrictions are called *inductive bias* [Shalev-Shwartz Shai [2014]]. The need for compromise between bias and overfitting is called *bias-complexity tradeoff*, or sometimes *bias-variance tradeoff*.

The generalization error $R(h)$ of a hypothesis $h \in \mathcal{H}$ can be decomposed. This is firstly done to better illuminate the relationship between bias and overfitting. Secondly different types of error are caused by different phenomena. Each phenomenon has a different set of approaches to minimize it, and so it is useful to differentiate between them.

Assume $R_{\mathcal{H}} = \inf_{h' \in \mathcal{H}} R(h')$. Then $R(h)$ can be decomposed into two[4] parts as $R(h) = (R(h) - R_{\mathcal{H}}) + R_{\mathcal{H}}$. The first component, $(R(h) - R_{\mathcal{H}})$, is called *approximation error*. Informally it denotes how close is the chosen hypothesis h to theoretical optimum. The remaining component $R_{\mathcal{H}}$ is called *estimation error*. It is not concerned with h but rather with a best error a hypothesis in $\mathcal{H}$ can achieve.

[4] Sometimes third component is added. *Bayes error* denotes the error of an ideal hypothesis that does not need to be 0. More details in Mohri et al. [2012].

Figure 3.4: Two images of human faces created by a GAN-based generator, Source: Karras et al. [2018]

An approximation error is concerned with the choice of a hypothesis by a learner. It can be decreased by introducing new data or by modifying the hypothesis set. This often means decreasing its size. An example is the previous case with polynomials where P_3 achieved better results than P_6. Estimation error on the other hand never changes when new evidence is presented, and it can only grow if $\mathcal{H}$ is reduced to its subset. On the contrary, learner must become more complex, its hypothesis set larger, to decrease $R_{\mathcal{H}}$. This is the core problem of the bias-complexity tradeoff.

3.2 Neural Network and Deep Learning

Neural networks are a type of machine learning models originally inspired by biological neurons. They were in some form researched and developed since 1940s but they have gained greater popularity only in the 21st century after more powerful hardware allowed to train larger networks.

There are many use-cases for neural networks as they can be used for classification or regression for general problems. But there are areas in which they have achieved unmatched performance. For example image or voice recognition, or certain reinforcement learning approaches that rely on neural networks. One of the examples is artificial intelligence for the board game Go [Silver et al. [2016]]. Another case of successful utilization is presented by generative adversarial neural networks [Karras et al. [2018]] that can be for utilized to create images similar to those from the training dataset.

A biological neuron [Nicholls et al. [1999]] that is found in organisms, notably in human brains, is a signal processing unit. Its goal is to transfer and control signal travelling between various parts of body. These parts either stimulate neurons, for example cells in an eye, or they become stimulated by them, like muscle tissue cells.

A neuron consists of three distinct parts, *a set of dendrites*, a *cell body*, and

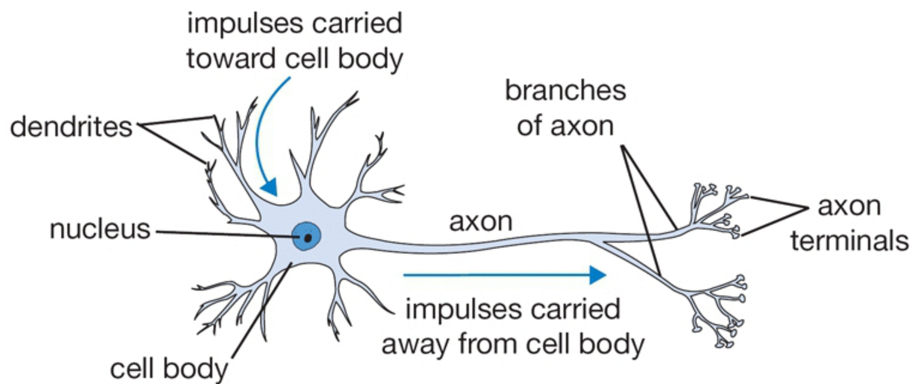

Figure 3.5: Biological neuron, Source: Karpathy [2015]

an *axon*. Dendrites form a connection to preceding cells that may send a signal. Then they are connected to a body of the neuron that may transfer an impulse of its own through an axon which splits into a series of axon terminals that can transfer signal further.

A strength of a signal received by a dendrite is influenced by a distance between the dendrite and the source of the signal. Additionally nearby transmitted impulses may have an inhibitory role as well. As a signal travels through the dendrite toward the cell body, it becomes stretched out and attenuated. As signals arrive into the body, they add up. If this signal exceeds a so called *activation potential*, an impulse is sent by the body through the axon forward. Otherwise no action is performed. It is useful to explicitly state that individual dendrites are not synchronized, and thus timing is important for eliciting a response from a cell body. When a response signal is sent forward, axon transfers it all the way to its terminals. These are typically connected to some following cell, e.g. another neuron. But as it was previously noted, a transfer of a signal may have an inhibitory role on the neighbourhood of an axon terminal. Thus sometimes these terminals are not connected to anything but they only perform this inhibitory role.

So while both timing of a signal transfer and positioning of dendrites are crucial, there are other phenomena that make things even more complex. Specifically sometimes when activation threshold is exceeded and a signal is sent forward over an axon, this impulse is also transfered "backwards" over the dendrites. This event is called *backpropagation* [Stuart et al. [1997]], but it does not have anything to do with learning process on artificial neural networks that is discussed later on.

Artificial neuron is a simplification of its organic counterpart. Dendrites are replaced each with a single value that represents the strength of a connection which may be also negative. Each such value corresponds to a single preceding input. Together these values form a vector $w \in \mathbb{R}^n$. Vector of inputs is aggregated to $x \in \mathbb{R}^n$. These signals are then multiplied by their weights and summed up together. Activation potential is here represented by $b \in \mathbb{R}$ which is called *bias*. Importantly bias can have a negative value, which partially breaks the activation potential metaphore. Nonetheless the value of bias is subtracted from the weighted sum of input signals. This result is then fed to a *transfer function* $f : \mathbb{R} \to \mathbb{R}$ that yields the final result. An example of f is a sigmoid function

$\sigma(x) = \frac{1}{1+e^{-x}}$ that reaches a value close to zero for negative values of x and a value close to 1 for positive values. The more extreme the value x, the closer is $\sigma(x)$ to either zero or one. This roughly corresponds to the impulse being sent, or not being sent across an axon. The following equation briefly summarizes this description[5].

$$h(x) = f(w^T x - b)$$

In comparison to a biological neuron, a signal is here represented as a single number rather than an impulse that changes during time. There is also zero probability of signal travelling backwards. This dramatically decreases the computational complexity, and makes artificial neurons easier to describe.

3.2.1 Multi-layered neural network

While a single neuron does have some uses in machine learning on its own, its main utility lies in being the basic building block of larger structures called *neural networks*[6]. Similarly to a neuron, a neural network has an input vector $x \in \mathbb{R}^n$. To simplify things a bit these values are also considered special cases of neurons, each with no inputs and thus a constant output. Every other neuron takes as input a set of values. Each value is either an element from the input vector x or an output of a different neuron. One can define a vertex for each neuron to allow the use of terminology from graph theory. Then the relation "the output of a neuron a is the input of neuron b" forms a directed edge on these vertices. In the current context, a graph of a neural networks needs to be acyclical. This allows for a non-ambiguous evaluation. At first only values of neurons utilizing elements of x are computed, then follows the computation for neurons utilizing x and outputs of neurons from the previous step, and so on. As we assume a finite number of neurons, this procedure will result in a correctly defined output of every network unit.

Networks are typically organized into *layers*, collections of neurons with the same "depth" in the previously mentioned acyclical graph. The input vector x forms the *input layer*. Neurons which do not have any successors, i.e. no other neuron depends on their output value, form the *output layer*. For all other units, situation is a bit more complicated, but typically neurons from k-th layer take as an input only output values from the (k-1)-th layer. In the same way their output is only directly utilized by the following, (k+1)-th layer. Layers which are neither input nor output are called *hidden*.

Similarly to the definition of an artificial neuron, neural networks have other alternative forms that do not fit the previous description. In addition to fairly minor deviations which respond to special types of input data, desired results or specific modelled hypothesis sets, there is far more to be said about neural networks that try to be more true to their biological predecessors. As such a general definition for all of them would prove to be very vague and thus almost useless. Neural networks fitting the previously mentioned and arguably restricted characterization are called *feed-forward neural networks*. This is due to their

[5]This is a description of a neuron typically used in deep learning. There are other, possibly more complex models that often resemble organic neuron in more detail.

[6]For this reason, neurons are also sometimes called units.

acyclicity as the output values are sequentially fed forward. In contrast *recurrent neural networks* were designed to work with sequences of data, and thus permit backward connections, although this reversed signal is used only in the following step of a sequence.

Now it is useful to talk more about activation functions in more detail as their existence plays a major role in the usefulness of neural networks. In addition to the previously noted sigmoid, there are other activation functions as well, notably *step function* [Kyurkchiev and Markov [2016]] or *hyperbolic tangent*. Step function equals 0 for input values smaller than 0 while for positive values it is always 1. For 0 step function equals $\frac{1}{2}$.

A hyperbolic tangent, typically shortened to *tanh*, is a scaled and shifted version of the sigmoid that instead of ranging from 0 to 1 has an infimum of -1 and a supremum of 1 with $tanh(0) = 0$. Although the definition of a hyperbolic tangent does not originate in machine learning, the direct relation to sigmoid is simply $tanh(x) = 2\sigma(2x) - 1$.

The currently most important, practical alternative to the sigmoid seems to be $ReLU$ which is defined as $ReLU(x) = max(0, x)$. Its crucial advantage is that its derivative is always 1 for $x > 0$, and it does not drop to values close to 0 like sigmoid does. The reasoning behind this assertion is explained in the following subsection about backpropagation as it is tightly linked to this concept.

Leaky $ReLU$ is a modified variant of $ReLU$ for which $LeakyReLU_\alpha(x) = x$ when $x > 0$, and $LeakyReLU_\alpha(x) = \alpha x$ otherwise. It is more computationally demanding, but it does not have the same problems as standard ReLU with "dead" neurons later presented in subsection 3.2.2.

There are are also activation functions that do not work on individual neurons but on whole layers instead. An example is the *softmax function* $s : \mathbb{R}^k \to \mathbb{R}^k$ which is for vector $v = (v_1, v_2, \ldots v_k)$ defined as

$$s(v)_i = \frac{e^{v_i}}{\sum\limits_{j=1}^{k} e^{v_j}} \tag{3.4}$$

Notably elements of the vector $s(v)$ sum up to 1.

The need for activation function becomes clear when one considers the possible functions represented by neural networks without them. Each neuron then represents a function $h(x) = f(w^T x - b) \underset{f\,=\,id}{=} w^T x - b$. We can extend the input $x = (x_1, x_2, \ldots x_n)$ with -1 to get $x' = (x_1, x_2, \ldots x_n, -1)$. Similarly for w', $w' = (w_1, w_2, \ldots w_n, b)$. Then it holds that $h(x) = w^T x - b = {w'}^T x'$. As the input vector x is identical for all neurons in a single layer, the output of this layer becomes $H_k(x) = W'^T x'$ with $w'_1, w'_2, \ldots w'_k$ being the extended weight vectors for each of the k neurons, which form the columns of W' [7] . A neural network with l layers and corresponding weight matrices $W'_1, W'_2, \ldots W'_l$ then for input x' yields $W'_1 \times W'_2 \ldots \times W'_l x'$. As the product $W = W'_1 \times W'_2 \ldots \times W'_l$ does not in any way depend on the value of x', this whole network can be then replaced by a single multiplication of x' by a matrix W. So for extended x' is every neural network equivalent to a linear map.

[7] To be rigorous this matrix needs to be extended in such a way that the last element of the output vector is again -1. As it is guaranteed that the last element of the input vector is -1, it suffices to append 0 to each column and add a new leftmost column $(0, \ldots 0, 1)$.

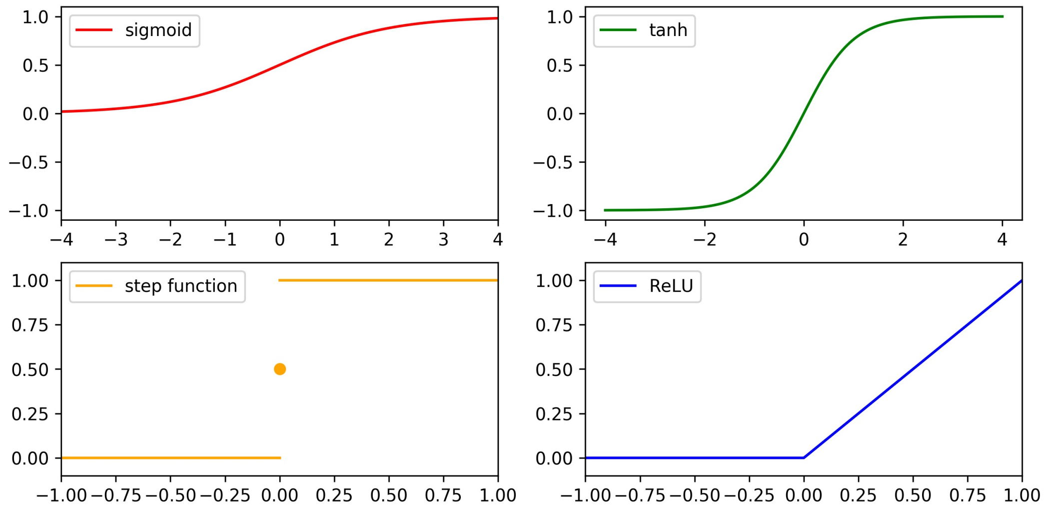

Figure 3.6: Activation functions; note the varying axis ranges.

The natural follow-up question to this result is "How much does the introduction of activation functions improve this situation?" Actualy quite a bit. The following theorem was in a slightly different form proved by Kolmogorov.

Theorem 2 (Universal Approximation Theorem [Cybenko [1989], Wikipedia contributors])**.** *Let $\phi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the* m*-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then given any $\epsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^>$ for $i = 1, \ldots, N$ such that it can be defined*

$$F(x) = \sum_{i=1}^{N} v_i \phi({w_i}^T x + b_i) \tag{3.5}$$

as an approximate realization of f, so $|F(x) - f(x)| < \epsilon$ for all $x \in I_m$.

So for a function f, there is a finite number of neurons in a single hidden layer such that they approximate f with correctly set parameters. This is a quite strong statement not only because due to this theorem, there is no need for more complex network structures. It can be additionally proved that the domain of f does not need to be $[0, 1]^m$, but instead $[a, b]^m$ for any $a, b \in \mathbb{R}$ such that $a < b$.

While this theorem exposes powerful theoretical properties of neural networks with only a single hidden layer, it shows no procedure to acquire the necessary weight parameters. So contrary to the implications of this theorem, practice seems to favour networks with large number of smaller layers rather than shallow networks where each layer contains a relatively large number of neurons. The general approach relying on putting a larger number of simpler layers in a sequence is called *deep learning*, and it is responsible for a majority of the successes mentioned in the beginning of this section. These advances were to a large extent enabled by a more powerful hardware capable of fast, parallel floating-point operations. That means both traditional GPUs as well as TPUs (Tensor Processing Units) designed explicitly for neural networks [Jouppi et al. [2017]].

3.2.2 Backpropagation

So far it was implicitly assumed that network weights are already set. To return back to hypothesis sets, a neural network structure and the corresponding weights completely specify a function $h : \mathbb{R}^k \to \mathbb{R}^l$ where k is the number of input neurons and l is the number of output neurons. h is a hypothesis for which the choice of activation function can heavily restrict the range of h. While it is possible to set weights manually for smaller tasks, it gets prohibitively more complex for non-trivial problems.

So it is necessary to find a mechanism that can choose a set of weights for a rigid network structure. These weights must then correspond to an optimal hypothesis or a hypothesis that is "close enough". A naive approach to this optimization problem might be to try all possible combinations of weights. While each weight can have every value from $\mathbb{R}$, this continuous set can be discretized to a finite number of values. If every weight can have only a finite number of values then the hypothesis set is finite as well. But even if the decreased precision does not lead to an insufficient solution, extensive search is not tractable as the size of the hypothesis set increases exponentially with the number of weights. Thus this approach will prove to be too computationally intesive for almost all non-trivial cases.

The general approach that is actually utilized here is not unique to neural networks. It is often used in situations when there is a well-defined error function, the state space is too large, and there is no known analytical solution. It's called *gradient descent* and it is an approach based on small incremental changes utilizing local information. In this case, a state is defined by weight values for all neurons.

An illuminating metaphor for such an approach is the situation of a hiker lost in mountains which is trying to find his way down. There is a heavy fog, and he sees only his close surroundings. As he doesn't know his position and thus cannot rely on a map, he is forced to depend only on his vision which is very limited. His approach is to look around and to find the lowest place he can see. By moving to this new position, he moves downwards. As he moves on, he discovers the area in front of him, and so he can again decide on his next move in this new position. In this way he proceeds to continue until he arrives to the valley.

The position of the hiker corresponds with the chosen hypothesis, and the height to the value of an error function. While the hiker tries to go downward, a learning algorithm tries to minimize the error on all samples it knows. While for the hiker the visibility is restricted by the fog, a learner is impeded by the complexity of the hypothesis space. To choose a next, better hypothesis, a learner searches the space around the currently chosen state. There may be a finite or an infinite number of candidates. This depends on the particular hypothesis space and the learner. When it selects the best option, it "moves" there, effectively selecting a new hypothesis. Then it again "looks around", evaluating the new neighbourhood. This process is repeated until no further improvement can be made, or until other terminating condition is met.

The arguably most important thing to note about this approach is that it can fail. The hiker can get unlucky and end up in an enclosed ravine high up in the mountains. Similarly a learning algorithm can reach a so-called *local minimum*, a hypothesis surrounded only by inferior or equal alternatives, but not the lowest

possibility. There are many modifications trying to alleviate this issue, the previously mentioned *tabu search* being one of them. Often they rely on procedures that in certain cases allow choosing a worse hypothesis with the hope that a better solution, one that does not lie in the currently found local minimum, can be achieved later on.

Backpropagation is a learning method for neural networks that is based on this gradient descent principle. It does not modify the structure of the neural networks, but only changes the weights. As each sequence of weight values corresponds to a particular hypothesis, this approach leads to selecting the hypothesis near the local minimum. In the most general view, backpropagation fits very closely the gradient descent pattern. An original hypothesis, in this case defined by randomly selected weights, is generated. Then a best candidate is chosen from a neighbourhood of this original hypothesis. As a hypothesis is defined by a network's set of weights, each hypothesis is defined by a point $h \in \mathbb{R}^n$ where n is the total number of variable network weights. Instead of testing a sequence of candidates to see which one is best, backpropagation instead chooses the direction in which to proceed. This direction is defined by the gradient of the error function in $\mathbb{R}^n$. How is this gradient computed is explained later on. Importantly backpropagation chooses to make a step in the opposite direction of the gradient vector thus proceeding in the "steepest" direction that minimizes error. As this happens, a new hypothesis defined by a vector in $\mathbb{R}^n$ is chosen and the whole cycle starts again until a terminating condition is met.

In a more detail, there is a weight vector $(w_1, w_2, \dots w_n) = w \in \mathbb{R}^n$ and a gradient ∇e of the error function in point w. To get the new hypothesis vector w', it is necessary to make a step in the opposite direction of the ∇e vector. The exact position of w' is specified by a constant $\alpha > 0$. For individual weights, this approach is defined by the equation below.

$$w'_i = w_i - \alpha(\nabla e)_i \tag{3.6}$$

While ∇e specifies the direction in which to move, it does not precisely define the new vector w'. For that vector, one needs to determine the value of α, a so-called *learning rate*. In the simplest case of backpropagation, α is set before training is even started, and then kept constant during the whole process. More sophisticated methods vary the learning rate during training, typically slowly decreasing it with time. Either way, learning rate can have a very large effect on the final state of the network and the training time. A learning rate that is too low can result in an extreme number of iterations needed to reach the local minimum. On the other hand, a large α can produce such a change that the metaphorical valley is jumped across, and thus missed. There does not seem to be a method to compute a balanced learning rate before a training process occurs, and as such it is an important hyperparameter for which multiple values should be tested.

So far a general gradient descent approach was defined, and gradient-based method was introduced to get a successor of a current hypothesis, but the gradient on all weights was just assumed to be provided by some abstract black-box mechanism. Actually the term backpropagation describes quite closely the principle in which gradient is computed even though it is not apparent straight away.

There are two basic pieces of information needed to understand backpropa-

gation. Originally a so-called *chain rule* must be defined. It allows to express a single derivative $\frac{d(f \circ g)}{dx}$ of a composite function $(f \circ g)$ with two derivatives $\frac{d\ f(g(x))}{d\ g(x)}$ and $\frac{d\ g(x)}{d\ x}$.

$$\frac{d\ (f \circ g)(x)}{d\ x} = \frac{d\ f(g(x))}{d\ x} = \frac{d\ f(g(x))}{d\ g(x)} \cdot \frac{d\ g(x)}{d\ x} \tag{3.7}$$

The second piece of information is more of an enforced invariant. It is necessary for all computational operations done on a neural network to be derivable. As all operations on the previously defined networks can be broken down into addition, multiplication, and activation functions, the only issue seems to be the choice of activation functions. These are not always derivable, which rules out for example the step function as it does not have a gradient at 0, and thus cannot be used for backpropagation.

Using the chain rule one can then compute gradient for every weight of every neuron. This is done by first computing the gradient of the error. This gradient is then propagated backwards using the chain rule and known derivatives of the various components like multiplication or various activation functions. This is done until all weights have its gradient computed.

As an example a backpropagation will be applied and presented on a single neuron. Input will be the extended input vector with -1 denoted as x, and a weight vector w will again also contain bias b as it was done in the previous subsection in an example about networks without activation functions. The output of a neuron is $y = f(w^T x) = f(\sum_{i=1}^{n} w_i x_i)$ where f is an activation function derivable in $w^T x$. The gradient "flowing" backwards is $\frac{d\ l}{d\ y}$ where l is the loss for the current sample. It does not matter if this neuron is in the last layer or if there are several layers after it.

What is needed to compute are the values $\frac{d\ l}{d\ w_i}$ and $\frac{d\ l}{d\ x_i}$ for $i \in [n]$. The first term is necessary for weight change, and the second term is necessary for propagation of gradient to lower layers if there are any. The procedure will be shown only for w_i as x_i is computed almost identically. Using the chain rule, the desired derivative $\frac{d\ l}{d\ w_i}$ equals $\frac{d\ l}{d\ y} \cdot \frac{d\ y}{d\ w_i}$. The derivative $\frac{d\ l}{d\ y}$ was the value propagated from the following layer, and thus it is known. $\frac{d\ y}{d\ w_i}$ can be broken down further using the chain rule.

$$\frac{d\ y}{d\ w_i} = \frac{d\ f(\sum_{j=1}^{n} w_j x_j)}{d\ w_i} = \frac{d\ f(\sum_{j=1}^{n} w_j x_j)}{d\ \sum_{j=1}^{n} w_j x_j} \cdot \frac{d\ \sum_{j=1}^{n} w_j x_j}{d\ w_i} \tag{3.8}$$

The first part is just a derivative of f for the value $w^T x$. As it is demanded that f is derivable at $w^T x$, this value is known. As for the second part, $\frac{d\ \sum_{j=1}^{n} w_j x_j}{d\ w_i}$, it equals x_i. Thus the results for w_i and x_i are following:

$$\frac{d\ l}{d\ w_i} = \frac{d\ l}{d\ y} \cdot f'(w^T x) \cdot x_i \tag{3.9}$$

$$\frac{d\ l}{d\ x_i} = \frac{d\ l}{d\ y} \cdot f'(w^T x) \cdot w_i \tag{3.10}$$

The crucial part of these equations is that they contain only the passed $\frac{d\ l}{d\ y}$ and values of given neuron, namely w and x. Thus the result does not in any way depend on other neurons in the same layer[8]; all values are local. This means that derivatives for one layer can be computed in parallel.

There is one pitfall that for some time restricted the use of backpropagation to shallower networks. Consider the previous example, namely the equation for $\frac{d\ l}{d\ x_i}$ with sigmoid being the activation function. The derivative of a sigmoid is always relatively low. It is highest for 0, namely $\frac{1}{4}$, and it tends to 0 for values approaching either $-\infty$ or ∞. So for network with k layers, the error gradient is multiplied by $f'(w_k^T x_k), f'(w_{k-1}^T x_{k-1}), \ldots f'(w_1^T x_1)$ before reaching the weights of the bottom layer. Each of these values will be at most $\frac{1}{4}$ but can be far lower. Thus the weights for lower layers will change markedly slower than their counterparts closer to output. This fact limits the number of layers a network can have as the bottom layers may need an excessive number of iterations to finish training.

This issue is called *vanishing gradient problem*, and it is the reason why ReLU activation functions have proven to be so popular even though they are technically not derivable in 0. The derivation is always either 0 or 1, and thus the gradient does not suffer from the same issues as sigmoid or tanh.

But ReLU units do have an another problem. Namely during a weight change, weight can be pushed to such an extreme value that $w^T x$ will be always negative and $\text{ReLU}(w^T x)$ will thus be always equal 0. So the gradient flowing backwards will also be always zero and the value of this extreme weight will never change resulting in a "dead" neuron. This issue can never be solved entirely but it can be alleviated by using a low enough learning rate and relatively small initial weights.

So far the details of an error function computation were not given. Typically for a single sample, L_1 or L_2 error[9] is used on the difference between the actual and the desired output. For multiple samples the accurate option is to compute error for each sample and then average these together. But this option demands to compute an output for all of them which becomes extremely slow for larger networks and larger datasets containing more than tens of thousands of samples. Similarly backpropagation needs to happen for all of these samples separately as their neuron activations are different. To cut down on computation time, one can perform backpropagation and subsequent weight change for only a single, randomly chosen sample. This approach is called *stochastic gradient descent*. It drastically cuts down the needed computation, but the error of a single sample is in many cases a poor approximation of the real error function. A *mini-batch backpropagation* uses small batches of training samples. In this way a single sequence of forward computation, backward propagation, and weight change is still relatively fast but the aggregated error is a closer approximation of the overall error. As such this approach typically converges in lower number of iterations and also faster.

Backpropagation is not the only option for learning network weights for a particular task. *Genetic algorithms* are also capable of setting neuron weights.

[8]There are exception, for example softmax layer. Nonetheless the computation remains local.

[9]Both are defined later in equations 3.20 and 3.21. Both loss functions sum errors for individual samples which are computed separately.

In many cases they are not limited to the standard supervised learning scenario where we always need to know all desired values for each training sample. Instead they rely on *fitness* which is a form of a score. It describes how well a network is doing. A notable genetic algorithm is *NEAT* [Stanley and Miikkulainen [2002]] which does not only learn weights but also the structure. It starts with a simple network with no hidden neurons which grows more complex if needed.

Another approach is *Hebbian learning* [Hebb [2005]] which is based on the rule that neurons with correlated outputs should have its connecting weight strengthened. This is summed up in the sentence "If they fire together, they wire together."

These alternatives are capable of learning network weights but they have its limitations. Typically they are not capable of training large networks, and as such they are poor subsitutes for backpropagation, especially for deep archictectures.

3.2.3 Convolutional networks

One of the first successes of deep learning was object recognition. Its goal is to recognize not only easily defined items, e.g. a circle or a cube, but also real-world objects like different types of animals, cars, etc. This should be possible with photographs made under various conditions like sunny weather or indoor environment. Deep learning has achieved state-of-the-art results in this task using a novel approach built on biological principles.

While the first impulse might be to just stack a series of standard neuron layers, this approach is bound to fail. Assuming colour pictures of size 256×256 with RGB encoding, each image is represented by $256 \times 256 \times 3 \approx 196k$ integer values in range $[0, 255]$. It is almost pointless in this case to try to come up with a neural network structure without further experimentation, but it is helpful to illustrate how many weights a basic feed-forward network might have. A network with a single hidden layer of size 1000 and a single output neuron, whose value represents an appearance of some phenomena[10] in an image, has over 196 million parameters. This is assuming that each neuron in the hidden layer is connected with each neuron from the previous, input layer. Such a layer where every neuron takes all values from the previous layer as input is called *fully-connected layer*.

In addition to the fact that this network would require an extreme number of samples to train, this approach largely diverges from vision processing discovered in organisms. For example in cats, neurons processing signal from the retina are connected only to a small number of local "inputs" [Hubel and Wiesel [1968]]. Additionally neighbours tend to work similarly. They all send a signal across the axon if there was a horizontal edge detected in their respective field. This general principle of locality and uniformity is emulated by *convolutional layers* which form a basic block for building *convolutional neural networks* capable of accurate object recognition.

A convolutional layer consists of a grid of neurons where each neuron is connected only to a spatially local selection of inputs from the previous layer. An image with each pixel represented in 3 RGB channels forms in the input layer a three dimensional grid. Height of the grid is the height of the image. Similarly for

[10] That may be for example an occurence of a certain object like a dog or a car. An output layer may consist of several neurons. Each one corresponds to a different phenomenon.

width. Depth equals three; with values of each channel saved in a single so-called *depth slice*. While depth is set constant for an image due to a constant number of channels, this is not an invariant that is always true. Actually a convolutional layer creates a grid of neuron with an arbitrary depth. For each depth slice a convolutional layer has a set of weights called *filter*, sometimes also *kernel*, that define the values in this depth slice. All filters are independent of each other, but they do have the same shape.

A filter is the structure that introduces the previously mentioned locality and uniformity. One way to think about it, is that all neuron in given depth slice share the same weight vector. Arguably a more descriptive way of looking at it gets rid of the concept of a layer as a grid of neurons altogether and replaces it with a simple 3D memory array. This memory is then filled by the set of filters. In this case each filter is a smaller 3D set of weights with depth identical to the input layer. Such an object "slides" across the input layer and sequentially generates a set of values given by its weights and the connected input values. These connected input values are called a *receptive field*.

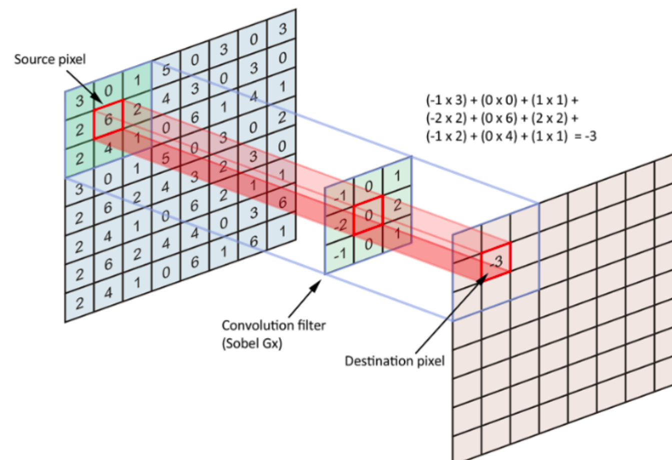

Figure 3.7: Convolutional filter (Source: Guo et al. [2015])

There are multiple hyperparameters for a convolutional layer. The most obvious choice that needs to be considered is the size filter. While a convolutional layer always takes values from all depth slices, a choice needs to be made for filter's width and height. Next is the depth of a convolutional layer or rather the number of filters. Depth typically grows monotonically from input layer toward output layer in convolutional layers. The choice of step a moving filter makes as it slides across its input is another important parameter. While it may seem natural for a filter to always only move one position in a particular direction, this is not neccessary. Additionally an input layer can be padded on its sides with constant values, typically 0. One of the consequences of such a modification is the change of possibly both height and width of output layer.

The convolutional layers are not the only layers used in convolutional networks. The representation of an image in an input layer is rather large compared to the output with only a handful of neurons. As such there must be a method for converting larger layers to ones with lower number of neurons. There are several possible approaches. A crude method is to use a fully connected layer with smaller number of layers than its predecessor. This is a valid solution for

upper parts of the network where the number of neurons is relatively small, but it is not reasonable for layers closer to the input layer. Another possibility is to use convolutional layers with step larger than 1. While this is feasible, *pooling layers* are specifically designed to solve this problem. They work very similarly to convolutional layers with a shared filter "moving" over the input layer and creating output only from a spatially limited neighbourhood. The crucial difference is that while the convolutional filter is composed of trainable weights, pooling layers filters are set by choosing the type of the pooling layer. They stay constant during backpropagation.

There are two main types of pooling layers. Filter of *average pooling layer* outputs the arithmetic average of all local input values. Conversely *max pooling layer* selects their maximum.

A first deep learning model for object recognition that successfully combined the previously defined layers was AlexNet [Krizhevsky et al. [2012b]]. It works by putting a max pooling layer after a convolutional layer or a sequence of convolutional layers several times. Lower convolutional layers extract features like edges or a single colour channel while those that are higher extract more abstract patterns. As it was noted previously, a pooling layer is used to sequentially decrease the size of processed data, and thus prepare it for the last sequence of fully connected layers. These are meant to work more flexibly with the features extracted by convolutional layers and to generate the final result.

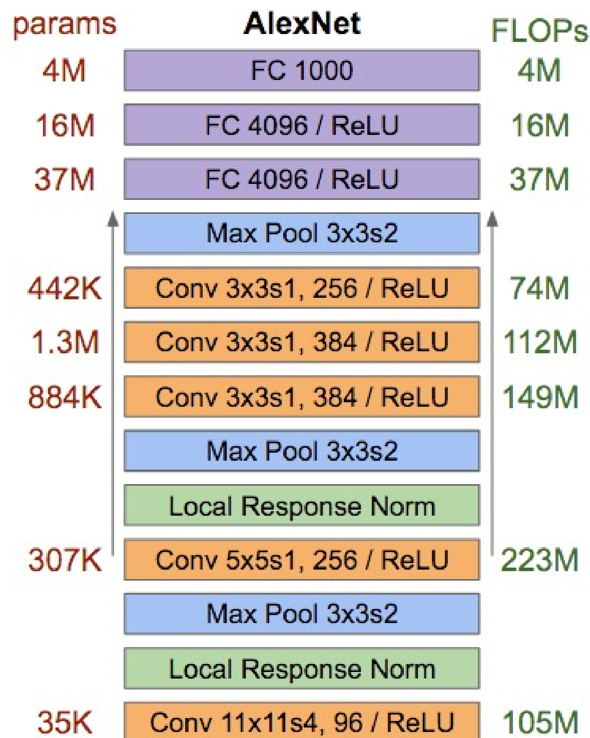

Figure 3.8: AlexNet layers. *params* denotes the number of trainable parameters for each layer. *FLOPs* denotes the number of floating-point operations needed for forward pass of each layer. (Source: Guo et al. [2015])

3.3 Model Evaluation[11]

Assuming that a model generated by some machine learning technique is already given, the training process is not yet finished. Each model has a set of attributes and characteristics that are important. In addition to the more straightforward aspects as model size or the time needed to acquire results for new data, the most crucial question that can be asked is "How good is the hypothesis?" This question is here purposefully asked in such a general fashion as there is a range of criteria that can influence the desired result. It is also relevant when a model is being trained as it is repeatedly decided, if training is to be continued.

For classification one of the most basic methods is taking the number of correct test results divided by the number of all test samples. This method is simply called *accuracy*. Although such a straightforward approach is often used, it is not always appropriate. Consider the case when a model is trained to detect a malignant tumour in a tissue. If a hypothesis mistakenly classifies harmless cells as malignant then the patient will be checked by a specialist and the subsequent costs will be only monetary. If the cases are reversed and a tumour is classified as benign, the situation will be graver. Here it might be useful to treat different cases of misclassification with different penalties.

Another case where accuracy might not be the correct measure is a situation in which the testing dataset is highly unbalanced. For example if samples are labeled either as positive or negative, and out of 1000 test samples only 10 are negative, the accuracy of a hypothesis that classifies every sample as positive is 99%. At first sight this seems like an excellent result yet every negative sample is misclassified.

An issue that also needs to be kept in mind is that there may not exist any hypotheses correctly classifying all given samples. That may be caused by noisy data or badly annotated training samples. Another option is that the dataset does not provide enough information to correctly discern between different results. Either way it is necessary to take into account these shortcomings and take them into account during model evaluation. Additionally instead of searching for a perfect result for each point in the sample space, a probabilistic approach needs to be taken. That means selecting the most likely correct answer or the one closest to the actual result.

In the following section several methods for model evaluation are presented. In addition to their definitions, their strengths and weaknesses are described as well. Cross-validation, an approach mainly used for estimating overfitting of a model, is also explained. Emphasis is put on more practical measures used during model training. For that reason more theoretical approaches and measures like *Rademacher complexity* or *Growth function* are not discussed.

3.3.1 Methods for Classification

Definition 3 (Accuracy)**.** *Given desired hypothesis $f : X \to Y$, modelled hypothesis $h : X \to Y$, and finite test set $X \subseteq X$, accuracy $a \in [0, 1]$ is defined as:*

[11] The structure of this section is loosely based on the notes of Martin Pilát for the lecture *Application of Computational Intelligence Methods* [Pilát [2018]]

	Real Positive	Real Negative
Predicted Positive	TP	FP
Predicted Negative	FN	TN

Figure 3.9: Confusion matrix

$$a = \frac{1}{|X|} \sum_{x \in X} I(f(x) = (h(x)) \tag{3.11}$$

where $I(a = b) = 1 \iff a = b$ *and* $I(a = b) = 0 \iff a \neq b$.

The first, already noted method for evaluating classification models is *accuracy*. It does not distinguish between different samples from the test set or between different types of mistakes. Despite its simplicity it is widely used as it is useful when the previously mentioned phenomena are not too significant. It is also helpful in situations where the sample set is not well described. Then it can be used as a starting position before a more detailed exploration.

Let's consider the situation when a given classifier is designed to assign either a positive or a negative label. This is the most basic, but also arguably the most common type of a classifier. For such a situation, there are 4 possible results in relation to original, or rather correct, labels. If a classifier assigned correctly then either both real and predicted labels are positive, such a case is called *true positive (TP)*, or both of them are false, then it is a *true negative (TN)*. The case when the true label is positive while negative label was predicted is called *false negative (FN)*. If the true label was negative but the predicted label was positive then such a case is denoted as *false positive (FP)*.

TP, TN, FP, and FN are often presented in a 2×2 matrix called a *confusion matrix* (3.3.1). It is used to succinctly present the distribution among the previously noted cases. This table can be also generalized for classifiers with $n \geq 3$ output labels. Then for a confusion matrix $M \in \mathbb{N}^{n \times n}$, the element M_{ij} denotes the number of samples with a real label j and a predicted label i.

Using the terms defined above, accuracy can be for binary classifiers redefined as $\frac{TP+TN}{TP+TN+FP+FN}$.

Definition 4 (Precision)**.**

$$P = \frac{TP}{TP + FP} \tag{3.12}$$

Definition 5 (Recall)**.**

$$R = \frac{TP}{TP + FN} \tag{3.13}$$

Let's return to the situation with an unbalanced dataset. There are 990 positive samples and 10 negative samples. The failure of accuracy in this scenario was already presented. The issue here is the ratio of positive and negative samples is not taken into account. As such even if TN is 0, TP is still very high for the always-positive classifier and accuracy is close to 1. Both precision and recall are useful as they are more sensitive to unbalanced datasets although this is not their only quality.

Precision denotes the portion of true positive samples among all those that are marked as positive. For that reason it is also sometimes called positive predictive

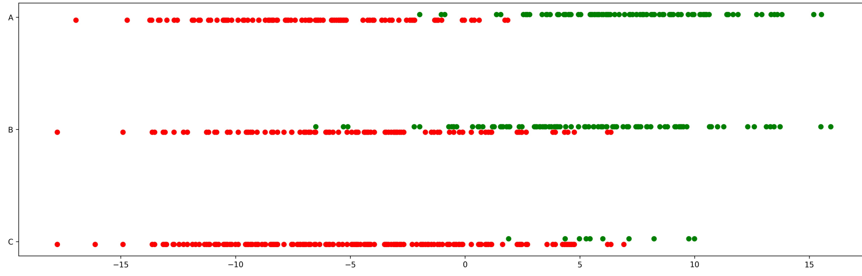

Figure 3.10: Datasets: A: balanced and small overlap, B: balanced and large overlap, C: unbalanced and large overlap; Positive samples are denoted by green colour while negative samples are red.

value. A useful example is the already mentioned tumour classification. While the always-negative classifier reaches very high accuracy as the percentage of positive samples in dataset will be most likely quite small, its precision will be exactly 0.

Recall on the other hand expresses the percentage of truly positive samples that are actually labeled as positive. Instead of recall, a value $1 - R$ that is called *miss rate* is sometimes used. It denotes the portion of true positive samples that were misclassfied, or said differently "lost".

Naturally a classifier with both high recall and high precision is desirable. It is possible to use the melanoma example again to showcase usage differences in both measures. While high precision implies that not many harmless lesions were classified as malignant, high recall means that at most a small percentage of dangerous samples was not selected. Here achieving recall close to 1 is very important, but in other cases precision might be crucial. It's important to note that neither one is generally better. Neither precision nor recall aims to substitute accuracy, they are measures meant to highlight distinct properties of hypothesis in relation to the data set.

Definition 6 (F-score)**.**

$$F_\alpha(R, P) = \frac{(\alpha + 1)RP}{R + \alpha P} \tag{3.14}$$

Both precision and recall stress some particular aspect of the given classification. *F-score* is a harmonic mean of precision and recall that aggregates these two values into a single real value. Harmonic mean is used instead of, for example, the average mean because it penalizes extreme values of both precision and recall more heavily. The α parameter is used to weigh the importance of precision in relation to recall and vice versa. Typically when neither of these two values is more important, α is set to 1.

Many classification algorithms do not directly output a binary value, but instead they assign some $x \in \mathbb{R}$ to every sample which then needs to be transformed into a binary value. For logistic regression this x represents probability that a label is positive while for neural networks the output may not be so easy to interpret.

In general a hypothesis h creates a sequence of real-valued numbers where higher values correspond to a positive result while lower values correspond to a negative result. At this situation it does not make much sense to label a sample as positive if there is another negatively-labeled sample with higher output value. So selecting a threshold $t \in \mathbb{R}$ unambiguously assigns a positive or a negative label to every sample. Input datum x is labeled as positive if $h(x) \geq t$ and as negative otherwise.

Naive way to evaluate such datasets is to first label the data and then use one of the previously presented methods. As only the real-valued hypothesis output can be used, some way to generate positive and negative labels must be devised, i.e. a threshold must be set. One way to do this is to try every possible value as a threshold and then take the best option. But it is not obvious what does "best" mean in this context. If the measure is recall, sometimes also called *true positive rate*, that is the used measure, the best solution is to label everything as positive. On the other hand using *false positive rate* defined as $\frac{FP}{FP+TN}$ yields classifier that labels everything as negative.

Additionally it is not only important for a hypothesis to yield results that can be separated by a threshold. Intuitively it is also useful for the positive cluster and the negative cluster to be far from each other so to that small perturbations to the threshold value do not change results too much or even at all.

Definition 7 (Receiver Operating Characteristic Curve)**.** *Given a true positive rate (recall) TPR and a false positive rate FPR,* Receiver Operating Characteristic Curve *is defined as:*

$$ROC(x) = TPR(FPR^{-1}(x)) \tag{3.15}$$

where $x \in [0, 1]$. And the area under ROC *is then defined as:*

$$AUR = \int_{x=0}^{1} ROC(x)dx \tag{3.16}$$

Instead of taking a single measure or even several measures at given point, AUR aggregates TPR and FPR for all possible thresholds. It does that by plotting the ROC curve that expresses the compromise between TRP and FPR and taking the area underneath. The ideal case is when a classifier assigns all negative samples to a single point $x \in \mathbb{R}$ and all positive samples to a single point $y \in \mathbb{R}$ where $x < y$. Then the ROC curve will go through the point $[0, 1]$ and AUC will equal 1.

To present behaviour of AUC and its shortcomings, 3 distinct datasets were created. Apart from a true label, a real value that may be interpreted as a direct output of some classifier is included. In all cases the mean of the positive cluster is higher than the mean of the negative cluster. Each time there is an overlap between the values of both positive and negative samples. The dataset A consists of the same number of positive and negative samples. Positive and negative samples cannot be separated with perfect accuracy, but setting the cut-off to about 0 will yield decent results. Set B is also balanced but this time the variance of the positive and the negative cluster is higher, and as such these clusters blend together. Set C is unbalanced as positive samples cover only 5% of the data. In addition, values of positive and negative samples are mixed quite thoroughly.

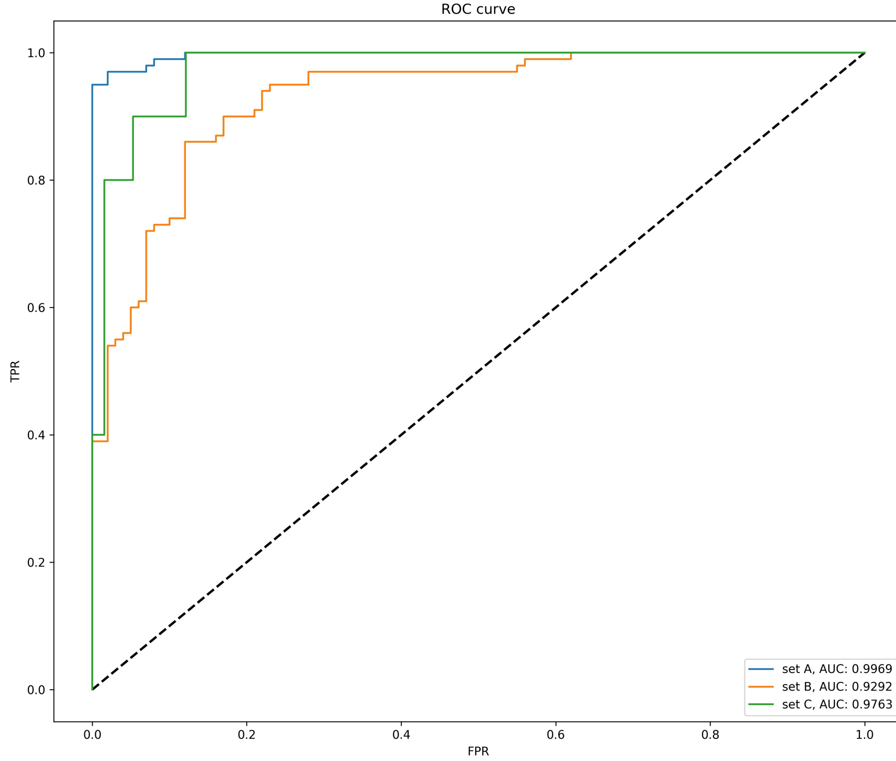

Figure 3.11: ROC curve plotted for all three datasets. The dashed line corresponds to a set of classifiers which assign both positive and negative label randomly with a probability $p \in [0, 1]$. Each classifier corresponds to a single point on the line.

From the corresponding figure it can be seen that results for sets A and B meet the previously stated requirements for an evaluation method. Meanwhile AUC for set C is almost as good as for set A. This is because set C is unbalanced and misclassified positive samples do not have such an impact.

Definition 8 (Precision/Recall Curve)**.** *For precision and recall,* Precision-Recall Curve *is defined as:*

$$PR(x) = recall(precision^{-1}(x)) = TPR(precision^{-1}(x)) \tag{3.17}$$

where $x \in [0, 1]$*. And the* area under PR *is then defined as:*

$$AUPR = \int_{x=0}^{1} PR(x)dx \tag{3.18}$$

The issues with ROC curve are especially problematic for medical data where the amount of negative data heavily outweighs the positive component. The Precision/Recall curve on the hand does not share these downsides as it can be seen from the results. While AUC seemed to be quite high for set C, the AuPR metric evaluates it more negatively. Actually it ends up even worse than set B.

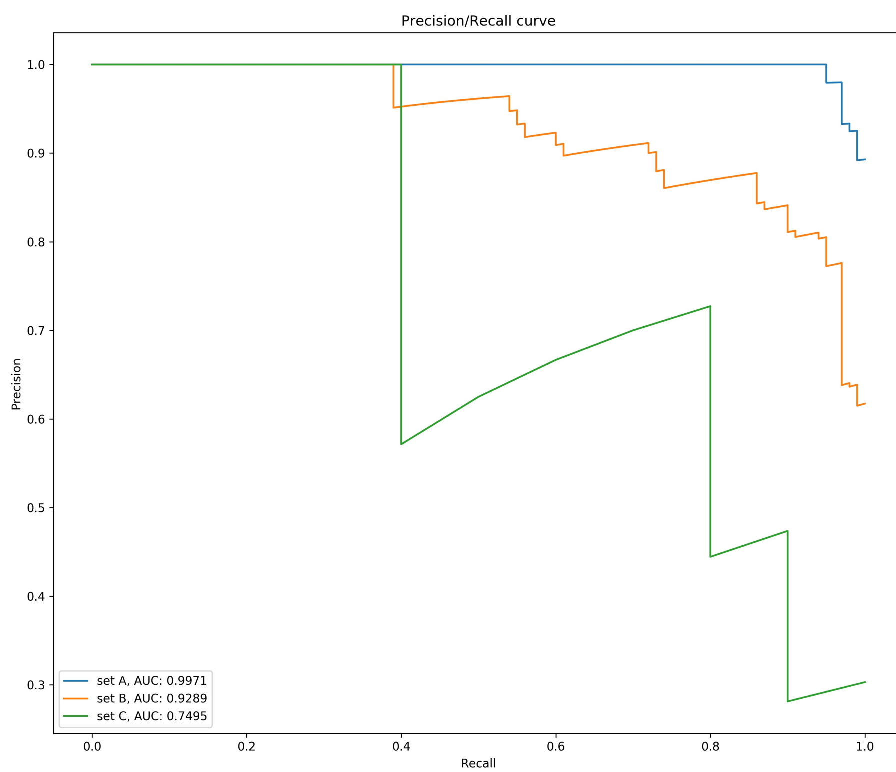

Figure 3.12: PR curve plotted for all three datasets.

The last mentioned method for evaluation of classification tasks is *Discounted Cumulative Gain* [Järvelin and Kekäläinen [2002]]. It is used in information retrieval for assessing the returned list of queries. Such a list is often limited as the user does not typically care about the ranking of all possibilities, but wants only top k results. An example is Google web search, where the user wants not only a limited number of links per page, but it is also expected that the most relevant ones will be at the beginning.

Definition 9 (Discounted Cumulative Gain). *For a vector of retrieved objects of length n with corresponding relevance $rel_1, \dots rel_p$,* Discounted Cumulative Gain *is defined as*

$$DCG_n = \sum_{i=1}^{n} \frac{rel_i}{log_2(i+1)} \tag{3.19}$$

Higher relevance implies a closer fit to the user query.

As it can be seen from the preceding equation, DCG_n is maximized when the most relevant results are at the beginning of the vector. Apart from web search, this metric is also relevant for recommendation systems as its goal is again to supply a user with a series of objects, e.g. films. Subsequently it is a valid metric for drug-target interaction prediction as the top k predictions are the further research candidates.

3.3.2 Methods for Regression

The previous methods were geared toward classification, i.e. hypotheses with discrete output. Meanwhile regression aims to predict a real-valued output based on the input data. There are several main differences between these methods. Primarily there is an implicit similarity between output points in regression tasks. Given points $x, y, z \in \mathbb{R}$ where $x < y < z$, result y is better compared to result z for desired output x. On the other hand, there may also be various similarity measures for classification, but they must be explicitly defined. This ties to the second point, the fact that rarely does a hypothesis generate the exact desired value. Mostly a machine learning approach returns only an approximation of a desired hypothesis. This renders certain classification methods such as accuracy useless.

Definition 10 (L_1 error). *Given a sample set $\{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)\} \subseteq X \times \mathbb{R}$ and a hypothesis $f : X \to \mathbb{R}$ L_1 error is described as:*

$$L_1 = \sum_{i=1}^{n} |y_i - f(x_i)| \tag{3.20}$$

L_1 error measures the absolute value between desired and returned output. In practice it means that several mistakes, even if very large, will not not impact the resulting error too much as long as the other samples do not differ too much from their desired values.

Definition 11 (L_2, (Root) Mean Squared Error). *Given a sample set*

$\{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)\} \subseteq X \times \mathbb{R}$ *and hypothesis* $f : X \to \mathbb{R}$*, mean squared error is defined as:*

$$L_2 = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{3.21}$$

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2 = \frac{1}{n} L_2 \tag{3.22}$$

Subsequently the root mean squared error is defined as:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2} \tag{3.23}$$

The MSE measure more heavily penalizes large differences. Compared to the L_1 error, it favours many small errors to few large ones. It is often used as a target to minimize during training of neural networks. In these cases it is mostly multiplied by $\frac{1}{2}$ to reduce the amount of computation needed during backpropagation. In this form it is also referred to as L_2 *error*. Both L_1 and L_2 errors are used for weight regularization of neural networks to prevent overfitting.

RMSE adds square root to the MSE to make results more easily interpretable. It is meant to normalize values that are disproportionate to the original inputs due to the quadratic function.

Definition 12 (Coefficient of Determination)**.** *Given a sample set* $T = ((x_1, y_1), (x_2, y_2) \dots (x_n, y_n)) \subseteq X \times \mathbb{R}$ *where sequence* $(y_1, y_2, \dots y_n)$ *is denoted as* y *and given hypothesis* $f : X \to \mathbb{R}$*, coefficient of determination, also called* R^2*, is defined as:*

$$R^2 = 1 - \frac{MSE}{var(y)} \tag{3.24}$$

where MSE is computed for the data set T *and hypothesis* f*.*

Another way to normalize results of MSE is to use the R-squared measure also called coefficient of determination. It uses a variance of data set outputs to normalize the MSE. R^2 represents goodness of fit where 1 corresponds to ideal fit with zero error.

3.3.3 Cross-Validation

There is another important technique often used during model evaluation. Its task isn't to provide a single value characterizing a specific facet of the hypothesis. Instead it is used to estimate overfitting. This need for cross-validation arises from the previously mentioned bias-overfitting trade-off. It is crucial especially for deep learning models that are complex and thus their hypothesis set size is considerable. The amount of data needed to perform training of such models is significant. While compiling training samples can be both costly and time-intesive, deep learning models are often trained with barely enough data. In such cases the probability of overfitting is high.

The purpose of *cross-validation* is to generate several hypotheses $h_1, h_2 \dots h_k$ using a single learner L and a dataset d. The learner is provided with a subset

of the original data $s_l \subseteq d$ to train each newly-created hypothesis h_l. These subsets are typically distinct, although this property does not need to hold for some stochastic variants of cross-validation. Each hypothesis h_l is then evaluated on its validation set $d \setminus s_l$. This sequence of evaluation results is then statiscally analyzed. Most notably variance of accuracy, AUC, or AUPR provides information about the "fragility" of the learner and the dataset respectively. High variance implies that modifying dataset can greatly influence the hypothesis.

The standard cross-validation algorithm takes n data-points for supervised learning as its input. It then proceeds to randomly divide this sequence into k non-overlapping groups $f_1, f_2, \dots f_k$ called *folds* of size $\lceil \frac{n}{k} \rceil$ or $\lfloor \frac{n}{k} \rfloor$. Using this decomposition, k distinct models are trained always taking 1 fold as a validation set and the rest as a training set. So the first model is trained on $f_2, f_3 \dots f_k$ and tested on f_1; the second model is trained on $f_1, f_3, \dots f_k$ and tested on f_2 etc. This variant is called *k-fold cross-validation*.

The extreme case of the previous approach is the so-called *one-out cross validation* where $k = n$. In this case $\forall i \in [n] : |s_i| = 1$ and so h_i is tested on exactly one sample.

The *Monte Carlo cross-validation* [Dubitzky et al. [2007]] is a stochastic approach where the folds are created by random sampling from the overall dataset. In this case the size of both training and validation set are not dependent on the number of folds, which can be seen as an advantage. On the other hand, there is no guarantee that any particular sample is tested. This may have large implications on results, for example with highly unbalanced datasets.

A similar situation may cause problems for standard k-fold cross-validation as well, if some rare labels are grouped into only a small number of folds. In this case, the result may cause a high variance among the resulting measurements. The answer to this issue is *stratified cross-validation*. It enforces an approximately uniform distribution of samples for each label across all folds. For classification it effectively performs $l \in \mathbb{N}$ distinct cross-validations partitionings where l is the number of distinct labels in the dataset. These distinct folds are then merged together.

To be more rigorous, assume a dataset $d = (s_1, s_2, \dots s_n)$ and $l \in \mathbb{N}$ again being the number of unique labels. Let's denote $\forall l' \in [l] : i_{l'} \subseteq [n]$ the set of indices that are labeled with label l' in d. The sets $i_1, i_2, \dots i_l$ form a partition of $[n]$. So they do not share any elements and $\bigcup_{l'=1}^{l} i_{l'} = [n]$. A standard k-fold cross-validation partitioning is performed for every i_j ($j \in [l]$) that results in k sets of indices $i_j^{(1)}, i_j^{(2)}, \dots i_j^k$. The final folds for the stratified k-fold cross-validation are $f_1, f_2, \dots f_k$ where $\forall k' \in [k] : f_{k'} = \{s_j | j \in \bigcup_{l'=1}^{l} i_{k'}^{l'}\}$.

Before running a k-fold cross-validation, the number of folds $k \in \mathbb{N}$ needs to be chosen. It is good to state the factors that might influence this decision. Increasing k enlarges the training set used for every hypothesis. This in turn at least in theory increases the probability of the learner yielding a better classifier. The negative effect is the most pronounced for $k = 2$ where only 50% of the complete dataset is used for training.

On the other hand as there is more data for training, the validation set grows smaller. This tends to increase bias on individual folds. Another factor is the computation time needed to train a model. While training a deep learning model can take hours even with a dedicated GPU, choosing a high number of folds can

prove to be untractable.

Setting k equal to 8, 10 or 20 was experimentally proven to achieve good results [Kohavi [1995]].

3.4 Previous work

While the drug-target interaction problen was presented already, the current methods used to solve it were not shown so far. The work in the following chapter was not performed in vaccuum, but was preceded by previous research upon which it builds. For that reason, several successful papers are here presented. These were chosen to showcase problems related to DTI and their corresponding answers. As such they focus mainly on used methods and the reasoning behind them, rather than on experiment results. Relevant evaluations are presented later on.

One of the first cases of machine learning being used to predict drug-target interaction was *Supervised prediction of drug–target interactions using bipartite local models* [Bleakley and Yamanishi [2009]]. Drugs and targets were represented as set $D = \{d_1, d_2, \ldots d_m\}$, respectively $T = \{t_1, t_2, \ldots t_n\}$. These sets then together form a bipartite graph where each existing edge is a known interaction. For each predicted edge (d, t), 2 local models are trained. First is trained on all targets except t. Target is considered positive, if it is connected to d via an edge and negative, if it isn't. The second model is trained on all drugs except d in similar fashion. In this particular research paper, the chosen learning algorithm was SVM. These trained models are used to predict results for t and d. Those results are then aggregated to generate a final prediction.

There was quite a large influx of solutions for drug-target interaction from Recommender Systems or Collaborative filtering to be more precise. *Collaborative filtering* is a method used to predict an interest of a set of users in a set of items [Sarwar et al. [2001]]. The underlying principle is trying to find similarities between users and similarities between items, for example represented by vectors in a high-dimensional space, and then making predictions for an entity based on the history of its neighbours. An example is Youtube which recommends a different sequence of videos to each user based on their viewing history. This has been a field that has gained a large attention in recent years as many companies like Amazon or Google are willing to invest large sums of money to improve their predictions in this area. Probably the most famous case is the Netflix prize whose contestants competed for 1 million US dollars as they tried to improve the Netflix recommendation engine [Bennett et al. [2007]].

Research in this area has been quite influential on machine learning approaches to drug-target interaction. This is true as proteins can be seen as users potentially interested in items, here drugs, based on their previously recorded activity. Based on a more general definition, collaborative filtering can be seen as filtering for information in a multi-agent system environment [Terveen and Hill [2001]]. In that case, no metaphore is needed and DTI is just a specific case of collaborative filtering. Either way, research into recommender systems was influential to DTI.

The several following subsections revolve around previous models used for drug-target interaction prediction. They are not the only ones. Apart from the already noted Bleakley and Yamanishi [2009], there are others like Öztürk

et al. [2018] or van Laarhoven et al. [2011]. What sets apart those presented below is either an exposition of characteristics and issues important to drug-target interaction or interesting machine learning approaches utilized to solve DTI.

3.4.1 Neighborhood Regularized Logistic Matrix Factorization for Drug-Target Interaction Prediction [Liu et al. [2016]]

The goal of collaborative filtering is to predict new interactions between two sets of objects S_U and S_V based on the interactions that are already known. One of the approaches to this task is to represent each object with a vector of constant length. So for every $u \in S_U \ \exists \ w_u \in \mathbb{R}^k$. Similarly for every $v \in V \ \exists \ w_v \in \mathbb{R}^k$. In this case these representations are not known a priori. They are not based on some individual characteristics of their corresponding items. Instead they are meant to model interactions between S_U and S_V, and are called *latent vectors*

One approach for expressing this interaction between $u \in S_U, v \in S_V$ is to take their dot product $u \cdot v = x \in \mathbb{R}$. One of the interpretations of x is considering the value of 0 as modelling lack of interaction. Value of 1 then implies some form of a relation. Taking $\sigma(x)$, also sometimes called *logistic function*, instead of x removes the need to interpret values lower than 0 and higher than 1. Values in between may model the strength of activation or the confidence in given relation. As such this method is called *matrix factorization* in case when we are searching for matrices composed of these latent vectors, or *logistic matrix factorization*[Johnson [2014]] when we afterwards explicitly use a logistic function for normalization.

Vectors for elements of S_U can be compactly expressed in a matrix $U \in \mathbb{R}^{|S_U| \times k}$ as its rows. In the same way elements of $\{w_v \mid v \in S_V\}$ form rows of $V \in \mathbb{R}^{|S_V| \times k}$. All interactions are then represented by a matrix $Y = UV^T \in \mathbb{R}^{|S_U| \times |S_V|}$.

For the learning process to make sense, only some elements of Y are known. These interactions define constraints on U and V, which when specified, generate the rest of Y. The problem can be characterized as solving $\arg\min_{U,V} ||Y - UV^T||_F$ where $|| \cdot ||_F$ is the Frobenius norm. At this point it is useful to notice that until now k remained unspecified. Its size is directly proportionate to the size of the hypothesis set, and as such it is subject to the previously mentioned bias-complexity tradeoff.

To prevent overfitting, values of U and V are penalized by adding a term $\lambda_1||U||$ and $\lambda_2||V||$ respectively to the expression being minimized. $|| \cdot ||$ is again a norm effectively restricting large values of elements in U and V. The final term is then

$$\arg\min_{U,V} ||Y - UV^T||_F + \lambda_1||U|| + \lambda_2||V|| \tag{3.25}$$

A variation of the gradient descent approach can be used to solve this problem.

This is the general method Liu et al. build on. $U \in \mathbb{R}^{m \times k}$ is the matrix representing drugs with their corresponding latent vectors being its rows. $V \in \mathbb{R}^{n \times k}$ is similarly the matrix containing the latent vectors of targets as its rows. $Y \in \{0, 1\}^{m \times n}$ is the matrix of interactions between those two groups where $Y_{ij} = 1$, if and only if it is known that drug i interacts with target j.

During prediction, $p_{ij} = \sigma(u \cdot v)$ models the probability of interaction, and should in ideal case be equal to Y_{ij}. The method in this paper as a whole takes the probabilistic approach as it is the goal to find U, V such that they maximize $P(Y \mid U, V)$ under several constraints. One of the observations the authors make is that more emphasis should be put on positive elements of Y as they correspond to experimentally tested results compared to elements which equal zero. These do not represent a clear result as they may either correspond to experimentally tested lack of interaction or just missing information. For this reason, positive samples are weighted more strongly with constant c empirically tested to behave well with value equal to 5. With this extension, the original version of $P(Y \mid U, V)$ then looks as follows:

$$\begin{aligned} p(Y \mid U, V) = & \prod_{i \in [m], j \in [n], y_{ij}=1} p_{ij}^{c y_{ij}} (1 - p_{ij})^{1-y_{ij}} \times \\ & \prod_{i \in [m], j \in [n], y_{ij}=0} p_{ij}^{c y_{ij}} (1 - p_{ij})^{1-y_{ij}} \\ = & \prod_{i=1}^{m} \prod_{j=1}^{n} p_{ij}^{c y_{ij}} (1 - p_{ij})^{1-y_{ij}} \end{aligned} \tag{3.26}$$

As was the case in the more straight matrix factorization case, some form of regularization is presented to reduce overfitting. In this work U and V are limited by Gaussian priors with zero mean and parametrizable σ^2 that is identical for every element in a latent vector.

$$p(U \mid \sigma_d^2) = \prod_{i=1}^{m} N(u_i \mid 0, \sigma_d^2 I), \; p(V \mid \sigma_d^2) = \prod_{i=1}^{m} N(u_i \mid 0, \sigma_t^2 I), \tag{3.27}$$

Due to the Bayesian rule and the chain rule, the relation

$$p(U, V \mid Y, \sigma_d^2, \sigma_t^2) \propto p(Y \mid U, V) \; p(U \mid \sigma_d^2) \; p(V \mid \sigma_t^2) \tag{3.28}$$

holds. The desired matrices are then computed using gradient descent, to be more precise, AdaGrad algorithm [Duchi et al. [2011]]. As the consecutive multiplications of both $p_{ij}^{c y_{ij}}$ and $(1-p_{ij})^{1-y_{ij}}$, which are both within the interval $[0, 1]$, may prove to be computationally difficult, $\log p(U, V \mid Y, \sigma_d^2, \sigma_t^2)$ is maximized instead. The equation is then as follows:

$$\begin{aligned} \log p(U, V \mid Y, \sigma_d^2, \sigma_t^2) = & \sum_{i=1}^{m} \sum_{j=1}^{n} c y_{ij} u_i v_j^T - (1 + c y_{ij} - y_{ij}) \log(1 + exp(u_i v_j^T)) \\ & - \frac{1}{2\sigma_d^2} \sum_{i=1}^{m} ||u_i||_2^2 - \frac{1}{2\sigma_t^2} \sum_{j=1}^{n} ||v_j||_2^2 + C \end{aligned} \tag{3.29}$$

for constant C independent of both U and V.

So far no information about the relationships between objects of the same class was used. But it can be reasonably expected, and for drug-target interaction it can be also empirically proved, that proteins that are structurally similar, often

bind to the same targets. It would be prudent to use such an information, and Liu et al. did indeed use similarities between both proteins and targets to improve the prediction model. For each object, either protein or drug, they introduce 5 most similar neighbours. Then they add terms to the equation 3.29 penalizing distance of each latent vector from its corresponding neighbours

$$\begin{aligned} &\frac{\alpha}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}||u_i - u_j||_F^2, \\ &\frac{\beta}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} b_{ij}||v_i - v_j||_F^2 \end{aligned} \tag{3.30}$$

where α and β are hyperparameters controlling the relative effect of these terms.

The complete term that is being maximized is then

$$\begin{aligned} &\sum_{i=1}^{m}\sum_{j=1}^{n} cy_{ij}u_i v_j^T - (1 + cy_{ij} - y_{ij})\log(1 + exp(u_i v_j^T)) \\ &-\frac{1}{2\sigma_d^2}\sum_{i=1}^{m}||u_i||_2^2 - \frac{1}{2\sigma_t^2}\sum_{j=1}^{n}||v_j||_2^2 \\ &-\frac{\alpha}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}||u_i - u_j||_F^2 - \frac{\beta}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} b_{ij}||v_i - v_j||_F^2 \end{aligned} \tag{3.31}$$

To recapitulate, the Neighbourhood Regularized Logistic Matrix Factorization approach to prediction of drug-target interaction uses a method originally used for recommender systems. The main idea behind this method is to represent every item with a latent vector. Logistic function is used to limit the range of results for two objects, and information about similar proteins and drugs is incorporated using the neighbourhood penalization term. The resulting matrices U and V are then found by maximizing the final function 3.31 using an algorithm based on a gradient descent.

At the same time the latent vectors of proteins and drugs are multiplied directly, only applying the logistic function to the result. Such an approach can be considered as too restrictive, as a similarity penalization is applied among other terms. A more general function from latent vector representation to "interaction" vectors can improve the model results.

3.4.2]

Drug repositioning by integrating target information through a heterogeneous network model [Wang et al. [2014]]

The previously noted model was successful at the time of its publication as it achieved state-of-the-art results. Nonetheless, there are several shortcomings that limit its performance. Primarily the method utilises only information about proteins and drugs. The paper itself notes use of 4 datasets. Together they consist of 989 proteins and 932 drugs with only 5127 documented interactions in total. That's only about 0.2% of all possible interactions. There exists additional

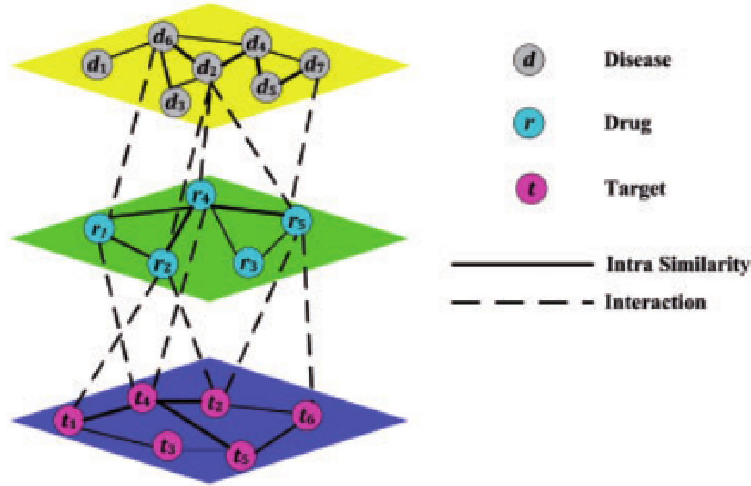

Figure 3.13: Relations among all three groups of data. Source: Wang et al. [2014]

information in the form of relationships between both drugs and proteins on one side and diseases or side-effects on the other. This data remains unused.

The following model aims to alleviate this issue by using several types of interactions. It also heavily depends on the transitive property of what they call a guilt-by-association relation [Barabási et al. [2011]]. This is just another term for the Swanson's ABC model. Basically, it means that if objects A and B interact, and B and C interact, then that increases the probability of interaction between A and C.

It is important to note that the aim of this model is not to predict interactions between drugs and targets, but instead between drugs and diseases. Nonetheless, it is not difficult to modify the algorithm for the former case.

Wang et al. work with interaction data best conceptualized as a graph. It consists of nodes of three types, diseases, drugs, and targets. These are denoted with D, R and T. Information about interactions are represented in E_{dr} and E_{rt} that contain edges between diseases and drugs and drugs and targets respectively. Similarities are encoded by E_{dd}, E_{rr}, and E_{tt}. Additionally the magnitudes of these relations are stored in weight matrices W_{dr}, W_{rt}, W_{dd}, W_{rr}, and W_{tt}.

The algorithm itself is iterative in nature. It takes the normalized weight matrices, and then using the guilt-by-association principle it converges to a steady state of the system. One case of this principle is here realized by the following equation for disease d and protein r.

$$w(d, r) = \sum_{t_i \in T} \sum_{t_j \in T} w(d, t_i) \times w(t_i, t_j) \times w(r, t_j) \tag{3.32}$$

Partial changes of this kind are done together by multiplying the individual matrices. For example, the corresponding "bulk" operation in $(k+1)$-th step for 3.4.2 is $W_{dr}^{k+1} := W_{dt}^{k} \times W_{tt}^{k} \times {W_{rt}^{k}}^{T}$. Together they are all listed below.

$$\begin{aligned} W_{dt}^{k+1} &:= W_{dr}^{k} \times W_{rr}^{k} \times W_{rt}^{k}, \\ W_{dr}^{k+1} &:= W_{dt}^{k} \times W_{tt}^{k} \times {W_{rt}^{k}}^{T}, \\ W_{rt}^{k+1} &:= {W_{dr}^{k}}^{T} \times W_{dd}^{k} \times W_{dt}^{k} \end{aligned} \tag{3.33}$$

Matrix W_{dt} is not originally present so it is used only as a shorthand, that is used in the other two equations. When W_{dt} is replaced by the right side in 3.33, the results look like so:

$$\begin{aligned} W_{dr}^{k+1} &:= (W_{dr}^{k} \times W_{rr}^{k} \times W_{rt}^{k}) \times W_{tt}^{k} \times {W_{rt}^{k}}^{T}, \\ W_{rt}^{k+1} &:= {W_{dr}^{k}}^{T} \times W_{dd}^{k} \times (W_{dr}^{k} \times W_{rr}^{k} \times W_{rt}^{k}) \end{aligned} \tag{3.34}$$

Although these two equations could denote the update rules in this form, the authors decided to put more weight on the original weight matrices. Thus they use a decay factor $\alpha \in (0, 1)$ that controls how conservative is the model in its predictions.

$$\begin{aligned} W_{dr}^{k+1} &:= \alpha(W_{dr}^{k} \times W_{rr} \times W_{rt}^{k} \times W_{tt} \times {W_{rt}^{k}}^{T}) + (1-\alpha)W_{dr}^{0}, \\ W_{rt}^{k+1} &:= \alpha({W_{dr}^{k}}^{T} \times W_{dd} \times W_{dr}^{k} \times W_{rr} \times W_{rt}^{k}) + (1-\alpha)W_{rt}^{0}, \end{aligned} \tag{3.35}$$

As it was previously noted, the model will converge given a proper normalization of all matrices. This normalization is defined by the equation 3.4.2 where $a_{ij} \in A$, for A being a matrix of size $m \times n$. The element a'_{ij} is then an element of the normalized matrix A' of the same size. The supplementary materials of this research paper do contain a full proof for this assertion. Here it is omitted for the sake of brevity.

$$a'_{ij} = \frac{a_{ij}}{\sqrt{\prod_{k=1}^{m} a_{ik} \prod_{k=1}^{n} a_{kj}}} \tag{3.36}$$

One of the drawbacks of this method is its inability to make predictions about drugs or diseases without any known interactions. For such cases, authors advise to use the previously described algorithm to generate weights for objects with interactions and then use these results as a spring board for an algorithm that does not share this shortcoming. Notably authors recommend their previous work, Wang et al. [2013].

Despite its simplicity, this model achieves almost surprisingly good results. Compared to the previous research paper utilizing matrix factorization, this algorithm works directly with graphs. Representing knowledge in this way has certainly its advantages. Additionally, previous iterative approach demonstrates the effectiveness of explicitly using the Swanson's ABC model in machine learning approaches.

3.4.3 NeoDTI: Neural integration of neighbor information from a heterogeneous network for discovering new drug-target interactions [Xiao et al. [2018]]

The previous research paper is capable of effectively using heterogenous sources of data using the iterative approach. This is indeed a positive feature, but its architecture is arguably too simple. NeoDTI, another drug-target prediction algorithm, aims to utilize data from multiple sources represented as a graph, but it

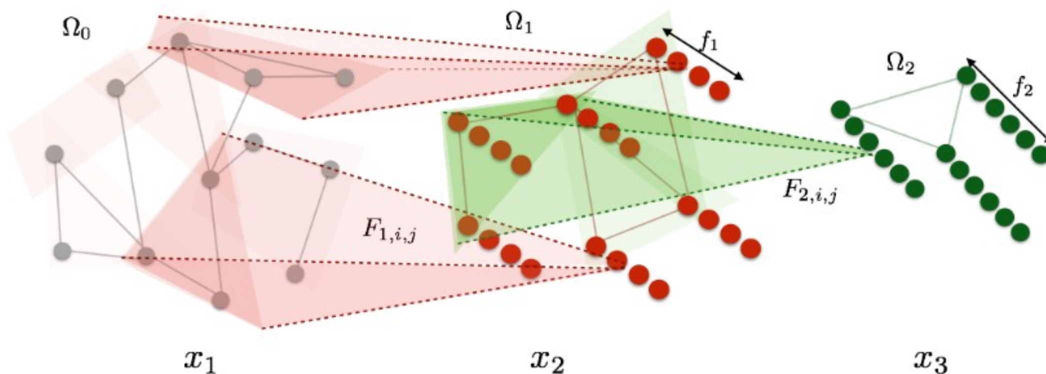

Figure 3.14: Convolutional neural network with 3 layers. Ω_0, Ω_1 and Ω_2 denote the sets of nodes for corresponding level. f_1 and f_2 denote the dimension of node vectors for layer 1 and 2 respectively. Convolutional kernel functions are denoted with F_1 and F_2. Source: [Bruna et al. [2014]]

uses latent vectors similarly to the first featured reaseach paper. It builds upon a preceding work *A network integration approach for drug-target interaction prediction and computational drug repositioning from heterogeneous information* [Luo et al. [2017]]. The main difference is that this model takes the form of a neural network, and is thus trained using backpropagation.

Typically graph-based data present a challenge for the more traditional feed-forward neural networks with fixed input sizes. Yet they can naturaly present connected heterogenous data, and so they pose an opportunity for ML approaches geared towards drug-target interaction or recommendation systems.

An example of a neural network tailored to work with graphs is the *graph convolutional neural network*. The previously presented convolutional neural network in 3.2 can be considered to take special types of graphs as input. To be more precise, it works with multidimensional matrices where each element represents a node in a graph. Each node is then connected to its neighbours with an edge. Convolutional networks that explicitly work with graph structures then accept input graphs of more general form, but in principle their mechanism is the same. Neighbourhoods are defined on each layer except of the last one. These neighbourhoods are then contracted using a layer-specific uniform function. Typically the dimension of a vector for each node grows as the number of nodes decreases. This process is nicely visualized on a figure 3.14 taken from the research paper Spectral Networks and Locally Connected Networks on Graphs [Bruna et al. [2014]] that discusses convolutional networks more broadly. Other examples of neural networks working with graphs can be found in Ying et al. [2018] or Gilmer et al. [2017].

Data used by Xiao et al. for drug-target interaction are the same that were already presented in section 2.3. To briefly re-introduce them, they consist of interactions between 4 distinct groups of objects. Those are proteins, drugs, diseases, and side-effects. As the model is used to predict matches between proteins and drugs, the matrix consisting of these interactions is present. But further matchings of type protein-protein, protein-disease, drug-drug, drug-disease, and drug-side-effect are also utilized. Set of all these relation types is denoted as R.

The neural network model used in this research paper represents the interaction information as a graph. All distinct objects, be it proteins, drugs or the auxiliary items, form the set of vertices V. Positive interactions of all relation types from R form edges E. So $(u, v) \in E$ if and only if there is a known interaction between objects u and v. Together V and E form $G = (V, E)$. Additionally the function $s : E \to \mathbb{R}^+$ assigns edges their weights.

Again each object is represented with a latent vector, and individual vectors are assigned to vertices from V using a function $f^0 : V \to \mathbb{R}^d$. The crucial difference is that instead of using only the latent vectors to perform matrix factorization, a neural network is used that also utilizes latent vectors of direct neighbours. This added information is defined like this:

$$a_v = \sum_{r \in R} \sum_{\substack{e=(u,v) \in E \\ e \text{ of type } r}} \frac{s(e)}{M_{v,r}} \sigma(W_r f^0(u) + b_r) \tag{3.37}$$

$W_r \in \mathbb{R}^{d \times d}$ and $b_r \in \mathbb{R}^d$ used in the previous equation are learned parameters that are randomly initialized. σ is a non-linear function, in this case $ReLU(x) = max(0, x)$, applied element-wise. $M_{v,r}$ is a normalization factor that for all neighbours of vertex v of type r averages their results. Arguably this can be seen as constructing a new vector in the place of the original latent vectors representing their aggregate information. Precisely it is defined as so:

$$M_{v,r} = \sum_{\{e=(u,v) \mid e \in E, \ e \text{ of type } R\}} s(e) \tag{3.38}$$

$a_v \in \mathbb{R}^d$ is then a vector representing the neighbourhood of vertex v. To merge latent vector $f^0(v)$ and a_v, those two vectors are concatenated. The result is then once again transformed using $W^1 \in \mathbb{R}^{2d \times d}$, $b^1 \in \mathbb{R}^{\times d}$ and the already defined σ. The new vector is normalized to form the resulting representation vector $f^1(v)$ used directly for interaction prediction.

$$f^1(v) = \frac{\sigma(W^1 \text{concat}(f^0(v), a_v) + b^1)}{||\sigma(W^1 \text{concat}(f^0(v), a_v) + b^1)||_2} \tag{3.39}$$

Due to normalization, $|f^1(v)| = 1$ for every $v \in V$. This means that all $f^1(v)$ lie on a unit sphere with its central point in 0. Also do note the stark difference in the amount of learnable parameters between matrix factorization and NeoDTI. Apart from latent vectors themselves, NeoDTI also defines W^1, b^1, and W_r and b_r for every $r \in R$ as learnable parameters just to get representation vectors $f^1(v)$.

The following term defines the loss function that is being optimized as well as all the learnable parameters. Notably instead of using directly the dot product $f^1(u) \cdot (f^1(v))^T$ to predict interaction, a linear projection defined by a matrix $G_r \in \mathbb{R}^{d \times d}$ is utilized. L_2 norm is used as an error function.

$$\min_{\substack{\{f^0(v), W^1, b^1, W_r, b_r, \\ G_r \mid v \in V, r \in R\}}} \sum_{e=(u,v) \in E, e \text{ is of type } r} (s(e) - f^1(u)^T G_r f^1(v))^2 \tag{3.40}$$

All the defined components, notably $f^0(v)$, $f^1(v)$ and a_v have properly defined derivatives. Thus the equation 3.4.3 can be used for parameter learning based on gradient descent. For weight training, the authors used the Adam optimizer

[Kingma and Ba [2014]]. With a trained model, the term $f^1(u)^T G f^1(v)$ can be used to predict interactions between protein u and drug v. Positive value corresponds to a match while 0 implies no effect between u and v.

Several performance evaluation strategies were tested. Notably during one variation of the experiment, the number of negative examples used for training was limited so the ratio of positives and negatives in the training set was 1:10. In all cases NeoDTI achieved better results that the regularized matrix factorization by Liu et al.

In summary, NeoDTI builds on the matrix factorization approach by using latent vectors to represent individual objects. It creates a more flexible model for predicting interactions, and successfully aggregates local neighbourhoods of graph vertices. This allows it to achieve state-of-the-art results.

4. Model

What follows is a chapter about the development of a machine learning model used for prediction of interactions between drugs and proteins. Known interactions will form the dataset further reinforced by additional types of connections with other types of discrete nodes, namely diseases and side effects. Used data were taken from the previously referenced NeoDTI paper, and are in more detail discussed in chapter 2. There are several versions of a model presented here. Together they share a large part of their experiment hyperparameters, implementation details, and evaluation methods, unless explicitly mentioned otherwise. The common core is described in section 4.7.

4.1 Considerations and influences

In the beginning the lack of a precise problem description should be noted. As with many other machine learning tasks, where deep learning was successfully utilized, the problem, or rather the input-output distribution, is not well mathematically defined. The list of visible proteins and drugs is not extensive, and the information about interactions is quite sparse. In addition, there is the implicit assumption that all drug-target pairs lacking positive edge are known not to interact. In the end, this works due to the large imbalance between positive and negative edges, but it introduces errors into the dataset. Such a disparity between actual interaction states and those presented by the dataset almost certainly negatively affects the learned model for any kind of a machine learning algorithm. To make things even worse, the considered model works only with binary states. Either there is an interaction, or there isn't. This is a simplification as the strength of an interaction is also a valid and potentially important value.

Due to these imperfections there does not seem to be a method to methodically analyze the dataset and select the best model. Instead the best current possibility seems to be to note parallels with similar problems, incorporate attributes of previously successful approaches, and iteratively test potentially successful models. The rest of this section lists some of the more important influences for the drug-target interaction problem. These also present a sort of an overview of several of the themes presented in the previous parts of this work.

The first crucial observation is that if protein A and protein B both interact with drug C and protein A interacts with drug D, then the likelihood of B-D interaction is increased. This assumption can be based for example on the lock-key model previously mentioned in section 2.2 as it can be hypothetized that both A and B share a "lock" on their surfaces for C, and this lock can be also "opened" by D. This principle directly allows for collaborative filtering approaches, and consequently machine learning approaches for collaborative filtering like matrix factorization. In that ML approach, likelihood increase is expressed by pulling vectors for A and B closer. Similarly for C and D.

As it was already presented in 2.3, the information about known interactions is rather sparse. This is a limitation for the use of more complex, and thus possibly more accurate, machine learning algorithms. As such, utilizing other

source of data can be lead to a notable increase in performance. One of the possible sources of relevant data are interactions of proteins or drugs with other entities, e.g. diseases or side-effects as in Xiao et al. [2018].

The empirical reasoning behind the utility of these other types of interactions is formalized in the Swanson's ABC model [Swanson [1991]]. This principle allows for indirect ties between drugs and proteins, or even drug-drug ties or protein-protein ties. Actually due to the transivity of this rule, it can also support the increased likelihood of B-D connection from the preceding example.

As for the actual machine learning algorithms, there are multiple alternatives to the simple matrix factorization approach. One of them is sketched in Xiao et al. [2018], where a neural network forms a preprocessing step before dot product operation that results in interaction strength further processed by a sigmoid function. Graph convolutional networks formalize local and uniform modifications on graphs, and can be also utilized to process the graph-based interaction data that form the input of a machine learning algorithm for prediction of drug-target interactions.

4.2 Modified logistic matrix factorization

The model described in this chapter will follow the general schema of a matrix factorization. That means that each entity, be it a protein or a drug, will be represented by a latent vector. This vector will be in the beginning randomly initialized. These latent vectors will then be optionally modified to incorporate other auxiliary information. Each valid pair of these modified latent vectors, e.g protein-drug, will then be used to predict interaction, or lack thereof.

In this section latent vectors will be kept unmodified and only the last step, most similar to pure logistic matrix factorization, will be used. This approach has two reasons. Primarily to set a baseline with which following, arguably more complex models can be compared. Secondarily, to allow for gradual explanation of model's properties.

To refresh the previously explained concept, and to define the terminology used from now on, a brief description of logistic matrix factorization is given. Assume the existence of two sequences of entities of different types A and B and their corresponding matrices $M_A \in \mathbb{R}^{t \times |A|}$ and $M_B \in \mathbb{R}^{t \times |B|}$ for $t \in \mathbb{N}$. Each row in M_A represents a latent vector of an element of A, and each row in M_B represents a latent vector of an element of B. Assuming a known interaction matrix of elements I_{AB}, the desired matrices M_A and M_B can be found by solving the equation

$$\arg \min_{M_A, M_B} ||I_{AB} - M_A M_B^T||_F + \lambda_1 ||M_A||_F + \lambda_2 ||M_B||_F \tag{4.1}$$

where $||\cdot||_F$ is a Frobenius (L2) norm while λ_1 and λ_2 are constant regularization terms. The value of I_{AB} at position (i, j) corresponds to the interaction of elements A_i and B_j. It is the scalar product of their corresponding latent vectors that are meant to approximate I_{AB}.

Unfortunately this equation is not too useful as it is hardly the case that a complete I_{AB} is known. Typically, it is exactly matrix factorization that aims to approximate unknown elements of I_{AB}. Additionally, vector dot products can

reach quite large values and thus high loss, while elements of I_{AB} are often drawn only from the domain $\{0, 1\}$. An element-wise application of a logistic function σ, otherwise known as sigmoid, is used to deal with this issue. The modified equation dealing with both problems is written below.

$$\arg \min_{M_A, M_B} ||K \circ (I_{AB} - \sigma(M_A M_B^T))||_F + \lambda_1 ||M_A||_F + \lambda_2 ||M_B||_F \tag{4.2}$$

$K \in \{0, 1\}^{|A| \times |B|}$ is a matrix masking undefined interactions. $K_{i,j} = 1$ if and only if position (i, j) in I_{AB} is known. The symbol $\circ$ denotes element-wise multiplication.

This would be a sufficient statement of a drug-target interaction task if there was no additional information provided. But that is not true as disease and side-effects form another two types of entities. Together these four types of elements present in total five different kinds of interactions. While it is only drug-target interactions, that are meant to be found, the other four kinds serve as auxiliary information.

Using multiple interaction types calls again for a modification of the optimized loss function. To allow for higher flexibility of different latent vector behaviours across different interaction types, additional projection matrix $P_{AB} \in \mathbb{R}^{|A| \times |B|}$ is added for each pair of entity sequences (A, B) with a defined interaction. Scalar $b_{AB} \in \mathbb{R}$ works as a standard neural network bias. Modified equation is once again written below.

$$\arg \min_{M_{\text{protein}}, M_{\text{drug}}} \sum_{(A,B) \in T} ||K \circ (I_{AB} - \sigma(M_A P_{AB} M_B^T + b_{AB}))||_F + \lambda_1 ||M_A||_F + \lambda_2 ||M_B||_F \tag{4.3}$$

Set T in this case holds all the entity pairs with interactions. These are protein-drug, protein-protein, protein-disease, drug-disease, and drug-side effect. This equation correctly defines the loss function for the first model iteration. The crucial question here is, how good is this simple solution compared to NeoDTI utilizing several layers of neural network preprocessing? Additionally, what is the optimal size of latent vectors?

A series of experiments was run with a dimension of latent vectors ranging from 128 to 2048. Results are in figure 4.1. At first sight, the larger the dimension of latent vector, the better. There is a significant jump from 128 to 256, and then from 256 to 512 as well. Further on increasing dimension yields little gain. Notably increasing latent vector size from 1024 to 2048 does not achieve statistically significant improvements.

The other interesting observation is that even this rudimentary collaborative filtering methods can yield results on average equivalent or slightly better than NeoDTI given the results from 8-fold cross-validation. This calls into question the usefulness of deep-learning preprocessing of latent vectors.

4.3 Penalized drug-target error

As it was briefly noted previously, other types of edges apart from protein-drug edges form only a supportive, auxiliary role. How closely does the final model

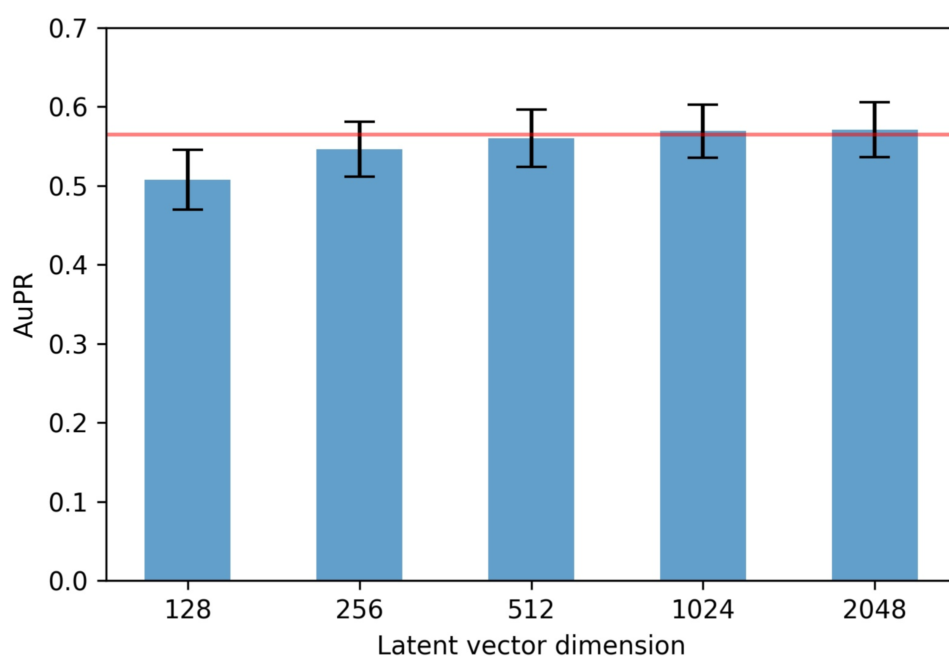

Figure 4.1: A bar chart showing the performance of the modified logistic matrix factorization model, in this case as area under precision-recall curve, relative to the dimension of the latent vectors of given model. Red line delimits the average AuPR of NeoDTI from 8 cross-validation folds. A symmetric range at individual bars denotes +/- single standard deviation.

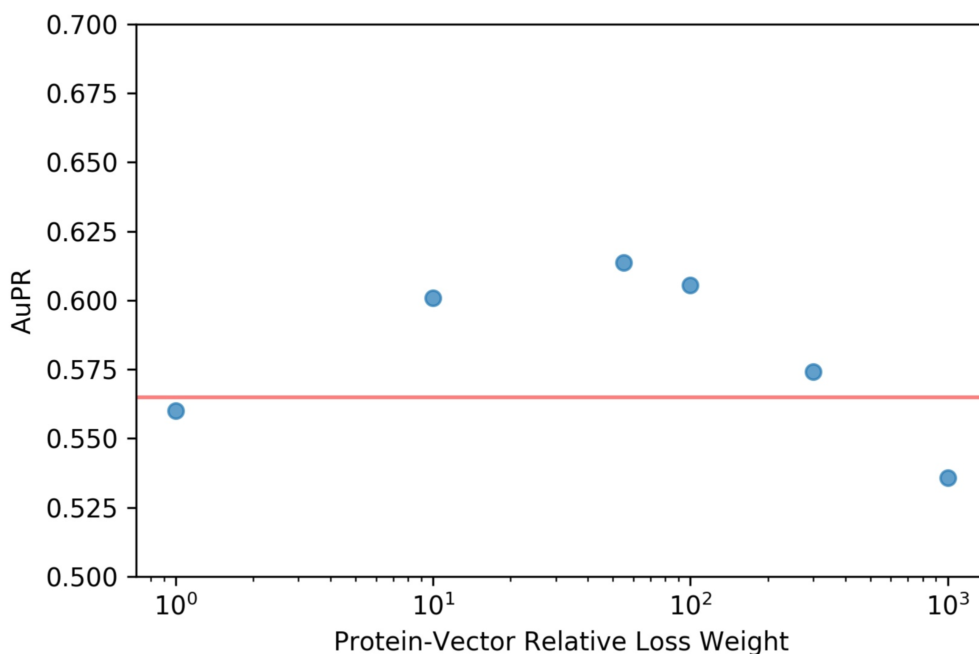

Figure 4.2: Average AuPR for different values of protein-drug loss weight. Be aware of the logarithm scale on x axis, and the restricted range of y axis. Once again NeoDTI performance is denoted by a red horizontal line.

fit their interactions is largely irrelevant. Nonetheless in the previously mentioned model, their interaction loss had the same loss as that of the protein-drug interactions. This rule is not inherent in the model, and can be modified.

Theoretically speaking, there are two opposing factors or forces. On the one hand, the lower the relative weight of protein-drug interaction loss is compared to other edge types, the greater "effort" will model put into fitting interactions that are in the end not useful. On the other hand, the higher it is, the less useful will those supportive edges be. This of course assumes that they present any utility at all. Nonetheless if the Swanson's ABC rule is generally true, then that is a factor to be considered.

In the following experiment, a loss of protein-drug interaction will be multiplied by a single scalar to find out which value is optimal, and where is the balance between the two previously mentioned factors. There is a only a single hyperparameter so there is little risk of overfitting. Also, rather unusually, the multiplication is done within the L2 error. There is no crucial difference though as both versions are apart from the size of the scalar equivalent.

The figure 4.2 presents the results. Overall there were six loss weights tested ranging from 1 to 1000. There was no multiplication loss weight for other edge types, or in other words it was kept equal to 1. Each model was tested with 8-fold cross-validation as before. Latent vector size was set to 512. The trend defined by the measured results supports the hypothesis of two opposing factors.

The maximal AuPR was achieved with a protein-drug loss weight of 55 and was equal to 0.614. It constituted 9.6% percent increase relative to loss weight of 1, and 8.7% increase relative to NeoDTI.

There is a natural follow-up question to these results. What are the optimal weights for other edge types? Considering different importances and subsequent edge type hierarchy implies setting a particular weight for each type. A hyperparameter grid search can be easily utilized to find the results, but such an approach is quite time-consuming. Setting loss weight multiplier for protein-protein interactions to 10 lead to increased AuPR of 0.618, so there is a potential for further improvement.

4.4 Adding descriptors

As it was previously mentioned already several times, the amount of data poses a problem for machine learning in general and more sophisticated deep learning approaches in particular. One of the solutions trying to rectify this issue in this model was presented already. Introducing other types of interactions adds significant amount of other edges, and in the end does increase the model performance. Yet this is not all the information present, that can be theoretically utilized. While, when we already have 3D structure, it is more prudent to use methods like molecular docking, those are not all options. Fingerprints and other types of descriptors briefly presented in section 2.3 form another source of auxiliary data.

Both molecular descriptors and other types of entity descriptors can improve performance as they encode structural and non-structural information relevant from the perspective of drug-target connections. Typically they form a vector of numbers which allows for efficient augmentation of a model.

In this experiment, PubChem fingerprints [Maggiora and Shanmugasundaram [2011]] for drugs and amino-acid frequencies for proteins were added to try to improve the model results. No additional information was provided for either diseases or side-effects. A PubChem fingerprint for a single drug is a vector of 881 boolean bits[1]. Each bit has a fixed meaning relating to the structure of the molecule. For example the bit in the 117-th position is set to 1 if and only if the drug contains at least one saturated or aromatic nitrogen-containing ring of size 3. As for amino-acid frequencies, they consist of relative number of occurences for each particular amino-acid divided by the total amount of amino-acids in protein. Their representing vectors have a dimension of 20.

This combination of auxiliary descriptions was used as it was previously utilized already in Lee et al. [2019]. Nonetheless, a deeper separate analysis would be needed to see if these are the best data sources for machine learning compared to other options.

In the model, both descriptors are appended as constants to variable latent vectors. Resulting vectors are in the case of proteins, diseases, and side-effects padded with zeros to get uniform-width augmented latent vectors. Both fingerprints and amino-acid frequencies do not form only an initialization for latent vectors, but remain fixed during the whole training. Backpropagation does not change them.

Apart from these changes, the following experiment is unmodified compared to the previous iterations. Apart from augmenting latent vectors, only the ma-

[1] For three drugs from 708, a relevant fingerprint was not found. In these cases a zero vector was provided instead.

trices grow larger. As such, this poses a quite simple transformation to utilize added information. Again, 8-fold cross-validation was performed resulting in 0.566 AuPR with standard deviation comparable to the previous experiments. Arguably the model as defined here does not provide enough flexibility to utilize the augmenting data.

4.5 Transformation layers

So far almost no transformation is done to the augmented latent vectors before their corresponding dot product operation. Apart from a single matrix multiplication right at the very end, they are left constant. This approach already achieved non-trivial results, but further modifications can prove to be beneficial especially with appended descriptors for drugs and proteins. The previously oft referenced NeoDTI model does itselfutilize a graph convolutional layer. This section is meant to explore other possibilities of neural network layers, and see if they can achieve better results than its simpler predecessors.

As for the descriptors added in the preceding section, a simple possibility to test their benefit is to use them in conjunction with latent vectors and convolutional layers. Each entity is in this case modified separately using a shared neural network. As such it is a sort of a degenerated form of a convolutional network. Latent vectors for diseases and side-effects are kept constant. The modified vector x' for either drug or protein latent vector x equals $\text{LeakyReLU}(x^T \times W_T + b_T^T)^T$ where $W_T \in \mathbb{R}^{L \times L}$ and $b_T \in \mathbb{R}^L$ are drug-specific or protein-specific model parameters with L being the dimension of already augmented latent vector, i.e. latent vectors potentially with added fingerprint or descriptor for drugs and proteins respectively. The slope of LeakyReLU is set to 0.2.

An issue encountered with this newly enhanced model was overfitting. Model very quickly reached a state, where all validation pairs were classified as zero, i.e. without interaction. This was caused by an unbalanced dataset and usage of the L2 norm. To rectify this issue, two other losses were tested. Logistic loss and binary cross-entropy loss. Logistic loss, defined as $L(y_{true}, y_{pred}) = log(1 + e^{-y_{true} \cdot y_{pred}})$ and used for example in Johnson [2014], did not yield results better than any reasonable baseline. Binary cross-entropy, defined as $L(y_{true}, y_{pred}) = -y_{true} log(y_{pred}) - (1 - y_{true}) log(1 - y_{pred})$, achieved on the other hand non-trivial results. But with AuPR 0.465, it did not have performance better than either NeoDTI or previous best model without a transformation layer. Several modifications were tested to achieve better results. Among them the already noted loss weights, and also higher penalties for misclassifying positives utilized to deal with label unbalance. Nonetheless neither of these changes did achieve a result comparable even to NeoDTI.

The previous model was mainly used to test more sophisticated ways to incorporate additional features rather than to serve as a graph convolutional network. Direct knowledge of local subgraph can prove useful as not all interaction data has to be stored in the latent vector. Ideally, a node would get information from its neighbours already in the forward phase, which could then be aggregated to the vector representing a node built from the original latent vector and received signal. As neighbours also “see” their neighbours, this mechanism creates an

additional path for the Swanson's ABC rule to propagate through the network. Unfortunately, there is a single issue with the current dataset. As it can be seen from histograms 2.5 and 2.6, there is a large difference between degrees of individual nodes. So a uniform graph convolutional layer must handle both vertices with high and low numbers of connections. One, possibly promising approach is to utilize *attention mechanism* to control from which neighbours will a node receive information. In this case, the principle lies in a separate, relatively simple machine learning mechanism, that will decide, which neighbours are important, and which should be silenced. This mechanism is part of the overall model, and as such it is also trained via a backpropagation. This principle was already successfully used in natural language processing, as well as in more similar tasks. An example of a machine learning model working with graph convolutional networks utilizing attention is Liu et al. [2019], which uses segmented textual data to classify entities in structured documents like receipts or forms.

The following model works similarly to the previous one. It utilizes transformed latent vectors and subsequent modified logistic matrix factorization described in 4.2 to make predictions. But instead of a single tranformation layer, a graph convolutional layer with attention is used. It aggregates information from those nodes, with which a selected vertex has a connection, i.e. those that interact with it.

From a high-level perspective, each vertex v_i aggregates its own latent vector x_i and already transformed vectors $\bar{x}_i^{T_1}, \dots \bar{x}_i^{T_k}$ with one vector $\bar{x}_i^{T}$ for each neighbouring entity type T in a graph. So for example a disease would have its latent vector, and then 2 vectors each representing its neighbours of a given type. In this case proteins and drugs. All these vectors are then concatenated[2], and turned into a new vector by a neural network layer. The whole operation is described in the equation below.

$$x_i' = LeakyReLU((x_i || \bar{x}_i^{T_1} || \dots || \bar{x}_i^{T_k}) W_{T^{(i)}} + b_{T^{(i)}}) \tag{4.4}$$

$T^{(i)}$ in this case denotes the entity type of vertex v_i. So each entity type T has its own $W_T \in \mathbb{R}^{(k_T+1)L \times L}$ and $b_T \in \mathbb{R}^L$, where L is the dimension of all latent vectors, and k_T is the number of all possible neighbouring entity types.

As for the transformed entity type vector $\bar{x}_i^T$, it is created by modifying the original latent vectors of neighbouring nodes of type T, and then summing the results, with each being multiplied by an attention weight.

$$\bar{x}_i^T = \sum_{j \text{ interacts with } i,\ T^{(j)}=T} \bar{\alpha}_{ij} (\bar{W}_{T^{(i)},T} x_j + \bar{b}_{T^{(i)},T}) \tag{4.5}$$

$\bar{W}_{T^{(i)},T} \in \mathbb{R}^{L \times L}$ and $\bar{b}_{T^{(i)},T} \in \mathbb{R}^L$ define the original transformation.
Attention weights are first defined by a simple network, where $v_T, w_T \in \mathbb{R}^L$.

$$\alpha_{ij} = ReLU(v_{T^{(i)}}^T x_i + w_{T^{(j)}}^T x_j + b_\alpha) \tag{4.6}$$

Then they are normalized so only the important ones are propagated into the

[2] In the following equation, concatenation is denoted by the symbol $||$.

Name	AUC	AuPR
NeoDTI	0.915	0.565
Latent vector size 512	0.928	0.560
Latent vector size 2048	**0.940**	0.571
Latent vector size 512, modified loss weights	0.934	**0.614**
Latent vector size 512, added descriptors	0.892	0.566
Latent vector size 512, 1 transformation layer	0.877	0.465
Latent vector size 512, 1 attention layer	0.812	0.476

Table 4.1: Result summary

transformed entity type vector. Do note that $\bar{\alpha}_{ij}$ does not necessarily equal $\bar{\alpha}_{ji}$.

$$\bar{\alpha}_{ij} = \frac{\alpha_{ij}}{\sum\limits_{k \text{ interacts with } i,\ T^{(j)}=T^{(k)}} \alpha_{ik}} \tag{4.7}$$

To run this model, a binary cross-entropy loss was used once again with latent vector dimension set to 512. The resulting AuPR was 0.476, which, while being marginally better than the preceding model, does not compare favourably to the currently best model. While modifying loss weights did originally achieve notable performance increase, no hyperparameter setting was sufficient to exceed this AuPR.

As a sidenote, both for the previous transformation layer model, and for the model with attention layer, stacking several layers on top was tested, but resulted in significant performance drop both in AUC and in AuPR. Also both layers presented in this section retain the latent vector dimension even for the transformed result. Getting rid of this invariant, no statistically significant increase was observed when running a simple train/test validation split.

4.6 Final evaluation

Overall, the goal of this work was to develop a model that would push the state-of-the-art for prediction of interacting drug-target pairs. The main area of focus was deep learning, yet the current best model does not utilize principles generally associated with that topic.

In the table 4.1, the aggregated results can be seen. While the model with modified loss weights did perform a little bit worse than its simpler predecessor in AUC, its AuPR is notably better. Notably it scored higher than the state-of-the-art approach of NeoDTI in both metrics. For AuPR this amounted to increase of 8.7%.

An interesting fact is the low performance of both models inspired by deep learning, although both utilized only a single additional layer. Several loss functions and other modifications were tested, yet there might be a solution that would alleviate at least some of the issues. This evaluation can be partly applied also to NeoDTI as most of its weights seem redundant in comparison to simpler models with equivalent AuPR. It begs the question if stacking multiple layers on top of each other, at least partially inspired by conceptually different image recognition tasks, is the correct approach for relatively small datasets. An

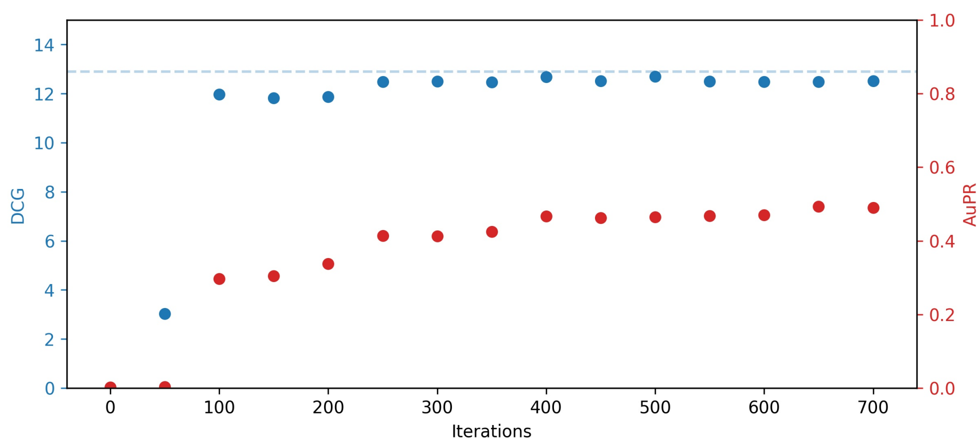

Figure 4.3: A development of AuPR and DCG in first 750 iterations of a single cross-validation fold. Dashed blue line denotes the theoretical maximum of DCG_{50}.

alternative may be a wider, rather than a deeper, network efficiently utilizing multiple types of data. An approach for extracting information from documents and research papers might prove worth the effort as these sources contain data not readily accessible in databases like DrugBank etc.

In addition to AuPR, Discounted Cumulative Gain, itself described in 3.3.1, can provide further interesting information. Namely, the model itself should be in ideal case used to highlight potential candidates worthy of further research. As whole output of the model is a set of interaction strengths, it is desirable to found out if the first k most likely options are really positive. Those are in the end the most relevant outputs. Drug-target pairs with the strongest interaction signals are considered to be the most likely candidates here. Relevance of a pair is in this case set to 1 for positive interactions and 0 for negative ones, i.e. those pairs that lack an interaction. DCG_{50} is measured on a validation fold. In figure 4.3[3], it can be seen that the top 50 most likely predictions do indeed in most cases contain true positives as $max\ DCG_{50} = \sum_{i=1}^{50} \frac{1}{log_2(i+1)} = 12.898$. Additionally, DCG_{50} grows quite a bit faster than AuPR. That implies that a model can relatively quickly filter out non-interacting pairs from the "top".

A machine learning prediction model can be indeed used just for preliminary search and prioritization. As it was mentioned in the preface, there are other, more precise experiments for which an ML model can filter candidates. Nonetheless, there is always the possibility of direct prediction. Seemingly, such a performance can be measured by simply dividing the current data, and running evaluations on the validation set. But arguably the currently undetected interactions differ from those in the dataset as progressively the process of testing and

[3]The sharp growth was observable in all cross-validation folds despite the fact that the figure contains only data from a single fold.

[4]Dopamine receptors D1, D2, D3, D4 have at least medium affinity with Asenapine, while only the interaction with D5 remains to be uncertain [Rampino et al. [2018]]

	Drug	Target	Truly interacting
1.	Diazepam	GABRA4	
2.	Estazolam	GABRA4	Yes (sub-unit of GABA-A receptor)
3.	Risperidone	HTR1B	
4.	Amoxapine	HTR2A	Yes
5.	Pentazocine	OPRD1	
6.	Methadone	OPRK1	
7.	Methylnaltrexone	OPRD1	
8.	Asenapine	HTR1D	
9.	Asenapine	DRD5	Likely[4]
10.	Pergolide	HTR7	
	...		
12.	Meperidine	ORPM1	Yes
14.	Atenolol	ADRB2	Yes

Table 4.2: A table with most likely interaction pairs, which are themselves not in the Drugbank 3.0 dataset. Results from the final model.

uncovering new pairs gets more difficult and more expensive. The low-hanging fruit has been already picked, and so a validation set distribution does not necessarily match the distribution of currently undiscovered interactions.

Thankfully, the DrugBank 3.0 database with drug-target interactions used in the preceding experiments comes from 2011, and as such it is possible to compare it with the current, updated version. That way the model has never seen the current database state, and so it can use it as an interesting, although imprecise validation set. Evaluation is again performed using a 8-fold cross-validation. After each finished training iteration, validation fold predictions are kept for pairs marked as non-interacting by the old dataset, and in the end aggregated together. This way a model is not biased against these individual "false positives", while each pair is only in a single validation fold.

Table 4.2 contains those drug-target pairs, that were not referenced in DrugBank 3.0 dataset as interacting, but that did have the strongest positive model predictions. While not even half in the first 10 was really positive, one must consider how broad was the database already in 2011. It is also necessary to keep in mind, that while an interaction might not be in the current version of the DrugBank database, it does not mean, that it doesn't exist.

4.7 Experiment details

Code for the experiments was written exclusively in Python 3.6 with Tensorflow 1.14.0 being the machine learning library used for gradient descent, and with Ubuntu 18.04 being the operating system. Tensorflow utilized an Nvidia GTX 1080 graphical processing unit for computation.

As previously noted, the dataset was taken from Xiao et al. [2018] to ease evaluation and comparison. It consists of 5 types of edges. Apart from the predicted protein-drug interactions, there are also interactions of type protein-protein, protein-disease, drug-disease, and drug-side effect. All four auxiliary data sources were kept whole, i.e. they were not divided to form a training or a testing

dataset. Protein-drug interactions were randomly split into 8 cross-validation folds for experiment evaluation. There were no other restrictions apart from equal number of individual positive and negative sample counts being distributed across all folds. Experiments were all evaluated with a constant, randomly selected seed for the random generator used to make the cross-validation splits. This was done to make sure that models work with exactly the same data.

As for the loss function, it was a sum of two components, optimized target loss and regularization loss. Optimized target loss consisted of L2 difference between correct interaction values on test set and the model's predictions for all edge types. Alternatively as noted above, binary cross-entropy loss was used instead for models described in section 4.5. Relative weight for differing entity losses was not always equal as is in more detail explained in section 4.3. Regularization L2 loss was used for all variable components of the model, notably latent vectors.

As a gradient descent optimizer, the Adam optimizer [Kingma and Ba [2014]] was used with a learning rate 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e - 8$. Results were taken after 1500 iterations of backpropagation. Together an average evaluation for a single fold took approximately 20 minutes, which is comparable with NeoDTI. As for the code itself, it is in more detail described in separate README file and in source file comments respectively.

5. Conclusion

The goal of this thesis was to develop a machine learning model for prediction of drug-target interactions. As the name suggests, utilized techniques were meant to explore deep learning approaches to this problem. This direction was informed by the successes of previous state-of-the-art algorithms. A method was in the end really developed with performance, in this case primarily measured using Area under Precision-Recall Curve, exceeding its predecessors on the relevant dataset. Nonetheless deep learning did not play such a crucial role for its success as it was hypothesized previously. Method itself is composed of a series of partial modifications built on the foundation of logistic matrix factorization, rather than a single consolidated, closed machine learning algorithm.

As for further work, there are two basic directions in which to go. On one hand, there are further unexplored possibilities for extending and improving the current model. Apart from simple grid search that in the end proved to be too time-consuming due to the high number of hyperparameters, other types of fingerprints and descriptors can be also possibly utilized to improve model's performance. Similarly data mining approach for building a robust training database can be potentially successful. There is also the question of experimenting with other models, for example those successfully deployed as recommender systems. Due to the relative low amount of available interaction data, hybrid approaches may prove to be an interesting alternative due to their ability of utilizing different data sources.

The other line of development is concerned with a practical use of this model. Generally, it can be used for pre-selection of more accurate, but also more expensive experiments. These have additional concerns of their own. Notably one has to consider experiment price, potential utility of given drug with relation to their relative targets, etc. These concerns can be partially encoded in the model, for example by increasing loss weight for relevant protein-drug families, but can be also explored after the model is trained. An ensemble or a more sophisticated meta-analysis can be done to further decrease the risk of error. Overall there is a series of topics more concerned with bioinformatics rather than with machine learning that may be interesting to explore in the future.

Bibliography

Drugbank. URL https://www.drugbank.ca/about.

MolCalX - molecular docking model and molecular recognition. http://www.molcalx.com.cn/%e5%88%86%e5%ad%90%e5%af%b9%e6%8e%a5%e7%9a%84%e6%a8%a1%e5%9e%8b. Accessed: 2019-04-23.

Fingerprints - screening and similarity, 2011. URL http://www.daylight.com/dayhtml/doc/theory/theory.finger.html.

M. Zouhair Atassi and Ettore Appella. *Methods in Protein Structure Analysis*. Springer, 1995. ISBN 978-1-4899-1031-8. doi: 10.1007/978-1-4899-1031-8.

Jürgen Bajorath. Fingerprint design and molecular complexity effects. http://infochim.u-strasbg.fr/CS3/program/material/Bajorath.pdf, 2008. Accessed: 2019-05-12.

Pedro J Ballester and John BO Mitchell. A machine learning approach to predicting protein–ligand binding affinity with applications to molecular docking. *Bioinformatics*, 26(9):1169–1175, 2010.

Albert-László Barabási, Natali Gulbahce, and Joseph Loscalzo. Network medicine: a network-based approach to human disease. *Nat Rev Genet*, 12 (1):56–68, Jan 2011. ISSN 1471-0064. doi: 10.1038/nrg2918.

James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA, 2007.

Kevin Bleakley and Yoshihiro Yamanishi. Supervised prediction of drug–target interactions using bipartite local models. *Bioinformatics*, 25, 2009.

David D. Boehr, Ruth Nussinov, and Peter E. Wright. The role of dynamic conformational ensembles in biomolecular recognition. *Nature Chemical Biology*, Oct 2009. URL https://doi.org/10.1038/nchembio.232.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014. URL http://arxiv.org/abs/1312.6203.

Michael L Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709–713, 1983.

George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

Allan Peter Davis, Cynthia Grondin Murphy, Robin Johnson, Jean M Lay, Kelley Lennon-Hopkins, Cynthia Saraceni-Richards, Daniela Sciaky, Benjamin L King, Michael C Rosenstein, Thomas C Wiegers, et al. The comparative toxicogenomics database: update 2013. *Nucleic acids research*, 41(D1), 2012.

Joseph A. Dimasi, Henry G. Grabowski, and Ronald W. Hansen. Innovation in the pharmaceutical industry: New estimates of R&D costs. *Journal of Health Economics*, 47, 2016.

Werner Dubitzky, Martin Granzow, and Daniel P Berrar. *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media, 2007.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Farlex. *Farlex Partner Medical Dictionary*. 2012.

Anna Maria Ferrari, Binqing Q Wei, Luca Costantino, and Brian K Shoichet. Soft docking and multiple receptor conformations in virtual screening. *Journal of medicinal chemistry*, 47(21):5076–5084, 2004.

Emil Fischer. Einflus der Configuration auf die Wirkung der Enzyme. *Berichte der deutschen chemischen Gesellschaft*, 27:2985 – 2993, 1894. doi: 10.1002/cber.18940270364.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

Yanming Guo, Yu Liu, Ard Oerlemans, and Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187, November 2015.

Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

Harren Jhoti and Andrew R Leach. *Structure-based drug discovery*. Springer, 2007.

Christopher C Johnson. Logistic matrix factorization for implicit feedback data. *Advances in Neural Information Processing Systems*, 27, 2014.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

Andrej Karpathy. Biological neuron, 2015. URL `https://cs231n.github.io/assets/nn1/neuron.png`.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*, 2018.

TS Keshava Prasad, Renu Goel, Kumaran Kandasamy, Shivakumar Keerthikumar, Sameer Kumar, Suresh Mathivanan, Deepthi Telikicherla, Rajesh Raju, Beema Shafreen, Abhilash Venugopal, et al. Human protein reference database—2009 update. *Nucleic acids research*, 37(suppl_1), 2008.

Mohamed A Khamis, Walid Gomaa, and Walaa F Ahmed. Machine learning in computational docking. *Artificial intelligence in medicine*, 63(3):135–152, 2015.

James. H. Kim and Anthony R. Scialli. Thalidomide: The Tragedy of Birth Defects and the Effective Treatment of Disease. *Toxicological Sciences*, 122, 2011. URL `https://doi.org/10.1093/toxsci/kfr088`.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Craig Knox, Vivian Law, Timothy Jewison, Philip Liu, Son Ly, Alex Frolkis, Allison Pon, Kelly Banco, Christine Mak, Vanessa Neveu, Yannick Djoumbou, Roman Eisner, An Chi Guo, and David S. Wishart. Drugbank 3.0: a comprehensive resource for 'omics' research on drugs. *Nucleic Acids Res*, 39(Database issue), Jan 2011. ISSN 1362-4962. doi: 10.1093/nar/gkq1126.

Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *International Joint Conference on Artificial Intelligence*, 1995.

D. E. Koshland. Application of a theory of enzyme specificity to protein synthesis. *Proc Natl Acad Sci U S A*, 44(2):98–104, Feb 1958. ISSN 0027-8424. URL `https://www.ncbi.nlm.nih.gov/pubmed/16590179`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012a.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012b.

Michael Kuhn, Monica Campillos, Ivica Letunic, Lars Juhl Jensen, and Peer Bork. A side effect resource to capture phenotypic effects of drugs. *Molecular systems biology*, 6(1), 2010.

Nikolay Kyurkchiev and Svetoslav Markov. On the hausdorff distance between the heaviside step function and verhulst logistic function. *Journal of Mathematical Chemistry*, pages 109–119, 2016.

Ingoo Lee, Jongsoo Keum, and Hojung Nam. Deepconv-dti: Prediction of drug-target interactions via deep learning with convolution on protein sequences. *PLoS computational biology*, 15(6):e1007129, 2019.

Xiaojing Liu, Feiyu Gao, Qiong Zhang, and Huasha Zhao. Graph convolution for multimodal information extraction from visually rich documents. *arXiv preprint arXiv:1903.11279*, 2019.

Yong Liu, Min Wu, Chunyan Miao, Peilin Zhao, and Xiao-Li Li. Neighborhood regularized logistic matrix factorization for drug-target interaction prediction. *PLoS Comput Biol*, 12(2), Feb 2016. doi: 10.1371/journal.pcbi.1004760. URL `https://www.ncbi.nlm.nih.gov/pubmed/26872142`.

Yunan Luo, Xinbin Zhao, Jingtian Zhou, Jinglin Yang, Yanqing Zhang, Wenhua Kuang, Jian Peng, Ligong Chen, and Jianyang Zeng. A network integration approach for drug-target interaction prediction and computational drug repositioning from heterogeneous information. *Nature Communications*, 8(1), 2017. ISSN 2041-1723. doi: 10.1038/s41467-017-00680-8.

Gerald M Maggiora and Veerabahu Shanmugasundaram. Molecular similarity measures. In *Chemoinformatics and computational chemical biology*, pages 39–100. Springer, 2011.

Alan E Mark and Wilfred F van Gunsteren. Decomposition of the free energy of a system in terms of specific interactions: implications for theoretical and experimental studies. *Journal of molecular biology*, 240(2):167–176, 1994.

Andrea Mauri, Viviana Consonni, Manuela Pavan, and Roberto Todeschini. Dragon software: An easy approach to molecular descriptor calculations. *Match*, 56(2):237–248, 2006.

Xuan-Yu Meng, Hong-Xing Zhang, Mihaly Mezei, and Meng Cui. Molecular docking: A powerful approach for structure-based drug discovery. *Current computer-aided drug design*, 7:146–57, 06 2011. doi: 10.2174/157340911795677602.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. Second Edition. MIT Press, 2012. ISBN 978-0-262-01825-8.

Mosby. *Mosby's medical dictionary*. Mosby/Elsevier, 2009. ISBN 9780323052900.

J.G. Nicholls, A.R. Martin, P.A. Fuchs, and B.G. Wallace. *From Neuron to Brain*. Sinauer Associates Incorporated, 1999. ISBN 9780878935826. URL `https://books.google.cz/books?id=4spqPgAACAAJ`.

Hakime Öztürk, Arzucan Özgür, and Elif Ozkirimli. Deepdta: deep drug–target binding affinity prediction. *Bioinformatics*, 34(17):i821–i829, 2018.

Martin Pilát. Application of computational intelligence methods lecture notes, 2018. URL `https://github.com/martinpilat/CImethods/blob/master/01Vizualizace/02-Vyhodnocen\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{\OT1\i\global\mathchardef\accent@`

spacefactor\spacefactor}\accent19\OT1\i\egroup\spacefactor\accent@spacefactormodel\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{u\global\mathchardef\accent@spacefactor\spacefactor}\accent23u\egroup\spacefactor\accent@spacefactor.ipynb.

Martin Prokop. Structural bioinformatics - molecular docking. http://ncbr.muni.cz/~martinp/C3210/StructBioinf9.pdf, 2015. Accessed: 2019-04-19.

Antonio Rampino, Aleksandra Marakhovskaia, Tiago Soares-Silva, Silvia Torretta, Federica Veneziani, and Jean Martin Beaulieu. Antipsychotic drug responsiveness and dopamine receptor signaling; old players and new prospects. *Frontiers in psychiatry*, 9, 2018.

Badrul Munir Sarwar, George Karypis, Joseph A Konstan, John Riedl, et al. Item-based collaborative filtering recommendation algorithms. *Www*, 1:285–295, 2001.

G Madhavi Sastry, Matvey Adzhigirey, Tyler Day, Ramakrishna Annabhimoju, and Woody Sherman. Protein and ligand preparation: parameters, protocols, and influence on virtual screening enrichments. *Journal of computer-aided molecular design*, 27(3):221–234, 2013.

Jack W. Scannell, Alex Blanckley, Helen Boldon, and Brian Warrington. Diagnosing the decline in pharmaceutical r&d efficiency. *Nature Reviews Drug Discovery*, 11, Mar 2012. URL https://doi.org/10.1038/nrd3681.

Ben-David Shai Shalev-Shwartz Shai. *Understanind Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014. URL http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/.

David Silver, Aja Huang, Chris J. Maddison, and Arthur et al Guez. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. URL https://doi.org/10.1038/nature16961. Article.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Greg Stuart, Nelson Spruston, Bert Sakmann, and Michael Häusser. Action potential initiation and backpropagation in neurons of the mammalian cns. *Trends in Neurosciences*, 20, March 1997.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Don R Swanson. Complementary structures in disjoint science literatures. In *Annual ACM Conference on Research and Development in Information Retrieval: Proceedings of the 14 th annual international ACM SIGIR conference on Research and development in information retrieval*, volume 13, pages 280–289, 1991.

Alan Talevi. Drug repositioning: current approaches and their implications in the precision medicine era. *Expert Review of Precision Medicine and Drug Development*, 3(1):49–61, 2018. doi: 10.1080/23808993.2018.1424535. URL https://doi.org/10.1080/23808993.2018.1424535.

Loren Terveen and Will Hill. Beyond recommender systems: Helping people help each other. *HCI in the New Millennium*, 1(2001):487–509, 2001.

Roberto Todeschini and Viviana Consonni. *Handbook of molecular descriptors*, volume 11. John Wiley & Sons, 2008.

Twan van Laarhoven, Sander B Nabuurs, and Elena Marchiori. Gaussian interaction profile kernels for predicting drug–target interaction. *Bioinformatics*, 27 (21):3036–3043, 2011.

Wenhui Wang, Sen Yang, and Jing Li. Drug target predictions based on heterogeneous graph inference. *Pac Symp Biocomput*, pages 53–64, 2013. ISSN 2335-6936. URL https://www.ncbi.nlm.nih.gov/pubmed/23424111.

Wenhui Wang, Sen Yang, Xiang Zhang, and Jing Li. Drug repositioning by integrating target information through a heterogeneous network model. *Bioinformatics*, 30(20):2923–2930, 2014. ISSN 1367-4811. doi: 10.1093/bioinformatics/btu403.

Wikipedia contributors. Universal approximation theorem — Wikipedia, the free encyclopedia. URL https://en.wikipedia.org/wiki/Universal_approximation_theorem.

An Xiao, Fangping Wan, Lixiang Hong, Jianyang Zeng, and Tao Jiang. NeoDTI: neural integration of neighbor information from a heterogeneous network for discovering new drug–target interactions. *Bioinformatics*, 35(1), 07 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty543. URL https://dx.doi.org/10.1093/bioinformatics/bty543.

Chun Wei Yap. Padel-descriptor: An open source software to calculate molecular descriptors and fingerprints. *Journal of computational chemistry*, 32(7):1466–1474, 2011.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. *CoRR*, abs/1806.01973, 2018. URL http://arxiv.org/abs/1806.01973.

List of Figures

2.1 Protein conformations 6
2.2 Illustration of the feature-based matching process 9
2.3 Entity connection graph 11
2.4 Node type histogram 12
2.5 Histogram of edge degrees for proteins and drugs 13
2.6 Histogram of edge degrees for drugs and side-effects 14
2.7 Histogram of edge degrees for proteins and drugs on a subgraph with only proteins and drugs 15

3.1 Clustering example 17
3.2 Feature set in R^2 20
3.3 Overfitting example 22
3.4 GAN-generated faces 23
3.5 Biological neuron 24
3.6 Activation functions 26
3.7 Convolutional layer 33
3.8 AlexNet layers 34
3.9 Confusion matrix 36
3.10 3 classification datasets from $\mathbb{R}$ 37
3.11 ROC curve 39
3.12 PR curve 40
3.13 Data relations, Wang et al. [2014] 48
3.14 Convolution on a graph 50

4.1 Latent vector dimension bar chart 56
4.2 Protein-Drug loss weight plot 57
4.3 AuPR and DCG development trend 62