Přírodovědecká fakulta
Faculty of Science

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

BACHELOR THESIS

INTERPRETABILITY OF NEURAL NETWORKS IN DRUG DESIGN

IMPLEMENTING THE INTEGRATED GRADIENTS METHOD

Author
Abd Alkareem ALJEIROUDI

Submission
Institute for Machine Learning

Thesis Supervisor
Univ. Prof. Dr. Sepp Hochreiter

Assistant Thesis Supervisor
Hubert Ramsauer, MSc

March 2020

České Budějovice

Bachelor Thesis

to confer the academic degree of

Bachelor of Science in Bioinformatics

in the Bachelor's Program

Bioinformatics – Bachelor's Program

Aljeiroudi, A. A., 2020: Interpretability of Neural Networks in Drug Design, BSc. Thesis, in English, - 52p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

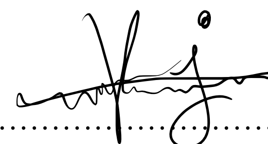
Annotation

Several artificial neural networks were implemented and evaluated. Best network's architecture was selected on the basis of the AUC analysis. Later on, Integrated Gradients (IG) was used to attribute the network's decisions to the learned input. The performance of IG using different baselines was evaluated. IG identifies a number of already known toxicophores listed in the literature.

Affirmation

I hereby declare that I have worked on my bachelor's thesis independently and used only the sources listed in the bibliography. I hereby declare that, in accordance with Article 47b of Act No. 111/1998 in the valid wording, I agree with the publication of my bachelor thesis, in full to be kept in the Faculty of Science archive, in electronic form in a publicly accessible part of the IS STAG database operated by the University of South Bohemia in České Budějovice accessible through its web pages. Further, I agree to the electronic publication of the comments of my supervisor and thesis opponents and the record of the proceedings and results of the thesis defence in accordance with aforementioned Act No. 111/1998. I also agree to the comparison of the text of my thesis with the Theses.cz thesis database operated by the National Registry of University Theses and a plagiarism detection system.

České Budějovice, 20.03.2020



....

Abd Alkareem ALJEIROUDI

Acknowledgments

First and foremost, I would like to thank my supervisor Hubert Ramsauer, MSc as well as my former supervisor Dr. Kristina Preuer, MSc. This work would not have been possible without you. Thanks for being constantly supportive and offering me help on the go.

To my dearest and nearest friend Dipl.-Ing. Alaa Mofleh the one who never allowed me to lose hope a second, thanks for providing me with support and motivation when I needed them most.

My biggest gratitude goes to the love of my life BSc, Nina Troppmair for having stood next to me. The lessons you taught me are incomparable.

Maria Grazia Dascanio, Adnan Aljeiroudi, Obadah Aljeiroudi and Ghaith Aljeiroudi, I show an immense gratitude to you all. Thank you for believing in me and giving me the strength with which I thrive every day.

I am endlessly grateful to all my colleagues at the Institute for Machine Learning in Linz for their helpful comments, insightful work, and exciting discussions.

Abstract

Interpretability in neural networks is a key to derive knowledge for toxicity-related research. Attributing the learned patterns in the input is one way towards Explaining network's decisions. In this work we touch on an important process in drug design, namely mutagenicity detection. In regard to mutagenicity, we explain one of the possible mechanisms that mutagens use in mutating DNA molecules. Next we introduce the Ames test, one of the most common mutagenicity detection tests. We explain that the Ames test has a relatively high interlaboratory reproducibility error. Neural networks can take over this repetitive time-consuming task. After that, we draw attention to the potential of deep learning in drug development, and how it can assist predicting the output of the Ames test. Interpretability in Neural Networks is the idea of relating the model's decisions back to patterns in the input data. We hint how interpretability can help drug development research strive forward. Later we peek into the data at hand and provide a quick description of the data processing techniques employed in our analysis. To encode the data, we use Extended Connectivity Fingerprints, a highly performant variant of the Morgan Fingerprint method. Our 2048-bit vectors are fed to our best trained network. Prior to that, the concept of Feed Forward Networks is recapitulated for better grasp of the topic. Following, we train 30 different networks and evaluate them using the Area Under the ROC curve as an evaluation metric. The hyperparameter settings of these networks are chosen using an optimization technique known as the Bayesian Optimization. In regard to the optimization process, we elaborate on our choice of parameters and justify how the AUC is an adequate metric for our case study. Last but not least, we describe the network's architecture suggested by our optimization technique. Subsequent to that, we revisit attribution methods, more specifically the integrated gradients method. Because integrated gradients is easily applicable and implementable, the method stood out in the recent few years. In that regard, we show both advantages and disadvantages of IG. The choice of baseline has to be made carefully, in order for IG to deliver good results. To investigate that, we try four different baselines: zero-vector, modal, average and random, and compare their results in the subsequent work. Remarks regarding potential numerical errors committed while approximating the integral are made. Furthermore, we provide a recipe to implement IG as well as our own implementation of the method. After that, results show that our best trained model is a good performant with an AUC of **~0.894**, **0.830** on **validation**, **test** set respectively. Next, we move to explaining the model's

output using analysis of IG. Attributions produced by Integrated Gradients are then used to find the weighted atomic contribution for all bits within one molecule. By and by, we make a few remarks about the final results. We show that our findings cohere with the findings of the literature to a high extent. For better comparison of the baselines, we compare the number of true positives with the respect to the number of positively attributed molecules per toxicophores. The findings show that the zero-vector managed to rediscover a high number of the toxicophores listed in the literature and is therefore the most adequate baseline for our research. Finally, our findings are consistent and matches the literature. However, in order for integrated gradients achieve a higher toxicophore rediscovery, there is room in our work for improvement.

Implementation: is available on GitHub https://github.com/kareemjeiroudi/molecules_and_ml

Table of Contents

SWORN DECLARATION I

ACKNOWLEDGMENTS III

ABSTRACT IV

1 INTRODUCTION 1

1.1. MUTAGENICITY 2
1.2. AMES TEST 3
1.3. DEEP LEARNING IN DRUG DESIGN 4
1.3.1. INTERPRETABILITY OF DEEP NEURAL NETWORK 6

2 METHODS 9

2.1. EXTENDED-CONNECTIVITY FINGERPRINTS (ECFP) 10
2.2. FEED FORWARD NEURAL NETWORKS 13
2.2.1. BACKPROPAGATION 16
2.2.2. VANISHING GRADIENT 19
2.2.3. ACTIVATION FUNCTIONS 19
2.3. HYPERPARAMETER OPTIMIZATION 22
2.4. AREA UNDER ROC CURVE (AUC_ROC) 26
2.5. ATTRIBUTION METHODS 28
2.5.1. INTEGRATED GRADIENTS 29
2.5.1.1. Pros 34
2.5.1.2. Cons 34
2.5.1.3. Baseline 35
2.5.1.4. Integral Approximation (Number of Integration Steps) 37

3 RESULTS 39

4 CONCLUSION 51

5 REFERENCES 53

List of Tables

Table 1: number of mutagens and non-mutagens in each of the training, validation and test set. 9

Table 2: types of evaluated hyperparameters by the bayesian optimization as well as their search space. '[]' is used to describe continuous sets, '()' for discrete, and '{}' for categorical ones. 22

Table 3: results of the bayesian optimization. Here are the 10 best scoring models found in our search history. Table is sorted to the auc score in a descending order top-to-bottom. The auc values correspond to the scores obtained by training a new model every time and evaluating it on the validation set. 25

Table 4: first 10 best scoring models evaluated 10 times on the validation set in order to avoid optimistic models. This data is a one-to-one comparison and is not sorted. Model 28 is the best scoring model in our experiments, because the selection criterion implies that the selected model has the highest average auc and lowest standard deviation. 25

Table 5: hyperparameter list of the best scoring network for quick reference. 40

Table 6: matrix containing all average numerical errors (a.n.e.) Corresponding to figure 13, but "zoomed out". Zero-vector baseline had best results for all tested integration steps. Modal baseline on the other hand gave the worst results, which can also be seen in figure 13. Also, higher integration steps had higher precision and therefore less numerical error. 40

Table 7: rediscovery rates of all four tested baselines. Zero vector had the highest rediscovery rate (**0.634**). The random baseline, not surprisingly, had the worst rediscovery rate. 45

Table 8: number of molecules where toxicophore i is found vs. Number of molecules where toxicophore i is positively attributed vs. Number of true positives, where toxicophore i is positively attributed. The reported results are all correspondent to the attributions calculated using the zero-vector baseline. 47

List of Figures

Figure 1: attributions obtained by applying the integrated gradients for two correctly classified samples. The pixels highlighted on the right-hand side are the ones responsible for the model's decision. The higher the intensity, the higher the attribution. All other pixels in the surroundings were given 0 attribution, since they do not contribute to the model's decision at all. 'T' stands for true label, 'P' stands for predicted label. 6

Figure 2: attributions obtained by applying the integrated gradients for two false predictions. See how the attributions correspond to the model's decision. The lower-left stroke in '5' in the first image is tight and makes the digit looks more like '6'. The attributions, in turn, highlight these pixels with high intensity. 'T' stands for true label, 'P' stands for predicted label. 8

Figure 3: class distributions of all training (**top**), validation (**middle**) and test set (**bottom**). Class imbalance is noticeable but is not significant enough to apply additional data preprocessing techniques such as oversampling the minority class or undersampling. Number of mutagens is always higher than that of non-mutagens in all three sets. 10

Figure 4: bit at position 1014 can be located in different molecules (e.g. Mol. 1, 18, 42, 76, 87). All of these molecules are existent in the test supplier (test set). 12

Figure 5: constructed environment after two iterations (radius = 2); once when atom 6 is core, and once when atom 5 is core. Both environments contain exactly the same atoms and bonds. Therefore, they are duplicate information. In comparison to Figure 6, these two core atoms will get different identifiers, and will be represented by the same bit at the same position, but not removed. This is known as *structure duplication*. 12

Figure 6: for both atoms 1 and 9, the identifiers will be identical even after n number of iterations. Not only do they have same atoms and bonds, but the two regions are mirror images to one another. The terminal step works on removing these duplicates. These two atoms are known as *stereoatoms. From a stereochemical point of view, these substructures are not identical.* 12

Figure 7: example of a fully-connected feed forward neural network with 2 hidden layers. Input size is 3. Output is size is 3. Information is propagated through the network's layers. Every neuron in layer l captures specific information (features detector), as the network adjusts its weights at layer l. 15

Figure 8: four examples of common activation functions. Rectified linear units (ReLU) on the **top-left**. Sigmoid to the **top-right**. To the **bottom-left** there is leaky relu. The **bottom-right** shows the hyperbolic tangent activation function. 21

Figure 9: after just a few points, the algorithm constructs a posterior that is close to the true maximum. Notice the difference between exploration and exploitation steps. The former implies exploring the parameter space. The later implies testing points that lie near the current known maximum. The posterior gets closer to the original target with more evaluation points, especially near maxima/minima. At early optimization levels, the algorithm makes fantasies about the shape of the target function (below 3 observations). The confidence interval is reevaluated with every

observation introduced. The uncertainty grows as we go further from the evaluated observations. 23

Figure 10: 10 best scoring models after having them initialized, trained (on training set), and evaluated (on validation set) 10 times. The x-axis shows models ids that match the ones listed in table 3. The y-axis shows their average auc scores. The whiskers represents the standrd deviation in the auc score. 26

Figure 11: example of a receiver operating characteristic (ROC) curve on the **left**. Example of precision-recall plot on the **right**. See how recall and precision are competing objectives. Which metric is preferable depends on the application requirements. 27

Figure 12: interpolating the input increasing in intensity from 0 to 1. Observe changes in the network's decision 30

Figure 13: three possible paths drawn between an arbitrary baseline x' and the original input x. The integrated gradients interpolate over a number of samples s that lie on that straight path that connects the two samples ($P3$). Examine the interpolated sample x_α for comprehension. 29

Figure 14: numerical errors found by each baseline vs. The number of integration steps. The x-axis shows the samples in the test data set. The y-axis represents the numerical error found when calculating the attributions at sample x (i.e. $\left|\left(F(x) - F(x')\right) - \sum_{i=1}^{n} a_i\right|$). The average error over all samples is then calculated (a.n.e.), the same operation is repeated with 50, 60, 70, 80, 90 and 100 integration steps and all four baselines (zero-baseline, modal, average, and random). In general, the zero-baseline had the lowest numerical errors with all integration steps. We can also see that the higher the number of integration steps the lower the error is. This statement, however, does not hold true for endlessly higher number of steps for computational reasons. 37

Figure 15: receiver operating characteristic curves of model 28 (best scoring model). The plot on the left-hand side corresponds to the validation set. The plot to the right corresponds to the test set. The auc scores of each curve is reported on the lower-right corner of the plot 39

Figure 16: comparison of unweighted (**left**) vs. Weighted attributions (**right**). In the figure left, all atoms are equally attributed. That is rarely the case in a fingerprint. The depicted molecule is number 100 in the test set. 41

Figure 17: weighted attributions of 12 randomly selected fingerprints found in molecule 1731 in the test set. Positive attributions are highlighted in red. Negative attributions are in green. The intensity of the color corresponds to how big the attribution value is. ‘P’ stands for predicted label. ‘T’ for true label. See how most substructures were recognized as mutagenic/toxic by our network, all of a reason for our network to classify the molecule as ‘mutagen’. 1 means ‘mutagen’ (positive class), 0 means ‘non-mutagen’ (negative class). 42

Figure 18: weighted attributions of 12 randomly selected fingerprints found in molecule 341 in the test set. The description is identical to Figure 16. However in contrary to Figure 16, here a true negative is depicted. 43

Figure 19: weighted attributions of 12 randomly selected fingerprints found in molecule 692 in the test set. The attributions show that the network understood the molecule as mutagen due to the presence of an aromic nitro. However, the ames test labeled the molecule as ‘non-mutagen’. 44

Figure 20: structural representation of some of the most common toxicophores listed in the literature. Some of these could not be plot due to technical errors. The original document provides a toxicity

index. Equivalently, our network suggests that some are more indicative of toxicity of others. Comparing that against that toxicity level would have been of knowledge. 49

Figure 21: few examples of the toxicophores detected in the test set. On the left side we show the toxicophores formula. On the right side we show 4 examples per toxicophore together with their attributions. Above every example in the plot is a comparison of the true label 'T' vs. The predicted label 'P'. Observe how areas where toxicophores are present are indicating toxicity. 50

Introduction

Drug design is a lengthy, complex, and costly process. The number of challenges this process faces has led to a high uncertainty when approving a newly synthesized drug. The high rate of failed clinical trials, regulatory issues, lack of target proteins and biomarkers are all nothing but a few of these challenges that this lengthy drug design pipeline faces. Let alone the rising costs, lack of knowledge of the underlying mechanisms of certain diseases or patient heterogeneity. In order for a drug to be approved, chemists must make sure that the drug does not hold any adverse or toxic properties. Toxic properties are tightly relatable to chemical structures, sites, atomic arrangement or conformers [1, 2]. This chemical toxic activity takes effect when the molecule of interest binds to a target protein (e.g. protein receptor). We identify these structures as toxicophores. A good amount of these toxicophores is already listed in the literature [2]. This was made possible only with years of chemists' extensive research.

The field of bioinformatics puts computer power at the fingertips of these hardworking chemists and tries to take over repetitive and computationally-demanding work. In the recent years, we have seen machine learning algorithms being employed almost everywhere: from security, to industrial work, medical engineering, health-care, all the way to science. Machine learning is the computer's ability to process data in a desired form, put it in an abstract shape, extract meaningful information out of data, identify patterns and finally make decisions and predictions on the outcome of future data without being explicitly programmed to make these decisions. This is also referred to as generalization. Generalization, in this context, is the ability of a learning machine to perform accurately on newly unseen examples after having seen precedent data. In mathematical terms, we assume that this learning data has generally unknown underlying probability distribution that represent the occurrence probability of each example in the data. A learner has to identify patterns (build a statistical model) in this space that enables it to produce sufficiently accurate predictions in new cases. This goes hand-in-hand with probability reasoning and statistical learning.

The bacterial reverse mutation assay (Ames test) is an essential step in this pipeline that detects potentially mutagenic compounds. The Ames test represents an early alerting system for potential toxicophores that may result in adverse chemical activity in later development processes. It was named after the American biochemist Bruce Ames [3]. This in vitro assay has become the standard test for mutagenicity determination. According to the Ames test, a chemical is Ames test positive, if a genetic damage is detected when the chemical is added to a strain of bacteria. The test enjoys several advantages such as simplicity and ease-of-use, however, the difficult reproducibility of the test hampers its potential. The reproducibility of the Ames test is dependent on the purity of the tested chemical, methods employed as well as additional toxic side effects.

In this work we provide assistance to the drug design process by employing the power of feed-forward networks in predicting the output of the Ames mutagenicity test. We provide a summary of our model optimization strategy. Furthermore, we use a well-known attribution method (known as the Integrated Gradients) to interpret the decisions of the model. Results of the integrated gradients are mapped to individual atoms and visualized for interpretability. The final course of this thesis compares our findings with already-known toxicophores described in the literature. Before we get started, let us first cover the theory we will need to best understand the applied analysis.

1.1. Mutagenicity

An essential step in drug approval is the omission of mutagen molecules. Mutagenicity is the compound's ability to induce DNA mutation, leading to either some deletions or adducts in the DNA. Some DNA repairing mechanisms get distorted because mutagenic compounds intercalate between the double stranded helix [4]. For instance, aromatic polycyclic substructures are likely to intercalate themselves between the base pairs of the DNA molecule forming stabilizing π bond [4]. This undesired effect will hinder DNA repair and replication mechanisms and will result in erroneous base replacements as a consequence. Once a toxic substructure has been identified, we refer to it as a toxicophore. The aromatic nitro and amine groups are well recognized toxicophores for mutagenicity [2]. It is important to note that detecting a toxicophore does not necessarily turn the molecule into a mutagen, but certainly indicates an increased potential for toxicity. With that in mind, no structural properties of non-

mutagens were found that could explain the absence of mutagenicity. I.e. there no detoxifying structural properties that signal 'non-toxicity' [2]. As mentioned earlier, several compounds with a polycyclic aromatic system with large substituents have been reported to intercalate into DNA molecules. Triazene groups were recognized as toxicophores due to their high degree of reactivity after enzymatic epoxidation (DNA replication) [2]. Because Mutagenicity is substructure-relatable, we can accurately detect mutagenicity by applying substructure-search studies. One of the most popular knowledge-based studies is Ames test. In contrast, our approach is computationally-driven (in silico) and aims to learn existing described toxicophores and effortlessly predict the output of the Ames test. More on the substructure search is discussed in **Extended-Connectivity Fingerprints** (ECFP). In this work, we roughly describe mutagenicity (as a wide concept). We do not describe the chemical/structural properties of toxicophores, and therefore we use the terms mutagen and toxic interchangeably, not to introduce any confusion to the reader.

1.2. Ames Test

The identification of mutagenic substances is an important yet a difficult procedure in mutagenicity assessments. The Ames test is an in-vitro mutagenicity assay that aims to detect wide range of chemical substances that can cause reverse mutation leading to detectable genetic damage. The test makes use of several histidine dependent bacteria strains [5]. I.e. the strains are histidine deficient mediums, and they restore their ability to synthesize histidine only then when a bacteria strain is exposed to a mutagen. Each strain carries different mutations in various genes in the histidine operon. These mutations, in turn, act as hot spots for mutagens that cause DNA alteration via various mechanisms. We briefly explored one of these mechanisms in **Mutagenicity**. Molecules that are Ames test positive are referred to as mutagen. The test recognizes a compound as mutagen, if any genetic damage appears in the bacterial assay. For example, a bacteria colony starts to grow at a higher rate than control colonies, when this specific colony is exposed to this chemical. When no mutagens are added to the assay, the number of spontaneously induced revertant colonies per plate is relatively constant, because bacteria do not regain histidine synthesis functionality and hence are not able to form colonies. With that being said, this bacterial assay is associated with high complexity: if one colony starts to grow at a higher rate, it is not immediately indicative of the presence of a mutagen. This is

why this test is associated with a high rate of trial error. Besides that, potential mutagens (ones detected in the assay) have to be further examined in mammalian cells, because a number of these compounds interact with genetic material (e.g. DNA molecule) only if specific enzymes are available during the metabolic activation. Unfortunately, these enzymes exist only in mammalian cells, but not in bacterial ones. As a side note, the assay uses bacteria strains that are highly sensitive to DNA-damaging agents such as Salmonella, Bacillus subtilis or Escherichia coli [6]. The Ames test is one of the heavily used tests in toxicology due to its simplicity, fast applicability and lower costs. Almost every drug discovery process includes the Ames test as an initial detector of potential mutagenicity. In the following section, we demonstrate how the output of the Ames test is reproducible using computational methods such as Neural Networks (NNs).

1.3. Deep Learning in Drug Design

In the literature, several computational methods to predict the Ames test outcome are described and evaluated [7, 8, 9]. However, computational models often suffer from insufficient accuracy, making them unreliable compared to biological experiments [11]. However, machine learning methods such as support vector machine (SVM) has obtained higher accuracies compared to non-learning algorithms when evaluated in a 5-fold cross-validation [9]. Another experiment evaluated several computational and non-computational tools on a benchmark data set - *which is contained in our data set too* - in a 5-fold cross-validation. Results showed that all evaluated machine learning methods (SVM, Gussian Process, Random Forest, k-Nearest Neighbors, and Pipline Pilot) outperformed the non-computational tools [12]. With that being said, the non-computational tools such as DEREK and MultiCASE still have their advantages especially in drug development, since they provide not only structure-activity but mechanistic information too [12]. Both of the previous experiments used the prediction accuracy for performance comparison. The performance of the previously mentioned methods depends on the type of encoded data (e.g. descriptors, atomic graphs), size of the training data, and techniques employed in both data processing and parameter optimization. Moreover, Deep Learning specifically excelled in toxicity prediction and outperformed many other computational approaches like naive Bayes, support vector machines, and random forests [11], because of its ability to detect abstract features in chemical structures. While approaches to

detect structural alerts could be roughly categorized into fragment-based, graph-based, and fingerprint-based approaches [14], all of these solutions had one common issue: interpretability of the model. The National Toxicology Program has determined a 15% inter-laboratory reproducibility error in the Ames test [2]. Therefore, to assist drug approval procedure, computer algorithms such as machine learning are employed. Already existing software tools have their drawbacks in comparison to learning methods: reliance on knowledge database, lengthy matching algorithms, and relatively lower performance. The reason behind that, is that commercial solutions are often set to default parameters, thus full control over the algorithm is not a possibility, let alone adjusting the parameters to the study-case dataset. In contrast to the commercial tools, machine learning algorithms exclusively derive their knowledge from the training data. Here we name a number of limitations that existing tools have when compared to machine learning approaches: poor statistical performance, technical inaccessibility for bench chemists and difficult adaptability to a lab's chemical environment. In light of these facts, future drug development strives for improving the accuracy of machine learning-based methods. All that make chemists desire a better performant and more accessible tool. Additionally, the rapid increase in number of synthesized drugs demanded a more adequate tools for safety assessment than traditional in vitro experiments, where the rate of successfully classifying potential toxic drugs cannot be scaled efficiently and easily. Computation predictions are the result of applying machine learning, artificial intelligence and statistical learning algorithms to simulate the output of chemical experiments - only it is thousand times faster. One could predict the output of thousands of experiments at a time. Data that has already been approved and labeled in labs can now serve these algorithms, by allowing them to learn features in this data. This is achieved by capture the structural information of the molecule in numerical representations and feeding it through several layers of non-linear, differentiable parameterized mappings. Having said that, choosing the right set of parameters is crucial in order for these algorithms to deliver correct predictions. Deep learning, in particular, needs some fine parameter tuning to exploit its predictive power. Finding the best model' architecture is an optimization problem and is necessary to obtain accurate prediction. In the past few years, Deep learning has increased in popularity in the tasks of predicting chemical properties. And when neural networks made their first accurate predictions, it opened the gate for new AI-drug discovery related studies. One of this project's aims is to construct a neural network and best optimize its parameters. Together we look at a binary classification task, where a neural network learns to classify these two classes that we have been discussing (mutagen; non-mutagen). Furthermore, extracting

knowledge from network's decision would not be possible without an accurately predicting network. Therefore, we work on improving the network's predictions by employing the Bayesian optimization compared to a precedent work that used a simple Grid Search. This enabled us to obtain a slight improvement in the AUC score. More on that in the Hyperparameter Optimization section**.** The goal is to then extract knowledge from the model's decisions and relate to the findings of former chemistry literature. One prominent drawback of neural networks is that the functions encoded in the network's layers are often impossible to interpret by humans, and it is therefore impossible for us humans to understand how a neural network arrives at a conclusion. In the following section, we motivate more for this topic.

1.3.1. Interpretability of Deep Neural Network

Deep learning is a highly promising machine learning technique to employ in mutagenicity prediction. However, the problem of being able to correctly predict the outcome of the Ames test does not end here. Without being able to interpret these predictions, we are still far from being able to improve toxicity assessment and extend our knowledge of molecular mutagenicity. Regardless whether in computer vision, toxicity prediction, or natural language processing (NLP), the notion that neural networks are black boxes has been widespread. Having access to the model's 'knowledge' and being able to interpret it is certainly helpful in the development of newly synthesized pharmaceuticals. One could argue about the notion of "neural networks are black boxes" – referring to the fact that we can neither have a look at the network's logic nor debug the network in case of false prediction. In that regard, several

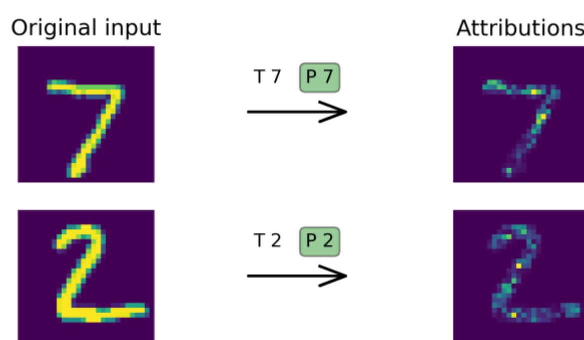


Figure 1: attributions obtained by applying the integrated gradients for two correctly classified samples. The pixels highlighted on the right-hand side are the ones responsible for the model's decision. The higher the intensity, the higher the attribution. All other pixels in the surroundings were given 0 attribution, since they do not contribute to the model's decision at all. 'T' stands for true label, 'P' stands for predicted label.

methods such as DeepLift, layer-wise relevance propagation (LRP) and integrated gradients have been proposed. Attribution methods help understand the model's output from a human's perspective. It is impossible to understand the failures and successes of model predictions (be it a true or a negative prediction). This statement holds true as long as there are no means of interpretation of the model's decision. Luckily for us, we can highlight those patterns in the input that are mostly responsible for the network's decision. The way we do that is by mutating the input and observing the changes in the network's output. Attribution methods exploit this power of NNs. The hint here is that interpretability of predictions can be used to detect the parts of a molecule that are important and analyze its biological properties. To motivate for this, we constructed a toy convolutional neural network to predict the target class of the MNIST data set[1], and applied the integrated gradients to the predicted labels. Results of the integrated gradients are visualized in both Figure 1 and Figure 2. See how pixels where the tint is are mostly informative to the network. Likewise, we want to examine these atoms that are mostly indicative of mutagenicity in our work. In this example we use the analogy of CNNs and the MNIST data. Interpretability of molecular descriptors is analogous too and should not introduce any confusion to the reader[2]. However, molecular descriptors require better understanding on how structural information is captured in the descriptors. Molecular descriptors are examined closely in **Extended-Connectivity Fingerprints (ECFP)**. For Figure 1, one could pose the question "how did the model infer that the image represents a 7 but not a 1?". More interestingly is to understand why the model predicts wrong labels when we expect it to predict the right label, such as in Figure 2. See how the network puts high emphasis on those pixels where the marker's stroke meets the other the other end. For a network, whenever this feature is present, the image looks more like 6 than 5. Thanks to attribution methods, we can explain why the model yielded a different output compared to a true prediction. With that being said, attribution methods do not fully investigate model decisions for every single neuron at every layer in the network. It is an attempt to demonstrate that network's decisions are relatable to the input and

[1] The MNIST data set is a large and curated database often used in machine learning experiments, more specifically image processing. A number of scientific papers rely on the MNIST data to conduct small experiments, since minimizing the generalization error with this data is easy (http://yann.lecun.com/exdb/mnist/).

[2] Here, we take a look at an example from convolutional neural networks (CNN) just for the sake of simplicity. While, we do not employ convolutional layers in our project, we believe that readers who are not familiar with mutagenicity can get a better grasp of model's interpretability with this example.

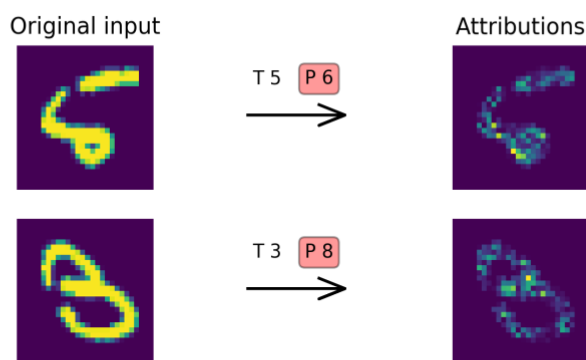


Figure 2: attributions obtained by applying the integrated gradients for two false predictions. See how the attributions correspond to the model's decision. The lower-left stroke in '5' in the first image is tight and makes the digit looks more like '6'. The attributions, in turn, highlight these pixels with high intensity. 'T' stands for true label, 'P' stands for predicted label.

can be explained in a way that matches humans understanding of data. In this work, we take a look at a well-known attribution method known as the integrated gradients [34]. A method that has been revisited by M. Sundararajan et al. to explain network's decisions using gradient calls. We take a close look at this method and demonstrate its potential to attribute neural networks decision in molecular data. Once features attributions haven been worked out, we visualize the learned patterns in order to better understand these learned patterns. With that said, there is no attempt in our work to delve deep into the highlighted representations of network's layers [14]. Instead, we only identify the most relevant patterns in a molecular structure for the prediction of the network. By identifying these patterns, we are able to compare our findings with the literature, allowing for neural networks acceptance in both research and medicine.

Methods

Now that we have gained some understanding of the challenge at hand, we present hereby the methods that were used to solve the stated challenge and conduct our analysis. In general, the capabilities of machine learning algorithms are highly dependent on the type and size of data of interest. As a starter, we give a summary of the dataset at hand. In 2004, J. Kazius et al. [2] constructed a dataset comprising of 4337 molecular structures with corresponding Ames test labels. In 2009, K. Hensen et al. [12] collected a benchmark dataset of 6512 compounds together with their Ames test data from public sources and made this data set of interest available to all researchers to be able to experiment with and compare their computational methods. Our data set is a larger one that has a split from both data sets. Idea is to allow our network to see different types of data samples. The first split yielded a test set consisting of 3315 structures and another set with 4437 structures. The second set was further down split into training and validation set, each with 4010 and 327 structures respectively. The final splitting is reported in the following table:

Table 1: number of mutagens and non-mutagens in each of the training, validation and test set.

	Training	Validation	Test
Mutagen	2220	181	1690
Non-mutagen	1790	146	1625

The reason for us to choose this dataset is because it was highly curated: for example, duplicate structures have already been omitted, since this dataset was collected from multiple sources. It is also put in SMILES format for ease of use [13]. Another reason is that the creators of this dataset declare that the positively labeled structures (mutagens) in this dataset will not change with further testing [12]. Classes distributions of the data set is more or less balanced (see Figure 3). Most interesting of all is the training set that had 430 samples difference. Although oversampling the non-mutagen class would have made a boost in the accuracy of our model, this is not necessary for our analysis, since we use the Area Under the ROC Curve (AUC) as metric to measure the performance of our models. The AUC score is robust to any changes in class distributions and is a valid metric as long as false predictions are equally costly. Let alone

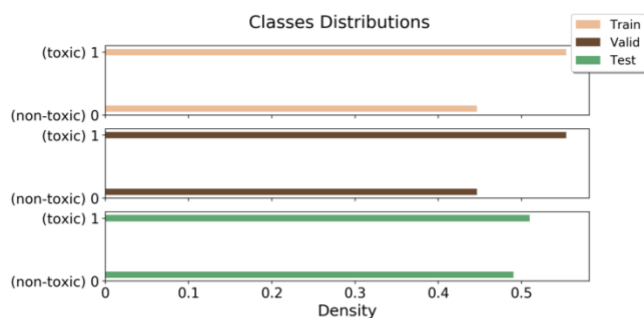


Figure 3: class distributions of all training (top), validation (middle) and test set (bottom). Class imbalance is noticeable but is not significant enough to apply additional data preprocessing techniques such as oversampling the minority class or undersampling. Number of mutagens is always higher than that of non-mutagens in all three sets.

the fact that it is not in our interest to apply these types of modifications to the data set. After splitting, we encoded the data in bits vectors (descriptor-based). For that purpose, we used the Extended-Connectivity Fingerprint (ECFP) [15], where a vector with a desired length is specified, and substructures with high similarity are represented by a bit at the same position in the vector (see Figure 4)[3]. For example, the substructure 'CC1CC(=O)C2CCCCC2C1=O' was found in the first molecule in the test set and was encoded with 1 at position 1014. If this substructure or a similar one was found in other molecules, we indicate its presence with 1 in the row that corresponds to that molecule. If the substructure is not found, we write 0 instead. The result of the ECFP algorithm is a matrix of shape $(n,\ m)$, where n is the number of molecules in the data set, and m is the number of features (the most m frequent substructures in the dataset). The number of features m is controlled by the substructure search parameters. More on that in the upcoming section **Extended-Connectivity Fingerprints (ECFP)**. The choice of the right representation is task dependent. While we could encode the data in a better way, fingerprinting is sufficient because it encodes unique structural representation.

2.1. Extended-Connectivity Fingerprints (ECFP)

There are a number of topological/similarity substructure-search algorithms [15, 16, 17]. We are particularly interested in molecular fingerprinting. Molecular fingerprinting is a

[3] Labels assigned to images are the molecules indices in the test set (molecule supplier). These indices correspond to the $true\ index - 1$ and therefore start at 0, that is because indices start at 0 in Python. However, the reported indices are the human-readable ones and start at 1.

class of methods designed to represent chemical structures with integer arrays. Originally these methods were developed for chemical database substructure searching, similarity searching, virtual screening, clustering and classification tasks [15, 16, 17]. Extended-Connectivity fingerprints, with one special difference to the original Morgan fingerprint algorithm, are designed to capture molecular features that are often associated with molecular activity [15]. It works by assigning (unique) numeric identifiers to each atom in the structure, and iteratively updates these identifiers. After a number of iterations, we obtain disambiguated substructures, that are easily identifiable and comparable. Eventually, the obtained substructures can be represented with bits (0s and 1s) in a bit vector. In order to understand the produced descriptor used in our workflow, and our parameter choice, it is essential to understand how ECFPs work. The algorithm can be broken down to three steps:

1. **Initialization**: in this step, each atom is assigned an integer identifier (e.g. their atomic number, Daylight, atomic invariants-derived rule) - the choice of the initialization method is outside the scope of this work. Hydrogen atoms and bonds to hydrogen atoms are ignored (only heavy atoms are considered). The identifiers can be either positive or negative integers. Here is an example of initial identifiers: [(1: 734603939), (2: 1559650422), (3: 1559650422), …, (6: -1074141656)].
2. **Iterative update**: in every iteration, each atom's identifier is updated to reflect the identifiers of neighboring atoms (i.e. collect substructural information about neighbor atoms). This step might include removal of duplicate structures, especially at early iterations. All identifiers from the previous iteration are collected into a fingerprint set. Next, each atom collects both its own identifier as well as the identifiers of its immediate neighbor atoms into an own array and passes the array to a hash function. A hash function takes care of reducing this array back into a new single integer, which gets assigned as a new identifier to the calling atom. Elements in the array must be ordered before being passed to the hash function - first to their identifier values, and second to their attaching bonds (e.g. single, double or triple bond). Once all atoms have generated their new identifiers, they replace their old identifiers with the new one. Now, the fingerprint set contains those identifiers from the current iteration. In ECFPs, there is no termination condition like in the original Morgan algorithm. Instead, the number of iterations is under the control of the user.

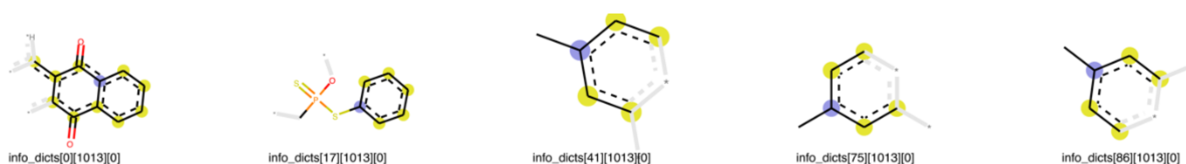


Figure 6: bit at position 1014 can be located in different molecules (e.g. mol. 1, 18, 42, 76, 87). All of these molecules are existent in the test supplier (test set).

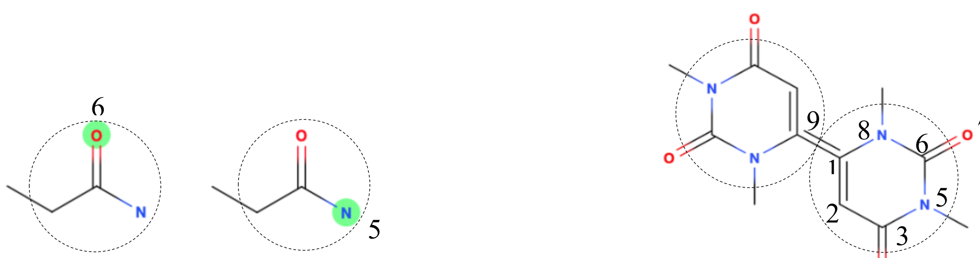


Figure 4: constructed environment after two iterations (radius = 2); once when atom 6 is core, and once when atom 5 is core. Both environments contain exactly the same atoms and bonds. Therefore, they are duplicate information. In comparison to Figure 5, these two core atoms will get different identifiers, and will be represented by the same bit at the same position, but not removed. This is known as *structure duplication*.

Figure 5: for both atoms 1 and 9, the identifiers will be identical even after n number of iterations. Not only do they have same atoms and bonds, but the two regions are mirror images to one another. The terminal step works on removing these duplicates. These two atoms are known as *stereoatoms*. *From a stereochemical point of view, these substructures are not identical.*

3. **Duplicate identifiers removal**: it might occur to anyone that at early iterations, identified substructures will look similar if not identical. Therefore, before moving to the next iteration, this last step takes care of preventing these substructures from appearing in the next iterations, but only then if they correspond to the same set of identifiers. To illustrate that, let us examine Figure 5 and Figure 6. In Figure 5, the algorithm identifies two similar substructures (having the same atoms, same bonds and same activity). These substructures (also referred to as environments) are set two times by two different core atoms: once by atom 6 with radius 2, and once by atom 5 with radius 2. This case is known as structure duplication. In structure duplication, environments get different set of identifiers, and are hence not removed. On the contrary, when the set of identifiers is identical such as the case in Figure 6, they are removed at the third step. This is quite common in symmetrical structures such as the one illustrated in Figure 6.

One variant of this algorithm uses an extra step, where identification of stereochemical fingerprints is possible [15]. However, in our application, we do not make use of this extra step. This process is executed over all atoms in the molecule. As a consequence, the final fingerprint

set is a mixture of substructures of different sizes for each atom in the molecule. In another word, it will contain substructural information from all parts of the molecule. In light of that, what is the appropriate number of iterations? This is very case-dependent. Typically, two to three iterations are sufficient to produce fingerprints used in similarity search or clustering [15]. Since our analysis is an activity-learning one, we choose 3. It is worth mentioning that increasing the number of iterations reduces the number of newly discovered identifiers with every subsequent iteration. Same can be said about the number of bits – what is the right number of bits? Traditionally, users specify the number of bits at ~1024 for most structure-activity learning tasks [15], however, we specified 2048 just so that we allow for less common substructures in the dataset to appear in the descriptor, also because 2048 is the default setting of the RDKit function `GetMorganFingerprintAsBitVect()` [15]. Increasing this number would make the algorithm take slightly longer time to calculate the bits, especially when having a large dataset, but the biggest advantage of ECFPs lies in the fact that they are rapidly computable. While ECFPs have been heavily adopted and updated [15], they have their downsides too: ECFPs identify highly precise structural features. For some purposes, this high level of precision isn't desirable. Instead, some level of abstraction could be more useful. For example, a chlorine or a bromine substituent on a ring may be functionally equivalent but are still distinguished in an ECFP. On the other hand, other variants of the algorithm ignore this detail and treat both of these substituents the same such as in functional-class fingerprints (FCFPs). One last remark in regard to the choice of hash functions: in theory, any hash function that maps arrays of integers to a single integer can be used to generate the new identifier. The only condition for the hash function to be scientifically valid is that it has to take neighboring identifiers into account (i.e. collect neighboring information).

2.2. Feed Forward Neural Networks

A deep neural network is a function that maps an input vector to an output vector. Feed forward neural networks consists of a set of hidden layers, where each layer in the network consists of a number of computing neurons. Neurons in turn are computing units that detect a particular feature using a parameterized function called *activation function*. The activation value h_j^l of neuron j in layer l is the weighted sum of (all) activations from neurons in the previous layer $(l-1)$, and is denoted as

$$h_j^l = a\left(\sum_{i=1}^{I} w_{ij}^l \, h_i^{l-1}\right)$$

where w_{ij}^l is the weight assigned to the connection between neuron i in layer $(l-1)$ and neuron j in layer l, and controls the activation h_i^{l-1} of neuron i in layer $(l-1)$. a is the activation function used in layer l (e.g. ReLU). See **Activation Functions** for better grasp of the transformation that an activation function applies to its input. The weighted sum of every neuron's activation in the previous layer is referred to with the *net input* of neuron j in layer l and is denoted

$$z_j^l = \sum_{i}^{I} w_{ij}^l h_i^{l-1}$$

Furthermore, each layer l transforms its input according to a parametrization to produce its own output, passing it to further layers down the line. Neurons in layer l are understood as feature detectors, since every neuron in the layer applies the same function but receives a different parametrized input from the layer below. What that feature is and how it is detected is all dependent on the input from the previous layer and controlled by the weight matrix

$$W^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1J}^l \\ w_{21}^l & \cdots & \cdots & w_{2J}^l \\ \vdots & \ddots & \ddots & \vdots \\ w_{I1}^l & w_{I2}^l & \cdots & w_{IJ}^l \end{bmatrix}$$

where I is the number of neurons in layer $(l-1)$, and J is the number of neurons in layer l. Neurons within the same layer are not connected to each other. A neural network is a function that maps an input vector x (independent variable) to an output vector y (dependent variable).

$$y = NN(x)$$

Therefore, the input of the input layer is the input vector x itself

$$h^0 = x$$

and the network's output is the activation of the output layer

$$y = h^L$$

where L is the total number of the layers in the network. In regression tasks, the output value is a real number representing the outcome of the dependent variable y, but we do not deal with regression in this work. In a classification task, the output is vector of size matching to the number of possible outcomes k. To make it more specific, let us consider the classification task at hand. There are only two possible outcomes $t_k = \{0,1\}$, where 0 corresponds to the negative class (non-mutagen), and 1 to the positive class (mutagen). We call this case *binary classification*. Data is fed to the input layer (first layer). The information (computation) flows forwards through the network's layers as depicted in Figure 7. The network makes a prediction by finding the maximum value in the output vector y. Every element in the output vector is the output value of one neuron in the output layer and corresponds to one label t_k from all possible outcomes. This value could be understood as the probability that the input x belongs to class k. For example, let us consider the output vector $y = (0.121, \mathbf{0.856}, 0.023)$ in Figure 7. For the network, it is most probable that the passed input x belongs to the second class. Thus, the network classifies the input sample as the second class $y_k = 2$. However, in binary classification such as ours, the output is a single-element vector containing the higher probability. Because probabilities in the output vector sum up to 1 (complementary event), we can easily find the second probability.

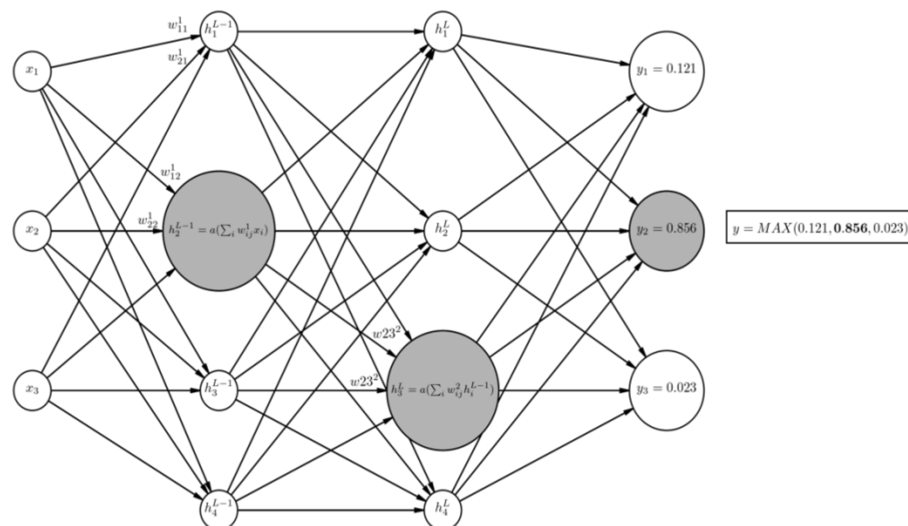


Figure 7: example of a fully-connected feed forward neural network with 2 hidden layers. Input size is 3. Output is size is 3. Information is propagated through the network's layers. Every Neuron in layer l captures specific information (features detector), as the network adjusts its weights at layer l.

2.2.1. Backpropagation

The network "learns" by adjusting the weights W^l layer by layer in order to "fit" the data best possible. To do that, we require an Error Function E that penalizes the network for making false predictions. The error function quantifies the difference between the computed output y_k (e.g. $y_k = 0.856$) and the true value t_k for input x. A very common error function is the mean squared error (MSE) that finds the mean error over a set of N input-output pairs and is defined as

$$E(X) = \frac{1}{2N}\sum_{n=1}^{N}(t_n - y_n)^2$$

where N is the size of input-output pairs, t_n is the true value for input x_n and y_n is the predicted value. $(t_n - y_{nj})^2$ alone is the squared difference of the activation output and the desired output for node j in the output layer L, and is interpreted as the loss for node j in Layer L. In this case, both t_n and y_{nj} are scalars. To find the average loss, we calculate the loss over a data subset $X = [x_1, x_2, .., x_N]$ called a *mini-batch*. When all predictions are accurate, i.e. $y_n = t_n$ for all input-output pairs (x_n, y_n) in X, we get $E(X) = 0$. When we have mismatches, $E(X)$ grows. Therefore, it is desired to have $E(x)$ as close to zero as possible. To do that we adjust the weights in the network such that $E(x)$ is minimized. Let $h_j^L = a^L(z_j^L)$ be the activation of neuron j in the output layer. We write

$$\frac{1}{2N}\sum_{n=1}^{N}(t_n - y_n)^2 = \frac{1}{2N}\sum_{n=1}^{N}(t_n - h_j^L)^2$$

We can minimize $E(x)$ by calculating the gradient of the error function with respect to the weights connecting neuron j with the all neurons in the previous layer. In loose terms, 'how much does the loss change having changed the weights by a little amount'. The gradient of the error function consists of the partial derivatives with respect to every individual weight

$$\nabla E(x)_{w_j^L} = \left(\frac{\partial E(x)}{\partial w_{1j}^L}, \frac{\partial E(x)}{\partial w_{2j}^L}, \ldots, \frac{\partial E(x)}{\partial w_{Ij}^L}\right)$$

The gradient of the error function with respect to one individual weight connecting neuron j in the output layer L and the neuron i in layer $L-1$ can be calculated by

$$\frac{\partial E(X)}{\partial w_{ij}^L} = \frac{\partial}{\partial w_{ij}^L} \frac{1}{2N} \sum_{n=1}^{N} \left(t_n - a\left(\sum_i w_{ij}^L . h_i^{L-1} \right) \right)^2$$

Since E depends on the activation h_j^L and the activation $h_j^L = a(z_j^L)$ depends on the net input z_j^L and z_j^L depends on the weight w_{ij}^L, then the chain rule tells us that to differentiate E w.r.t. w_{ij}^L, we take the product of the derivatives of the composed function

$$\frac{\partial E}{\partial w_{ij}^L} = \frac{\partial E}{\partial h_j^L} \cdot \frac{\partial h_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ij}^L}$$

Let us break down each term from the expression on the right-hand side of the above equation.

The first term is calculated by

$$\begin{aligned} \frac{\partial E}{\partial h_j^L} &= \frac{\partial}{\partial h_j^L} \frac{1}{2N} \sum_{n=1}^{N} (t_n - h_j^L)^2 \\ &= \frac{1}{2N} \sum_{n=1}^{N} \frac{\partial}{\partial h_j^L} (t_n - h_j^L)^2 \\ &= \frac{1}{2N} \sum_{n=1}^{N} 2(t_n - h_j^L) \end{aligned}$$

This means that the loss from the network for N input samples will respond to a small change in the activation output from node j in layer L by an amount equal to the average value of two times the difference of the activation output h_j^L for node j and the desired output t_n.

The second term is calculated by

$$\frac{\partial h_j^L}{\partial z_{ij}^L} = \frac{\partial}{\partial z_j^L} a^L(z_j^L) = a'^L(z_j^L)$$

Because a^L is the activation function employed at the output layer L, the derivative a'^L is dependent on the employed function. We explain activation functions in an upcoming subsection (see Activation Functions). Last but not least the last term is calculated by

$$\frac{\partial z_{ij}^L}{\partial w_{ij}^L} = \frac{\partial}{\partial w_{ij}^L} \sum_{i}^{I} w_{ij}^L \, h_i^{L-1} = I h_i^{L-1}$$

So the input for node j in layer L will respond to a change in the weight w_{ij}^L by an amount equal to the activation output for node i in the previous layer $(L-1)$ times the number of nodes in layer $(L-1)$. Now, combining all terms we obtain

$$\frac{\partial E}{\partial w_{ij}^L} = \frac{\partial E}{\partial h_j^L} \cdot \frac{\partial h_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ij}^L}$$

$$= \left(\frac{1}{2N} \sum_{n=1}^{N} 2\left(t_n - h_j^L\right) \right) \cdot \left(a'^L\left(z_j^L\right) \right) \cdot \left(Ih_i^{L-1} \right)$$

So now we have seen how to calculate the derivative of the loss with respect to one individual weight in the output layer L for N training samples. Now the weight w_{ij}^L is updated using the following rule:

$$w_{ij}^L \leftarrow w_{ij}^L +\cdot \lambda \frac{\partial E(X)}{\partial w_{ij}^L}$$

where λ is called the step size or learning rate, and controls how strongly the weight w_{ij}^L is updated. That is, if λ is too small, the weights converge slowly to the local optimum. If the step size is too high, the gradients will explode causing the weights to diverge. We repeat the same process for each weight in the network. Calculating the derivative of $E(X)$ with the respect to each weight in the network in a backward manner will eventually minimize the error function. This process is known as Backpropagation. This is a typical optimization problem where we have a parameterized function and is solved by minimizing an objective function (loss/error function) by iteratively updating the parameters using gradient descent until a certain condition is met such as finding a stationary point (i.e. local minimum) in the objective. However, what we demonstrated is known as stochastic gradient descent (SGD) or mini-batch SGD. The difference is that gradient descent is applied to the whole training data set at once where SGD is appied to a subset of the training data. Therefore, SGD is computationally more efficient and converges faster towards the minimum of the loss function[20]. For a large data set of size N, in gradient descent we would need to calculate the gradient of the loss for all N samples, and only then can we update the weights. On the other hand, SGD is a more simplified approach that calculates the gradient over an equally sized set of (randomly) chosen training samples, referred to with *mini-batches*. The disadvantage of SGD is that parameter updates are not as precise as in gradient descent [11]. Since the parameters search space contains several local minima, the algorithm is unlikely to find the global minimum, but converges to a local minimum with every epoch. In this example we have discussed one specific error function which is the mean squared error (MSE).

There exists a number of error functions, but most commonly for classification tasks is cross-entropy

$$-\sum_{n=1}^{N} t_n\, log(y_n) + (1 - t_n)\, log\,(1 - y_n)$$

2.2.2. Vanishing Gradient

Problem in deep neural networks is that as we propagate the gradients through the layers, the gradient length decreases exponentially and could become too small for learning in the lower layers (close to the input layer). This problem is known as the *vanishing gradient*. This is one of the main problems that deep neural networks suffer from during training. More specifically, lower layers in the network are more subject to this problem. As noted earlier, once we have calculated the gradient of the loss w.r.t. a particular weight, the gradient is then used to update the weight. If that weight is in the output layer (or any high layer), the calculated gradient is a product of only few terms. However, more and more terms[4] are included in the product, when calculating the gradient at lower layers in the network, allowing the gradient to become extremely small. If the gradient is extremely small, the update in the weight will become extremely small too. This small change in the weight is not going to carry through the network well enough such that the loss is reduced. In another word, the weight is barely updated and does not converge towards the optimal value as a result. Moreover, because weights in earlier layers have implications for the remainder of the network (higher layers), vanishing gradients eventually impair the network's ability to learn. To overcome this problem, we use the Rectified Linear Units activation.

2.2.3. Activation Functions

So far, we have been talking about neuron activations but have not specifically defined what an activation function is. As stated earlier, an activation h_j^l of neuron j in layer l is the output of

[4] The multiplied terms are often small real values (below 1). Therefore, the product gets smaller with more terms.

an activation function $a^l(z_j^l)$ that is employed in layer l. An activation function defines what that output is given a set of inputs by transforming the input's value. Every neuron in layer l utilizes the same activation function to make some transformations on the input but yields a different output due to it receiving a different input. Activation functions are biologically inspired by activity in our brains, where different neurons fire or are activated by different stimuli. Rectified Linear Units (ReLU) is a well known activation function that outputs the same value as the net input (preactivation) if the input is greater than 0, otherwise, it outputs 0

$$ReLU(z) = max\ (0, z)$$

ReLU has the advantage that they are a remedy for the vanishing gradient problem [11]. Sigmoid too is a well known activation function that used in the logistic regression

$$sigmoid(z) = \frac{e^z}{e^z + 1}$$

Sigmoid transforms the input to a value that is close to 0 if the input is extremely negative and to a value close to 1 if the input is extremely positive. See Figure 8 to understand the output of Sigmoid and compare it to other activation functions. The output layer in a neural network often has a special activation function, depending on the possible outcome of the network. For example, in binary classification, it is most suitable to use sigmoid in the output layer, because it outputs a real value between 0 and 1. The closer the value to 1, the higher the probability that the input belongs to class 1, and vice versa for values closer to 0. In multi-class classifications, we need a function that yields probabilities equal to the number of predictable classes such as SoftMax. The basic idea of SoftMax is to distribute the probability of different classes so that they sum up to 1. If we have 5 predictable classes in total, then we need 5 units in the final layer of our network activated by SoftMax. The formula for SoftMax is given by

$$SoftMax(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

Neural networks were historically inspired from perceptrons. The simplest type of feedforward networks are perceptrons. They have no hidden units, thus a perceptron has only an input layer and an output layer. The output units of a perceptron are computed directly from the sum of the product of their weights with the corresponding input units. One prominent advantage of neural networks is their ability to learn high dimensional data. Also, neural networks are complex predictive models that can fit highly complex data, because as we introduce more layers in the

network our predictive model gets more complex due to a higher number of adjustable parameters. This complexity has its drawbacks too. In machine learning, the idea is to build a model that can predict unseen data points. If the model complexity is too high, eventually the model will overfit the learning data, and will not be able to predict unseen data. This concept is known as *Overfitting*. The counter-case is known as *Underfitting*. In underfitting, the model is too simple to fit the training data points. Neither overfitting nor underfitting is discussed in our work, because it requires more background on the *generalization error*. The aim instead is to provide an initial understanding of how neural networks "learn" data. With that being said, the downside to deep neural networks compared to other machine learning algorithms is that NNs have more hyperparameters to tune. One has to test the algorithm under different hyperparameter settings to evaluate the performance of the network. We discuss hyperparameter tuning in the upcoming section. To implement our network's architecture, we use the sequential models from the Keras API [19]. It provides all activation functions, optimizers, loss functions as well as gradient calls that we need for our analysis. In order to have our data ready for training, a molecule has to be described as a vector with input features $x = (x_1, x_2, .., x_m)$, where m is the number of features in the input vector. This has been covered in the previous section Extended-Connectivity Fingerprints (ECFP).

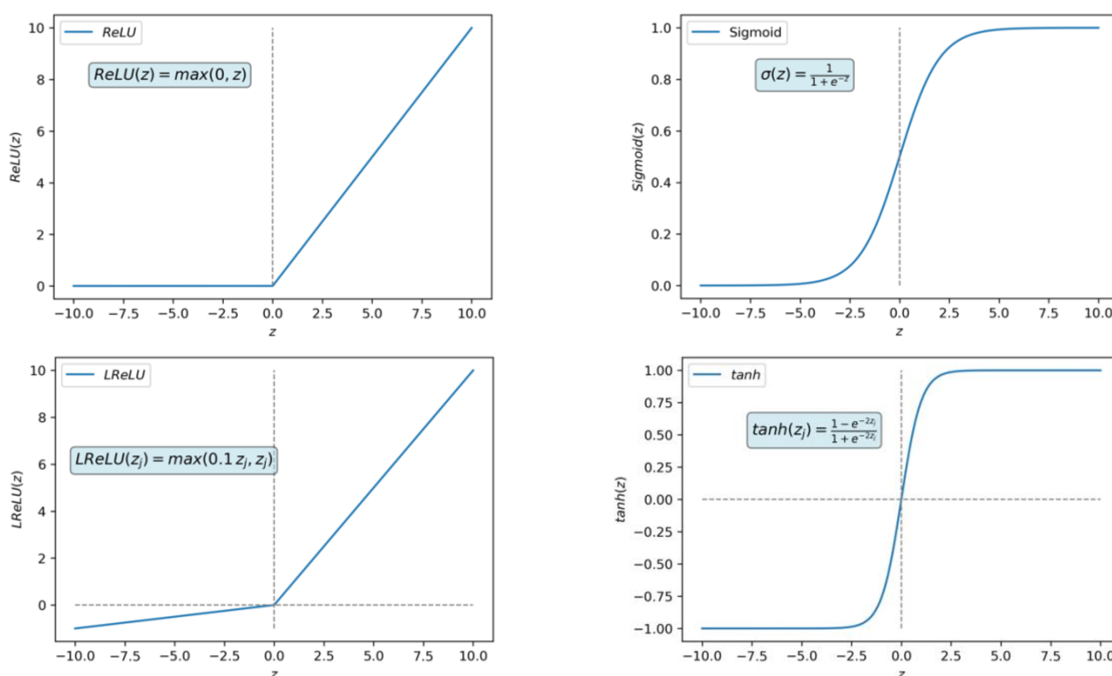


Figure 8: four examples of common activation functions. Rectified linear units (ReLU) on the **top-left**. Sigmoid to the **top-right**. To the **bottom-left** there is leaky ReLU. The **bottom-right** shows the hyperbolic tangent activation function.

2.3. Hyperparameter Optimization

Almost all machine learning algorithms have two different types of parameters: *trainable parameters*; those get updated in the learning process with every epoch (weights of the network), and *hyperparameters*; are usually determined before training and are not updated during learning. The learning rate, choice of optimizer, number of layers in a network are all hyperparameters and are set before training. Choosing the best hyperparameter settings for the learning task at hand is challenging but crucial too for good performance on unseen data. There are a number of model selection techniques to choose from [21, 22, 23]. In a precedent work of ours utilizing the same data set, we used grid search in model selection [24]. The previously selected model yielded an AUC score of ~0.81 on the test. In a grid search, a hyperparameter search space is specified manually, and values from the search space are exhaustively substituted such that we create all possible hyperparameter combinations [21, 22]. This technique suffers from curse of dimensionality [26]. I.e. the search space dimensionality gets n times larger with every parameter setting specified, where n is the number of values to be tested for that particularly specified setting. This time around, we wanted to improve the performance of our model where possible by employing the Bayesian Optimization technique. An automatic tuning technique where a function maps the hyperparameters from the search space to a specified objective. The objective can by anything that we want to maximize or minimize (e.g. accuracy, AUC). The objective is then evaluated on a validation set. In the Bayesian optimization, we differentiate between two types of hyperparameter updates: exploitation, where the selected hyperparameters are expected to get closer to the objective's maximum; and exploration, where using the selected hyperparameters for leads to uncertain outcome. The Bayesian optimization has shown better results in fewer evaluations compared to grid search and random search [22, 25], because it makes informed decisions about the next trial of

Table 2: types of evaluated hyperparameters by the Bayesian optimization as well as their search space. '[]' is used to describe continuous sets, '()' for discrete, and '{}' for categorical ones.

Units (Discrete)	Activation function (Categorical)	**Optimizer** (Categorical)	Learning rate (Continuous)	Num. hidden layers (Discrete)
(5, 1024)	{ReLU, Sigmoid, SeLU}	{SGD, Adam, RMSprop}	[0.00001, 0.2]	(2, 15)
Epochs (Discrete)	Batch size (Discrete)	**Initializer** (Categorical)	**Momentum** (Continuous)	Dropout rate (Continuous)
(5, 30)	(1, 200)	{lecun_uniform, he_uniform, uniform, he_normal, normal}	[0.001, 0.5]	[0.0001, 0.5]

hyperparameter selection. But how do we learn about the next trial? Let $f(x)$ be an objective function, and H a bounded set of hyperparameters. We are interested in finding the global maximum of $f(x)$. To achieve that, the Bayesian optimization builds a probabilistic model for $f(x)$, and exploits this model to make better decisions on which parameters in H to evaluate next [25]. Furthermore, one must also select a prior that will make assumptions about the function being optimized [25, 27]. An acquisition function determines what point in H should be evaluated next and makes use of the information gathered from previous trials [25]. Several acquisition functions are used in the Bayesian optimization, and they vary in the details, in which they suggest the next point in H and previous observations [25, 28]:

- **Probability of Improvement**: maximize the probability of improving over the best current value of the objective function
- **Expected Improvement**: maximize the expected improvement (EI) over the current best
- **GP Upper Confidence Bound (UCB):** minimize the regret over the course the optimization

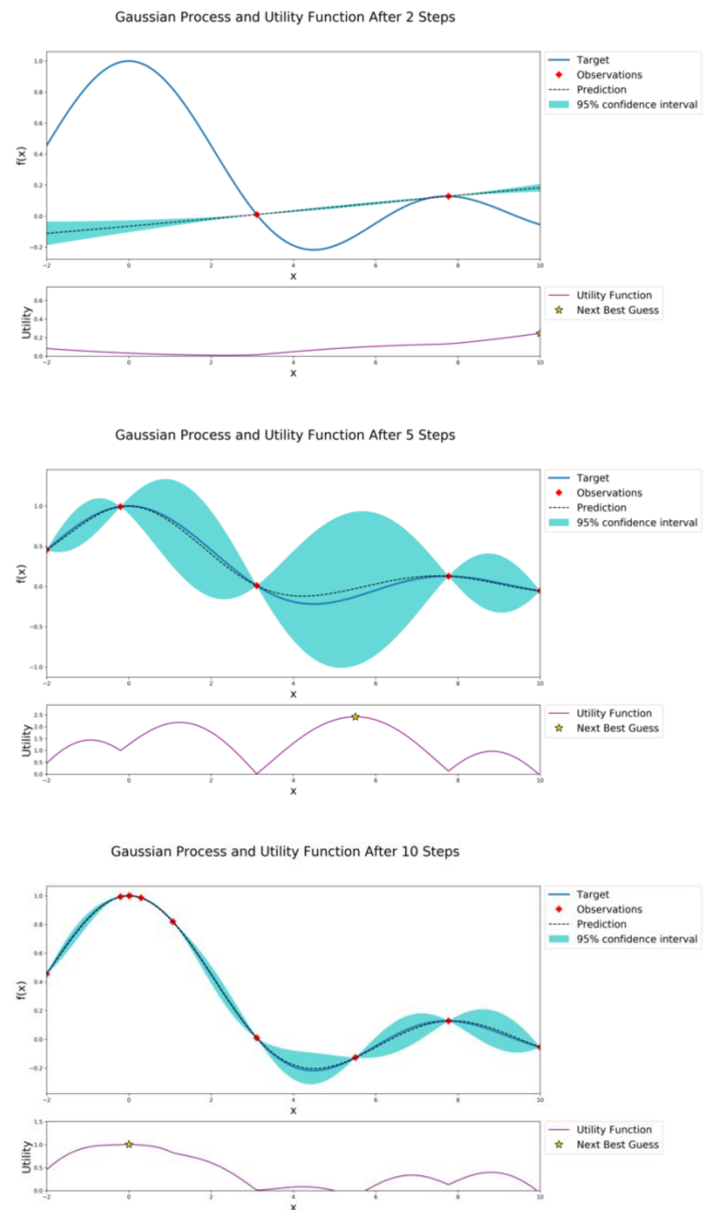


Figure 9: after just a few points, the algorithm constructs a posterior that is close to the true maximum. Notice the difference between exploration and exploitation steps. The former implies exploring the parameter space. The later implies testing points that lie near the current known maximum. The posterior gets closer to the original target with more evaluation points, especially near maxima/minima. At early optimization levels, the algorithm makes fantasies about the shape of the target function (below 3 observations). The confidence interval is reevaluated with every observation introduced. The uncertainty grows as we go further from the evaluated observations.

At early optimization levels, the algorithm is still making some "guess work" about the underlaying distribution – we call *fantasies* [25]. After a number of sampled points, the posterior (dotted line in Figure 9) looks more like the target, especially at maxima and minima. The problem of Bayesian optimization is that it exploits those regions where performance was good in comparison to previous tested points. Thus, the algorithm can easily get trapped in a local maximum, instead of moving towards the global maximum. Furthermore, it is crucial to choose a proper number of exploitation and exploration steps for this algorithm to deliver good results. Exploration, on the other hand, is forcing the algorithm to look for points that have equally or less Expected Improvement (for example) but are far from the currently exploited region in the bounded set H. Another way to get around this trap, is using utility functions that are controlled by an exploration parameter. The upper confidence bound (UCB) for example has a free parameter κ that controls how conservative the utility function is (more vs. less exploration) [28]. Note that these evaluations are often computationally expensive, because they involve running the entire machine learning algorithm until completion. A great advantage of the Bayesian optimization is that it takes significantly less time for evaluation in comparison to other optimization methods (e.g. Grid search) [22, 25], because it makes better choices about where to seek next, and hence less experiments to run. It has outperformed all other state of the art global optimization algorithms [25]. Our search took 4 hours: 59 min: 51 sec when set to 20 exploitation steps and 10 exploration steps. The optimization was run on a 3.1 GHz Dual-Core Intel Core i5. In practice as well, one should be careful about the choice of exploitation steps, because as we exploit some regions in the parameter search space (e.g. learning rate), the algorithm might exploit endlessly precise values, which we cannot represent with a floating point in memory, and the computation would crash as a result. We happened to experience some crashes when setting the exploitation steps too high compared to the exploration steps. To get over this problem we chose the ratio exploitation: exploration steps as 2:1, such that for every two exploitations we take one exploration step. This is in general a good idea to avoid local maxima, because we know that maximizing the AUC is non-convex objective (i.e. has several local and global maxima). To conduct these optimization experiments, we used the BayesianOptimization github package [29]. It provides a constrained global optimization with gaussian processes. In Table 2 we report the type of hyperparameters that we worked to optimize as well as the search space of each. As for acquisition functions, we used the UCB, because across a number of experiments done by Wilson et al., the UCB function outperformed EI on all acquisition tasks, and proved to be more deterministic of exploitation points [25].

Table 3 lists 10 best scoring models in our hyperparameters optimization history sorted by their AUC score. Out of 30 models, we leave out all those models that are obviously poor, maintaining a subset of 10 models for further consideration. This has the benefit of saving computer time and analyst attention too. There is a possibility for any of these listed models to have stochastically scored a high AUC. There are a number of reasons for that: one model might have been 'lucky' to get an easy validation split (i.e. samples in the validation set were similar in structure to the ones already seen in the training set); another reason would be the proper initialization of weights. When the model's weights are initialized before training, they get assigned either randomly or hardly coded for particular reasons. In either case, they are drawn from a prespecified distribution such as a normal or uniform distribution. We refer to these previous cases with 'optimistic' models. Therefore, in order to avoid these optimistic models, we let these exact same ten best scoring models train from scratch and have them predict on the validation set several times (10 times here). Then, we choose the model that scored highest on average with minimum deviation. This is a common practice in machine learning to avoid the trap of optimistic models. With that being sad, there is no way to identify a 'best' set of hyperparameters, we can only hope to find a global maximum in the search space. Our selection

Table 4: results of the Bayesian optimization. Here are the 10 best scoring models found in our search history. Table is sorted to the AUC score in a descending order top-to-bottom. The AUC values correspond to the scores obtained by training a new model every time and evaluating it on the validation set.

Model ID	AUC	activation	batch_size	dropout_rate	epochs	init	lr	momentum	n_layers	optimizer	units
22	0.893741013	relu	200	0.0001	5	normal	0.2	0.5	2	SGD	858
28	0.891016423	relu	200	0.5	30	normal	0.2	0.001	2	SGD	904
24	0.879701809	selu	85	0.5	5	normal	0.2	0.001	2	SGD	586
10	0.874214788	sigmoid	134	0.065985751	23	he_uniform	0.03664644	0.293669954	2	RMSprop	10
12	0.867933096	selu	200	0.5	5	normal	0.2	0.001	2	SGD	103
8	0.867441156	selu	94	0.488382868	20	he_normal	0.007847167	0.142120674	4	Adam	126
25	0.858359192	selu	200	0.0001	5	lecun_uniform	1.00E-05	0.5	15	RMSprop	936
30	0.856750927	selu	1	0.0001	30	lecun_uniform	1.00E-05	0.001	15	RMSprop	373
27	0.841784606	relu	119	0.0001	30	normal	1.00E-05	0.5	2	RMSprop	543
11	0.808332703	selu	1	0.5	30	normal	1.00E-05	0.001	2	SGD	654

Table 3: first 10 best scoring models evaluated 10 times on the validation set in order to avoid optimistic models. This data is a one-to-one comparison and is not sorted. Model 28 is the best scoring model in our experiments, because the selection criterion implies that the selected model has the highest average AUC and lowest standard deviation.

Model ID	1st run	2nd run	3rd run	4th run	5th run	6th run	7th run	8th run	9th run	10th run	Average AUC	Standard Deviation
22	0.881518202	0.8884432	0.870165746	0.888367517	0.882842655	0.888556724	0.876863695	0.878112465	0.858624082	0.868576402	0.878207069	0.009945967
28	0.897979263	0.889237872	0.890259593	0.901574207	0.89417619	0.890656929	0.881877696	0.894119428	0.897392719	0.90089306	0.893816696	0.006025245
24	0.880874896	0.887799894	0.855899493	0.872322712	0.8949141	0.886437599	0.882994021	0.81374404	0.882539923	0.874366155	0.873189283	0.023424586
10	0.874820253	0.858094301	0.8595512	0.856429274	0.854385832	0.848879891	0.856713086	0.854612881	0.848898812	0.859683645	0.857206917	0.007276652
12	0.87724211	0.849012336	0.871338833	0.868879134	0.853174904	0.868879134	0.889843336	0.867176266	0.84685537	0.867592523	0.865999395	0.013180255
8	0.871055022	0.849939454	0.871490199	0.869314312	0.859967456	0.848955574	0.846817528	0.863997578	0.86976841	0.810149096	0.85614546 3	0.018814615
25	0.872814652	0.836259744	0.848293347	0.828085976	0.838681601	0.843260425	0.830583516	0.848974495	0.836713842	0.845190343	0.842885794	0.012638368
30	0.855180504	0.859532279	0.848615	0.829599637	0.847233785	0.841898131	0.852758647	0.825228941	0.853629002	0.822088095	0.843576402	0.013374606
27	0.835238023	0.855558919	0.844925452	0.863089382	0.836297586	0.858510558	0.863732688	0.849428593	0.857186105	0.859948536	0.852391584	0.010479696
11	0.822069174	0.784416862	0.814879286	0.807386665	0.81276016	0.827669719	0.817566033	0.814803603	0.797661394	0.808748959	0.810796186	0.012375706

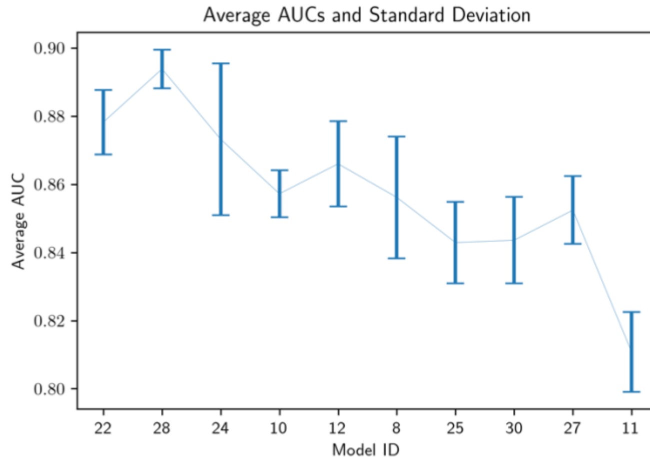


Figure 10: 10 best scoring models after having them initialized, trained (on training set), and evaluated (on validation set) 10 times. The x-axis shows models IDs that match the ones listed in Table 4. The y-axis shows their average AUC scores. The whiskers represents the standrd deviation in the AUC score.

criterion is best presented in Figure 10. In Table 4 we report the AUC scores obtained from each hyperparameter setting in all 10 runs. In the next subsection, we explain the objective used for the evaluation of the models.

2.4. Area Under ROC Curve (AUC_ROC)

In regular learning, we treat all misclassifications equally, which causes issues in imbalanced classification problems. Furthermore, there is no extra reward for accurately predicting a minority class. One could apply cost-sensitive learning to overcome this issue such as in security-related applications and fraud-detection. In this case it is more costly to falsely predict non-fraud than falsely predict fraud – the latter is recoverable, but the former is not. However, we do not always have a ground to assign cost values, and therefore it is rather difficult to construct a cost matrix. Also, the decision between *recall* and *precision* have to made in some machine learning contexts. Recall is the percent of truly positive instances that were classified as such.

$$Recall = \frac{tp}{tp + fn}$$

Precision, on the other hand, is the percent of positive classifications that are truly positive.

$$Precision = \frac{tp}{tp + fp}$$

Depending on whether you lay your focus on the positive or negative class, one metric is more appropriate than the other. For that purpose, we use the AUC score (also known as ROC_AUC), which is the area enclosed under the Receiver Operating Characteristic. While the ROC curve represents the same confusion matrix under different classification threshold settings, the AUC tells how good the performance of the classifier independently of the pronounced threshold setting. Therefore, every point on the ROC curve represents one classifier under one threshold setting. The point (0,1) in the diagram makes a perfect classifier that commits no false predictions. The contrary of that is a classifier whose false positive rate (FPR) is 1 and true positive rate (TPR) is 0 (at point (1,0)), because it constantly makes false predictions. Point (0,0), makes a classifier that classifies all samples in the data set as negative and at (1,1) all samples are classified as positive. Any point on the diagonal is a random classifier whose FPR is equal to its TRP and therefore predicts class p with probability x and class n with probability $(1 - x)$. To see how AUC announces the better classifier independently of the class distributions, let us take a look at the two depicted classifiers in Figure 11. Given two classifiers with the same accuracy (say 85%) such $C1$ and $C2$ in Figure 11, one could pose the question "are these two classifiers equally good in a real application?". For data where correct recognition of negatives is more important, $C2$ would be preferable. For data, where correct recognition of positives is more important, $C1$ would be the better classifier. What if it is

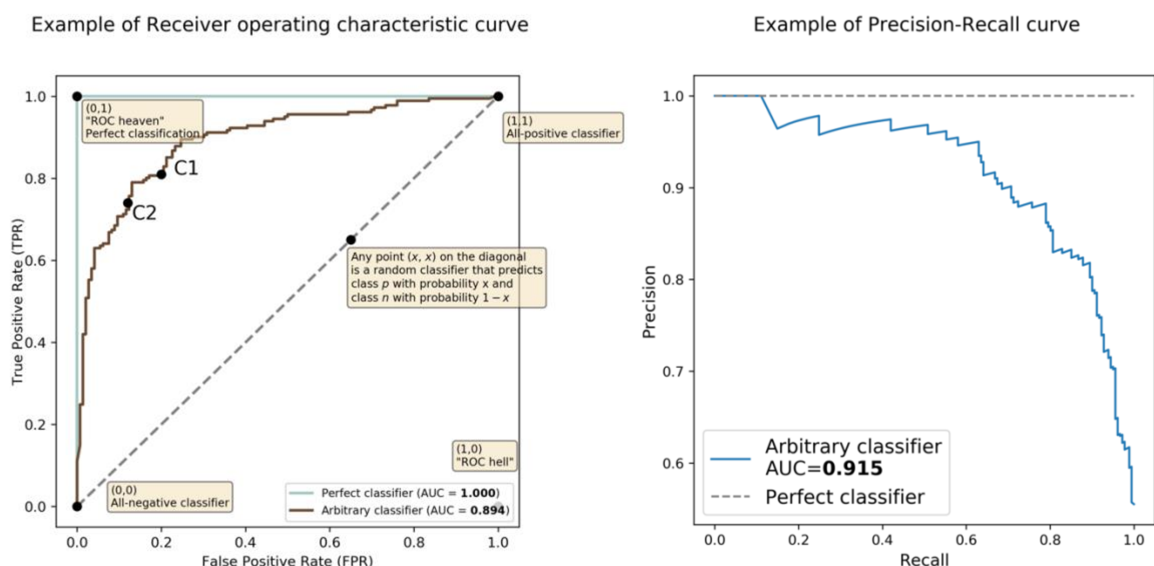


Figure 11: example of a Receiver Operating Characteristic (ROC) curve on the **left**. Example of Precision-Recall plot on the **right**. See how recall and precision are competing objectives. Which metric is preferable depends on the application requirements.

equally important to correctly classify both positives and negatives? – we evaluate the classifiers under different thresholds. To compare the overall performance of two classifiers regardless of the threshold settings, we measure the area enclosed under the ROC curve. The larger the area, the better the classifier. The difference is that AUC allows us to compare two or more classifiers, while ROC alone is not comparable. Another desired property of the AUC score is that it is robust to changes in the class distributions and is therefore a suitable metric for imbalanced dataset with no prediction costs. The value of the AUC ranges between 0.5 and 1.0. An AUC of 0.5 represents a model that does not learn any patterns in the data and commits random predictions. On the other hand, a model that scores 1.0 does not make any false predictions. In practice such a model is unlikely to exist when dealing with real world data. We heavily rely on the AUC in our work. When evaluating models in the optimization process, the AUC represents our objective that we try to maximize. Furthermore, the ROC curve together with the AUC score of our best scoring model are reported in the Results section.

2.5. Attribution Methods

Ever since deep neural networks emerged, they are referred to as black boxes. This notion applies to some but not to entirety. While it is still controversial what pattern in the input a neuron n in layer l recognizes, the question asked today is "why did the network make this prediction?'' rather than "how" if we had to rely on these predictions. A system driven by machine learning that assists the doctor diagnose diabetic Retinopathy is of no use to the doctor unless it can explain its output. And that is why interpretability of neural network is that important. Several methods have been proposed to unravel that mystery of these networks [14]. Attribution methods is one approach towards interpretability of neural networks. We focus on one particular method known as the Integrated Gradients [34] that relates the network's output back to its input by assigning an attribution value (any real number) to each feature in input vector. These values describe the importance of that feature, and thus giving that feature either credit or blame for the model's decision. This of course implies having neutral features too. In another word, features that do not influence the model's decision. Neutrality of features is an important concept too, especially in the integrated gradients (see **Baseline**). As a whole concept, attribution methods are here to help us understand the reasoning of a neural network, as well as give some transparency on the network's final prediction. Not only do attributions play a big

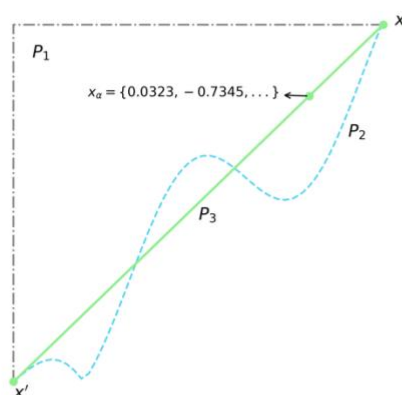


Figure 12: three possible paths drawn between an arbitrary baseline x' and the original input x. The integrated gradients interpolate over a number of samples s that lie on that straight path that connects the two samples (P_3). Examine the interpolated sample x_α for comprehension.

role in understanding the weaknesses of the trained model, but also help boost the performance of the model. Moreover, we could discover new unknown mutagenic patterns in the data. It is particularly interesting to understand the network's decision with the used input in cases where the network makes false predictions. That said, attribution methods are completely independent of the learning process and are not to be considered proof tests that make the model avoid making false predictions. Instead, they are involved in the post-analysis.

2.5.1. Integrated Gradients

As mentioned earlier, attribution methods try to correlate the model's output with the current input to have more insight on the model's decision. Considering a molecule that was classified mutagen, what features in the input can we blame for this decision. To find out, we can interpolate the input over a number of possible inputs increasing in intensity from 0 to 1 (see Figure 12). The closer we get to the original input, the higher the probability that it is mutagen. Calculating the gradients along this path will highlight those non-stagnant features (changing features). On the other hand, the gradient of features that are not changing along the path will be 0. Moreover, the sum of those accumulated gradients highlights will return the attributions highlighting non-stagnant gradients. One question remains unanswered so far -what are we comparing the original input against? Increasing the current input in intensity implies having a reference input referred to with *baseline*. We discuss the role of baselines in the upcoming subsection. By giving an attribution value (importance) to every feature in the input, IG "explains" the decisions of a neural network. To elaborate more on the gradient calculations,

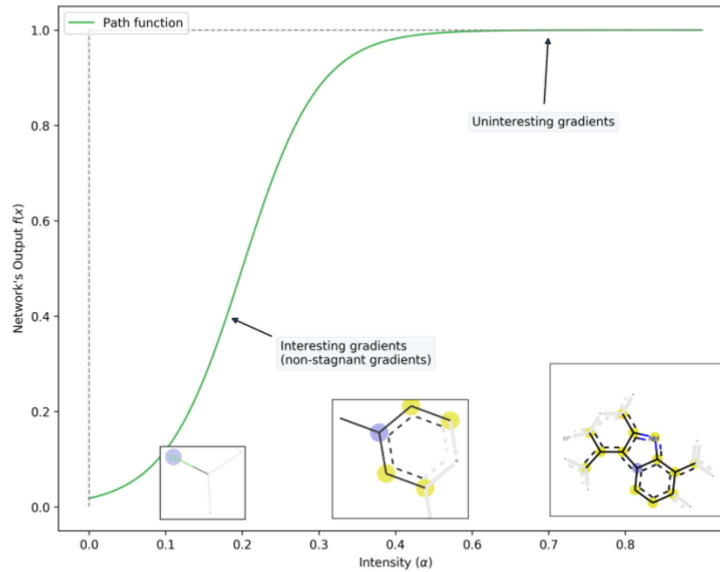


Figure 13: interpolating the input increasing in intensity from 0 to 1. Observe changes in the network's decision

suppose $F: \mathbb{R}^n \to [0,1]$ is a function that represents a neural network that outputs a real value between 0 and 1 (y-axis in Figure 12), given the input $x = (x_1, \dots, x_i) \in \mathbb{R}^n$, we pose the following question: for an arbitrary input vector x, how different is the model's output, having changed feature x_i in the input vector? In another word, we want to understand the changes that occur to the output $F(x)$, having changed, say, one single feature in the input. A common way for humans to perform attribution relies on counterparts' comparisons. This implies the need for a baseline, to which we compare our input. Using a path function we can interpolate the original input over a number of inputs that lie on the straight line between a baseline x' and the original input x (see Figure 12 for simplicity). We define the path function $\gamma: [0,1] \to \mathbb{R}^n$ that can written out in a formal way as $\gamma(\alpha) = x' + \alpha \times (x - x')$, where n is the input dimension. α now controls how close the current sample is to the original input x, such that:

$$\gamma(\alpha = 0) = x'$$
$$\gamma(\alpha = 1) = x$$

Attribution methods based on path integrated gradients are collectively known as path methods. The integrated gradients is such a one. It is important therefore to note that the integrated gradients is a path method for the straight line path specified by $\gamma(\alpha)$. Now that integrated gradients are expressed as the integral of the gradient of the network's output with respect to the input

$$IG(x) ::= \int_{\alpha=0}^{1} \frac{\partial F(\gamma(\alpha))}{\partial \alpha} d\alpha$$

Because the output is a composition of function, we get the following by applying the chain rule

$$\int_{\alpha=0}^{1} \frac{\partial F(\gamma(\alpha))}{\partial \alpha} d\alpha = \int_{\alpha=0}^{1} \frac{\partial F(\gamma(\alpha))}{\partial \gamma(\alpha)} \cdot \frac{\partial \gamma(\alpha)}{\partial \alpha} d\alpha$$

The later gradient is easily computable

$$\frac{\partial \gamma(\alpha)}{\partial \alpha} = \frac{\partial\big(x' + \alpha \times (x - x')\big)}{\partial \alpha} = (x - x')$$

Now the end formula looks like

$$IG(x) ::= (x - x') \int_{\alpha=0}^{1} \frac{\partial F\big(x' + \alpha \times (x - x')\big)}{\partial \gamma(\alpha)} d\alpha$$

The integral is estimated with

$$(x - x') \int_{\alpha=0}^{1} \frac{\partial F\big(x' + \alpha \times (x - x')\big)}{\partial \gamma(\alpha)} d\alpha \approx (x - x') \sum_{k=1}^{s} \frac{\partial F\big(x' + \alpha_k \times (x - x')\big)}{\partial \gamma(\alpha_k)}$$

The number of steps s to estimate the integral is crucial in numerical calculation and is discussed in **Integral Approximation (Number of Summation Steps)**. Here x represents one input sample from the data set that can be interpolated over a number of steps. Results of the integrated gradients is a vector of equal size as the input vector containing attribution values that correspond to the importance of the features. For every feature x_i in the input vector x it assigns a value a_i that represents the blame/credit value. This value is then interpreted as the contribution of x_i to the final output y. IG is an axiomatic approach and satisfies the following four axioms compared to other attribution methods [34]:

- **Completeness:** the sum of attributions in the attribution vector must be equal to the difference between the output at x and at the baseline x'. I.e.

  $$F(x) - F(x') = \sum_{i} a_i \qquad \text{F. 1}$$

  where a_i is the attribution value that corresponds to feature x_i in the input vector x. This axiom is beneficial when computing the gradients numerically, because we can always double-check the correctness of the implementation by comparing the two sides of the equation.

- **Sensitivity:** implies that for two inputs differing in one feature value and same baseline, the attribution value of that feature must be different and non-zero too, if the model yields a new output. We expect the attribution a_i to change, if we mutate feature x_i in the input vector. Sensitivity is implied by completeness. I.e. if all attributions sum up to the difference $F(x) - F(x')$, then we expect that attributions to be sensitive to changes in the input.
- **Implementation Invariance:** two neural networks are functionally equivalent if their outputs are equal for all same inputs, and regardless of the actual implementation. Integrated gradients is an easy-to-implement axiomatic method and requires no modification to the original network. It is completely independent of the concrete choice of model's architecture or trainable parameters, in contrast to other attribution methods such as DeepLift and LPR [34]. This is implied by the fact that they are defined using the underlaying gradients, which do not depend on the implementation. Empirically, no changes were made to apply the integrated gradients to other networks in our experiments.
- **Linearity:** attributions obtained by the integrated gradient preserve any linearity within the network. I.e. a linear combination of two neural networks f_1 and f_2 is $a \times f_1 + b \times f_2$, receives attributions that are equal to the weighted sum of the attribution for f_1 and f_2. Linearly is a property of path methods.

Discussion of these four axioms is out of the scope of this thesis. IG is relatively easy to implement and requires no changes in the original network, which enables us to use the same implementation for a number of networks. Here we provide a summary (recipe) on how to easily implement IG as well as our own implementation of the method:

1. Consider a baseline that when passed to the network, the network predicts the default class when no informative features are present in the input. E.g. black image (each pixel 0) in computer vision.
2. Now, we consider an input sample x and interpolate it over a number of inputs s, between the baseline image and the original image such that the input vector x_α looks closer to the original input vector x as we increase the intensity. When intensity is at maximum, we obtain the original input x, and it is at minimum, we obtain the baseline x'.

Snippet 1: implementation of the integrated gradients that was used in our computations. It requires only specifying the input and output tensors and a few gradient calls. To compute the gradients, we use the function `gradients()` provided by the backend module of Keras.

```
from keras import backend as K
import numpy as np

def integrated_gradients(inp, baseline=None, steps=50):
    # when no baseline is specified, take zero-vector by default
    if baseline is None:
        baseline = np.zeros(inp.shape)
    # gamma calculates the path function for a given alpha
    gamma = lambda alpha: baseline + alpha*(inp - baseline)
    input_tensor = np.array([gamma(alpha) for alpha in np.linspace(0, 1,
    num=steps)])
    output_tensor = model.predict(input_tensor)
    gradient = K.gradients(model.output, model.input)[0]
    sess = K.get_session()
    results = sess.run(gradient, feed_dict={model.output: output_tensor,
    model.input: input_tensor})
    return (inp-baseline) * np.average(results, axis=0)
```

3. For each input on that straight line, we compute the gradient of the network's output w.r.t. the input.

4. We average the computed gradients over the number of interpolated inputs. This is an approximation to calculating the integral. Then, we multiply the result by the difference
$(x - x')$.

We repeat the previous steps for every sample in the data set. To test the performance of our implementation we compared the implementation in Snippet 1 to a GitHub packages[5] that we happened to use once and to an older implementation of ours. The comparison included calculating the attributions of the first 100 samples in the test set 10 times and averaging these values. Each of these runs were timed and run on the same CPU as the hyperparameter optimization. In the Results section we report the average timings. Although our implementation was not the best performant in these experiments, we decided to use our implementation of IG only for demonstration purposes. The easiness of implementing IG has

[5] Link to the Integrated Gradients package by Naozumi Hiranuma. The package was used for comparison purposes. https://github.com/hiranumn/IntegratedGradients

led a number of developers to favor it. Also, integrated gradients is applicable to a variety of areas (e.g. text, vision, medicine and finance).

2.5.1.1. Pros

Here we make a couple advantages that we observed when using IG:

1. IG is an easy-to-implement axiomatic approach to attribute the input and is independent of the implemented architecture, as long as we the gradient of the out w.r.t the input is computable
2. The completeness axiom allows us to evaluate the implementation of IG
3. Sensitivity is a desired property of the integrated gradients, since it highlights those non-stagnant gradients along the path of changing input
4. Results of the integrated gradients are easily interpretable. In our analysis, results of the integrated gradients need extra processing to have them in an interpretable form. We demonstrate this step in the Results section.

2.5.1.2. Cons

Herewith we list some of the main downsides that we happened to experience when using IG:

1. The attribution values obtained by IG are relative, and sometimes hard to interpret alone viewed alone: “How important is a feature with attribution value of -0.00429?”. When compared to another value (e.g. -0.39918), the value is interpreted easier, otherwise, the value alone is meaningful. Because we never project these values alone but always together with the rest of features in the input, this downside is not a stumbling rock.
2. The choice of proper baseline is essential in order for the integrated gradients to work [34, 36]. A common way for humans to perform attribution relies on counter-facts. E.g. the absence of all features in an image is a black image (with zero pixel intensities). This makes the method less desirable, since for some types of data, we

often might not find a proper (meaningful) baseline. For most neural networks, a neutral baseline exists in the input space such as the zero-vector in our case, however, this baseline might be uninterpretable to us humans. We discussed the choice of baseline the following subsection **Baseline**.

3. Features combination is irrelevant to IG. I.e. It doesn't reveal the logic the network uses in combining features. In mutagenicity, some substructures are not mutagenic unless in the presence of a substituent or other substructures. If either of these substructures is present alone, the molecule is not classified as mutagen. While neural networks can build this logic, IG does not reveal this type of features correlation.

2.5.1.3. Baseline

Choosing an adequate baseline is a crucial step in the integrated gradients [34, 36]. While the original paper uses black pixels as the baselines [34], choosing other baselines proved to be equally good in some cases. Selecting a reliable baseline for your input type is not spices you add to the integrated gradients technique. Selecting an inadequate baseline could result in noisy gradients along the path, and could make it look like as if all features are equally responsible. Fortunately for us, features are not equally responsible in our data. Having a reliable baseline on the other hand makes the absence of some features more informative, thus less noisy gradients. Having said that, what is the best way to select a baseline? Depending on the problem you are trying to solve, there is a number of baselines to choose from. It is recommended for both binary and continuous features to use all-zero embedding vectors [34]. Passing the zero-vector baseline to our network resulted in predicting the negative class, because if no features of mutagenicity are present, the mutagenicity signal is low. I.e. the input is less indicative of mutagenicity. In binary classifications, the network learns both types of features: positive ones which increase the model's confidence in predicting the positive class, and negative ones that decrease the model's confidence. Both types of features are informative to the network. In our dataset, we have features that indicate mutagenicity when present, and others that indicate non-mutagenicity (intoxicity) when present. The network learns both types, however we are not aware of the second type of features, because we have a description of only mutagenic features (structures). To elaborate more, a baseline where all mutagenic features are present and non-mutagenic are absent is the most indicative input of mutagenicity. In the same

manner, a baseline where only non-mutagenic features are present is least indicative of mutagenicity. Selecting either of the two baselines is adequate. It is nevertheless hard to construct either of them. Instead, we can construct a baseline that is rather unindicative of mutagenicity just by assuming that the absence of mutagenicity indicators is responsible for predicting non-mutagen. I.e. we consider one of the predictable classes the "default" and that whenever no learned features are present, the input is rather non-mutagen. It is therefore desirable to have our baseline correspond to the "default" class such as the zero-vector. In general, this baseline works almost for all datasets, where the absence of features is indicative of another class label [35]. Other alternatives are there too such as using the average input of several training samples. It is a good practice to try out different baselines when applying the integrated gradients. In our analysis, we test the following four baselines:

1. **Zero-vector**: a vector of size (2048) consisting of only 0s
2. **Average**: at every column i the average of the bits in the descriptor's matrix is calculated, and then rounded it to either 0 or 1
3. **Modal**: at every column i the average of the bits in the descriptor's matrix is calculated
4. **Random**: at every position i 0 or 1 is randomly selected

Choosing a baseline is relevant to the case study at hand and is independent of the network's implementation. Sometimes we cannot assume that finding an adequate baseline for every dataset is manageable, and therefore we state that several baselines are equally adequate for these types of datasets. In light of that, selecting the zero-vector as a baseline is adequate and straightforward in this case.

A common way to visualize the attributions is by scaling the feature values by the attribution values. If the feature's value is 0 this would make the product 0, and the feature is not highlighted as result. Another simpler way is to avoid the multiplication and highlight the attributions alone. We used the second technique with one main difference: the attributions obtained by IG correspond to entire fingerprint. Allowing all atoms in a substructure to be equally attributed is not informative either. It is more desirable to have weighted attributions, where one could see the contribution of individual atoms to the mutagenicity predication. Attributions were weighted in a way such that each atom is weighted by the number of

occurrences (i.e. number of times it appeared in a fingerprint within one molecule). More on that in the Results section.

2.5.1.4. Integral Approximation (Number of Summation Steps)

Computing integrated gradients involves approximating a path integral via a summation. The value of the integral is expected to be equal to the difference $F(x) - F(x')$

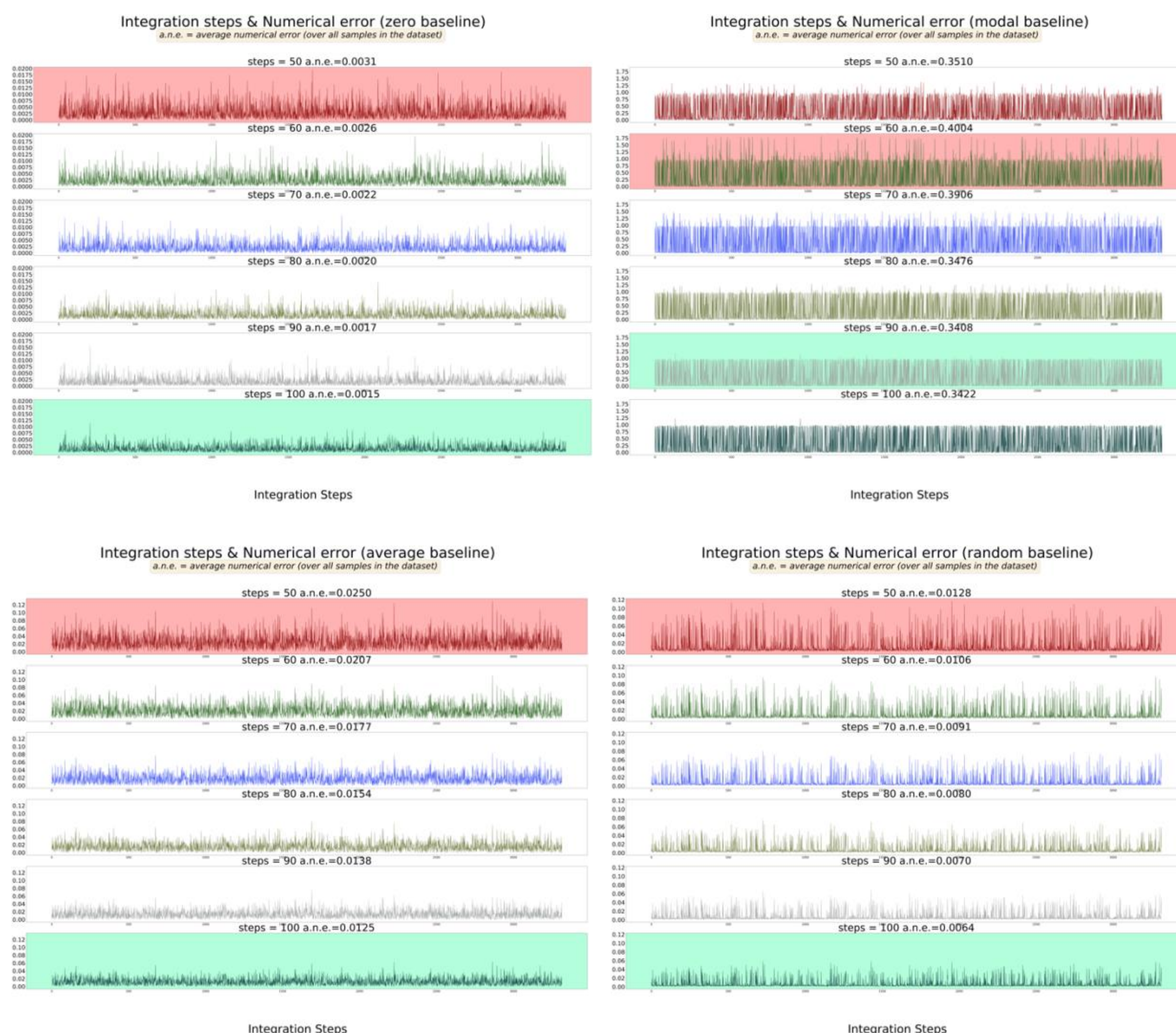


Figure 14: numerical errors found by each baseline vs. the number of integration steps. The x-axis shows the samples in the test data set. The y-axis represents the numerical error found when calculating the attributions at sample x (i.e. $\left|\left(F(x) - F(x')\right) - \sum_{i=1}^{n} a_i\right|$). The average error over all samples is then calculated (a.n.e.), the same operation is repeated with 50, 60, 70, 80, 90 and 100 integration steps and all four baselines (zero-baseline, modal, average, and random). In general, the zero-baseline had the lowest numerical errors with all integration steps. We can also see that the higher the number of integration steps the lower the error is. This statement, however, does not hold true for endlessly higher number of steps for computational reasons.

[F. 1], where $F(x)$ is the output of the network when passing input x. This theoretical property is of advantage for us to check the correctness of our implementation. In practice, however, we still find a slight difference between both sides of the equation. This numerical error is committed with every float number we introduce in the calculation. Should this numerical error be significantly large, we conclude to incorrectness of the implementation (e.g. bug in the code, erroneous differentiation). Should this numerical error be insignificantly small, we can increase the number of integration steps. Theoretically, more summation steps, more precision when calculating the integral according to the fundamental theorem of differentiation and integration. In practice, as we increase the number of steps, we reach a point, where the float points representing the gradient values are highly precise, and thus computer memory cannot accommodate for that high precision level. In the literature, it was recommended to use a value between 50 and 100 [34]. In practice, values in this range should return a 'good' estimate of the integral in the application of the integrated gradients. To illustrate that, we calculate all the numerical errors found using all four baselines (zero-vector, modal, average and random baseline) with integration steps of 50, 60, 70, 80, 90 and 100 for every input sample in the dataset. Then we average the errors corresponding to every sample over the number of samples in the dataset to get better insight on the changes of integral value. This should also give us an idea about which the safest baseline to use computationally. Last but not least, to avoid the randomness in these calculations we repeated the same calculations ten times over several sessions.

Results

When encoding the data, we used the Extended-Connectivity fingerprints to produce bit vectors including either 0s or 1s. The bits indicate the presence or absence of the substructure at position i. Results of ECFP is a matrix of shape (4010, 2048): (327, 2048): (3315, 2048) for training: validation: test set respectively, where every row represents one single molecule and in the dataset. To avoid the trap of optimistic models, we used average validation. The selected model was number 28 and had an average AUC score of ~0.894 and standard deviation of ~0.006 when evaluated 10 times on the validation set. The selected model with 2 hidden layers and 904 rectified linear units per each yielded an AUC score of **0.830** on the test set. This is a slight improvement over our last constructed model from the previous work[6] which had an AUC of ~0.810 on the test set. Figure 15 shows the ROC curves of model 28 after having it

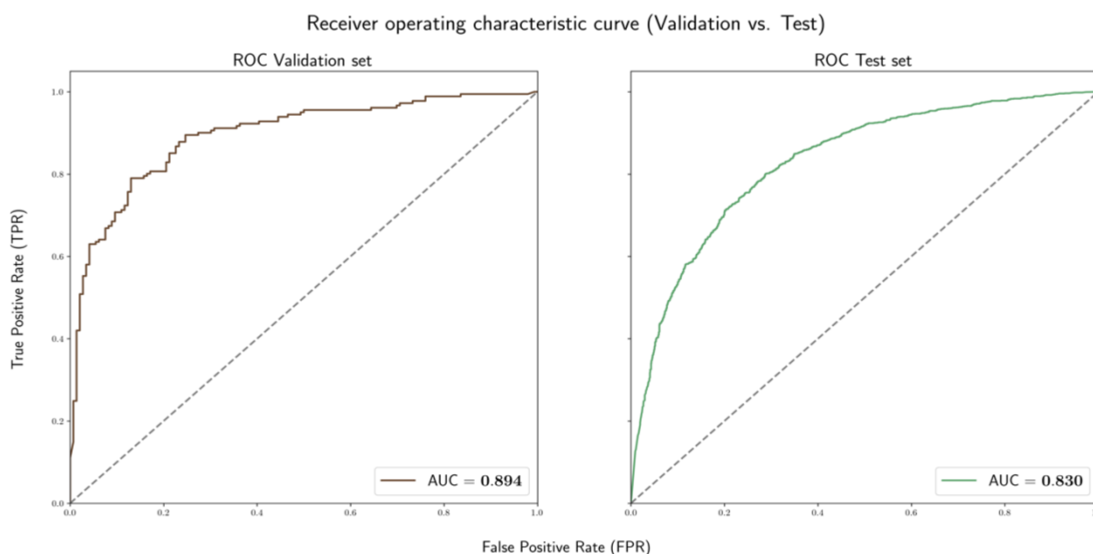


Figure 15: Receiver operating characteristic curves of model 28 (best scoring model). The plot on the left-hand side corresponds to the validation set. The plot to the right corresponds to the test set. The AUC score of each curve is reported on the lower-right corner of the plot.

[6] Link to the previous work on predicting the output of the Ames test: https://github.com/kareemjeiroudi/molecules_and_ml/blob/master/doc/old_(Grid_Search_optimization)/project_report.pdf

Table 5: hyperparameter list of the best scoring network for quick reference.

Model ID	activation	batch_size	dropout_rate	epochs	init	lr	momentum	n_layers	optimizer	units
28	relu	200	0.5	30	normal	0.2	0.001	2	SGD	904

Table 6: matrix containing all average numerical errors (a.n.e.) corresponding to Figure 14, but "zoomed out". Zero-vector baseline had best results for all tested integration steps. Modal baseline on the other hand gave the worst results, which can also be seen in Figure 14. Also, higher integration steps had higher precision and therefore less numerical error.

Num. Steps/ baseline	zero_vector	modal	average	random
50	0.0030856	0.35104253	0.02498267	0.01275446
60	0.00258399	0.40042657	0.02068872	0.01057354
70	0.00222174	0.39062123	0.01771157	0.00905473
80	0.00195577	0.34763111	0.01537029	0.00795998
90	0.0017401	0.34082637	0.01381169	0.00702143
100	0.00152749	0.34223731	0.01246695	0.00636338

reinitialized and trained on the training set. The network was initialized using normal initialization, contains one input layer, 2 hidden layers with ReLU activation, two additional dropout layers with dropout rate of 0.5, and one output layer with 1 single unit activated by Sigmoid. For backpropagation, we used SGD with a learning rate of 0.2 and momentum 0.001 and the binary cross entropy loss function. The network was trained on a batch size of 200 and 30 epochs. It is obvious that there is a huge drop in the AUC score between the validation and test set. One could relate that to the fact that test set has more unseen data samples than the validation set. Possibly, model selection using cross validation such as k-fold cross validation might have given a better estimate of generalization error. In the late analysis, we removed those bits from the test set's descriptor that correspond to fingerprints set by radius 0 (e.g. single atom substituents), because these are redundant structures and usually are not informative. Then we let the network predict the test set, and we surprisingly obtained a much higher AUC score (**~0.894**). We are uncertain why the network had made more accurate predictions with less fingerprints in the descriptors. It could be that these features introduce more noise and less signal to the network. However, this must be tested by training networks with no 0-radius fingerprints. Unfortunately, we left out his part due to time limits. Then we moved to implementing the Integrated gradients. Our new implementation is ~20 minutes faster on average than the old one, but 3 minutes and 15 seconds slower than the prementioned GitHub package. The reason for this improvement over the last implementation is because the gradient call is performed once per sample. In total it took 1 hour and 50 minutes to calculate the attributions for the entire test set on a 3.1 GHz Dual-Core Intel Core i5. To avoid large numerical errors when approximating the integral, we tested four different baselines using 6

different step sizes. Table 6 from our analysis shows the average numerical error that was committed across samples in the test set for every baseline-steps combination. Using the zero baseline produces least amount of numerical error and delivers a better approximation of the path integral. The table shows that the zero-vector had lowest average numerical error for all integration steps. Also, 100 integration steps empirically proved to be a better estimate of the integral for all baselines except the modal one. The modal baseline, on the other hand, always delivered erroneous results four all integration steps. Results of the integrated gradients is a vector of equal size to the input vector (2048). Because we calculated the attributions of the test set only, the final matrix is of shape (3315, 2048). For every tested baseline, we obtained a different matrix. Each attribution a_{ij} in the attribution matrix corresponds to an entire substructure that is found in molecule i and is encoded at column j in the attribution matrix. The fact the attribution value corresponds to an entire substructure, makes it look as if all atoms in a substructure are equally attributable (see Figure 16 **left**). To have the substructure explainable on the atomic level (see Figure 16 **right**), we divided the attribution over the number of atoms existing within one substructure - we call this value "substructural attribution". Then, every atom in the molecule gets an "atomic attribution" that is equal to the sum of all "substructural attributions" only if the atom is present in the substructure. There is no strict way on how to weight the attributions, but this is only one way to arrive at the "atomic attributions". Another suggested way is to average the "substructural attributions" over the number of substructures where the atom is present. Now it is easier to interpret the attributions. In order to best understand the difference between weighted vs. unweighted attributions, let us look at a real-world example that chemists have to deal with: examine the molecule in Figure 16. This molecule is **not** a mutagen but was classified as such by our network. Question here is, what qualifies this molecule to be a mutagen? In loose terms, why does the network "think" it is a

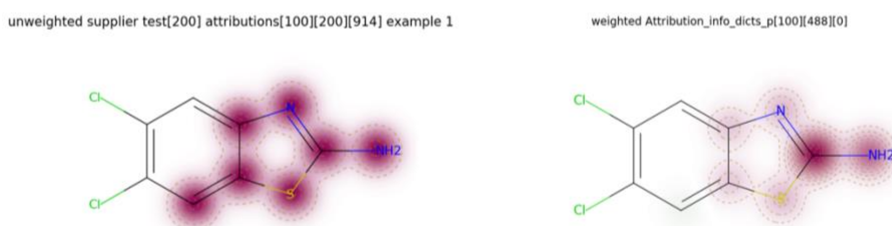


Figure 16: Comparison of unweighted (**left**) vs. weighted attributions (**right**). In the figure left, all atoms are equally attributed. That is rarely the case in a fingerprint. The depicted molecule is number 100 in the test set.

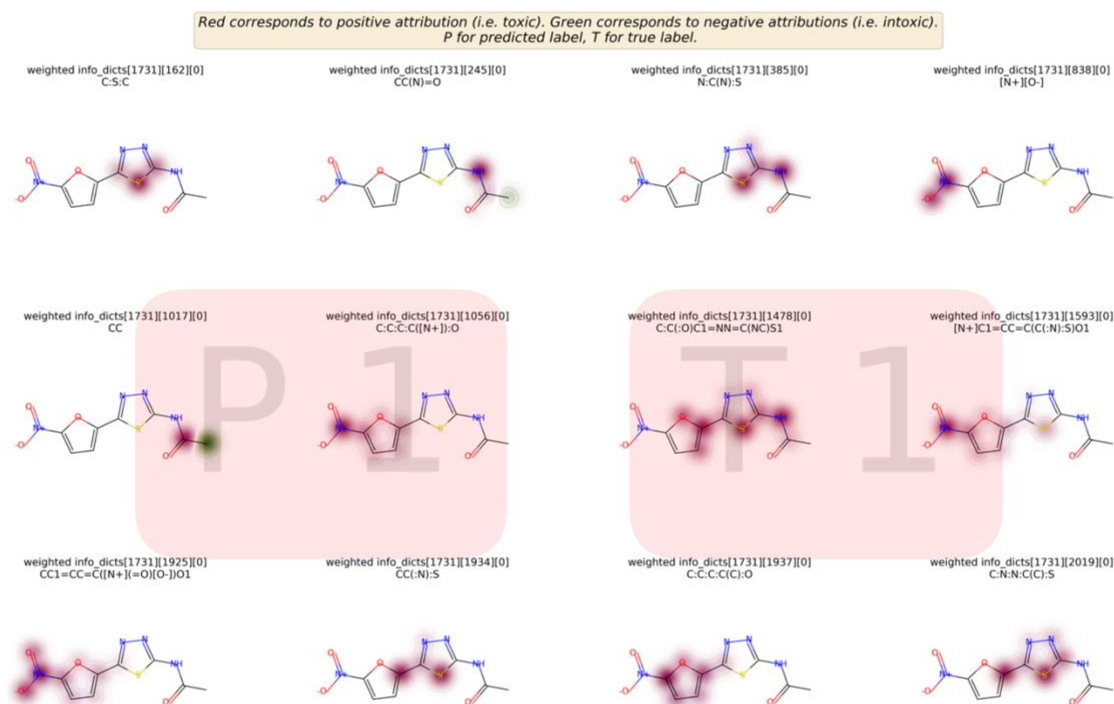


Figure 17: weighted attributions of 12 randomly selected fingerprints found in molecule 1731 in the test set. Positive attributions are highlighted in red. Negative attributions are in green. The intensity of the color corresponds to how big the attribution value is. 'P' stands for predicted label. 'T' for true label. See how most substructures were recognized as mutagenic/toxic by our network, all of a reason for our network to classify the molecule as 'mutagen'. 1 means 'mutagen' (positive class), 0 means 'non-mutagen' (negative class).

mutagen. An attribution vector tells us which features in the bits vector are mostly responsible for this prediction. However, one feature (fingerprint) in the input could cover multiple substructures at the same time. On the other hand, atomic attributions enable us to see the reason behind this prediction on a less abstract level. Viewed this way, if a fingerprint covers two or more substructures in a molecule where only one of the substructures is known to be mutagenic, atomic contributions can translate this information to us. It is certainly of benefit to toxicologists to be able to interpret network's predictions and attribute that directly to the presence of distinct chemical patterns. See how Figure 17 successfully explains the network's decision too, by highlighting a large part of the molecule as 'mutagenic'. For the network, this structure is highly indicative of mutagenicity, just as we would expect since there are two five-membered aromatic nitro structures and one unsubstituted heteroatom ($\mathrm{C}-\mathbf{S}-\mathrm{C}$) present in the molecule. A large part of molecule 1731 in the test set is indicative of mutagenicity. The predicted label matches the Ames test output making it a true positive. Let us examine another example from the test where the network's prediction matches the output of the Ames test such as the one depicted in

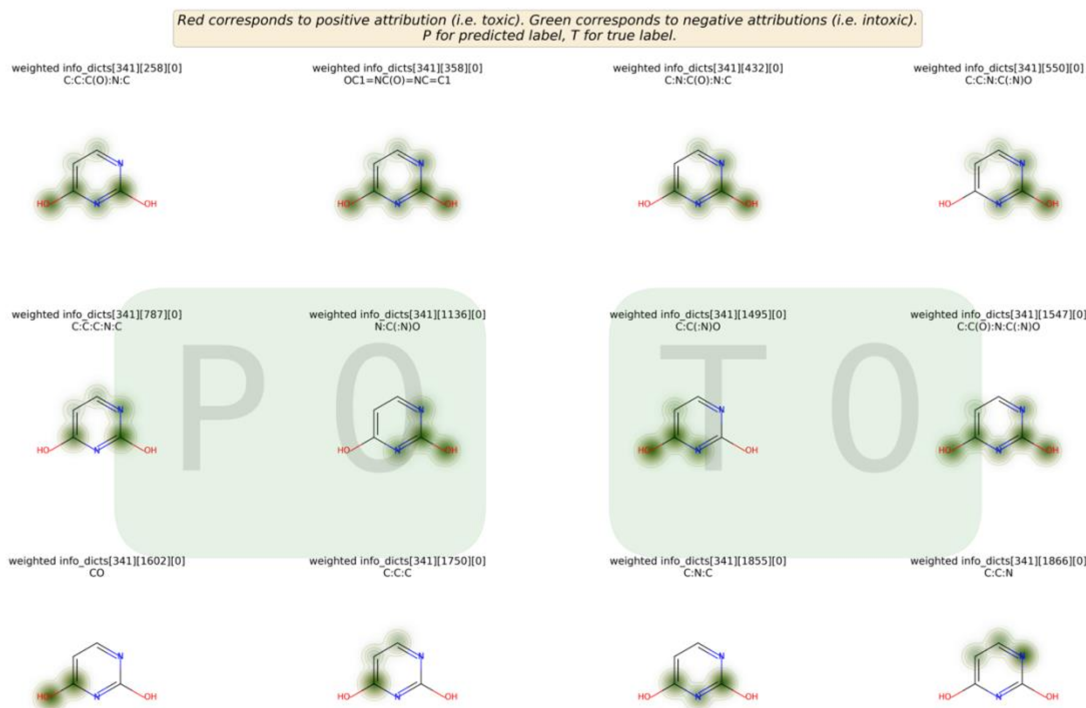


Figure 18: weighted attributions of 12 randomly selected fingerprints found in molecule 341 in the test set. The description is identical to Figure 16. However, in contrary to Figure 16, here a true negative is depicted.

Figure 18. We take a look at a true negative here. We are not able to detect any toxicophores in the molecular structure. Furthermore, the network recognizes both hydroxyl substituents ($-OH$) as 'non-mutagenic'. Again, this is a straightforward decision, and we expect the network's output to match the Ames test true label. It is nevertheless harder to interpret the output in case of a false positive or a false negative, such as in Figure 19. An aromatic nitro is present in the structure, leading the network to 'think' that it is a mutagenic structure. This kind of mismatches require specific knowledge of the functional groups in a molecule in order to be interpreted. We are uncertain why the Ames test of molecule 692 is negative (non-mutagen). Next step was to evaluate the performance of IG on different baselines. To evaluate the performance of the four tested baselines (zero vector, average, modal, random), we constructed Table 7 that allows us to compare the rediscovery rates of each baseline. I.e. to find out if IG is able to detect already known toxicophores and if so, then by how much. We considered only the true positives subset in the data that contains at least one toxicophore in their molecular structure. If we assume that IG is functional, then we expect it to positively attribute those structures where toxicophores are present. Therefore, we want to find the ratio of

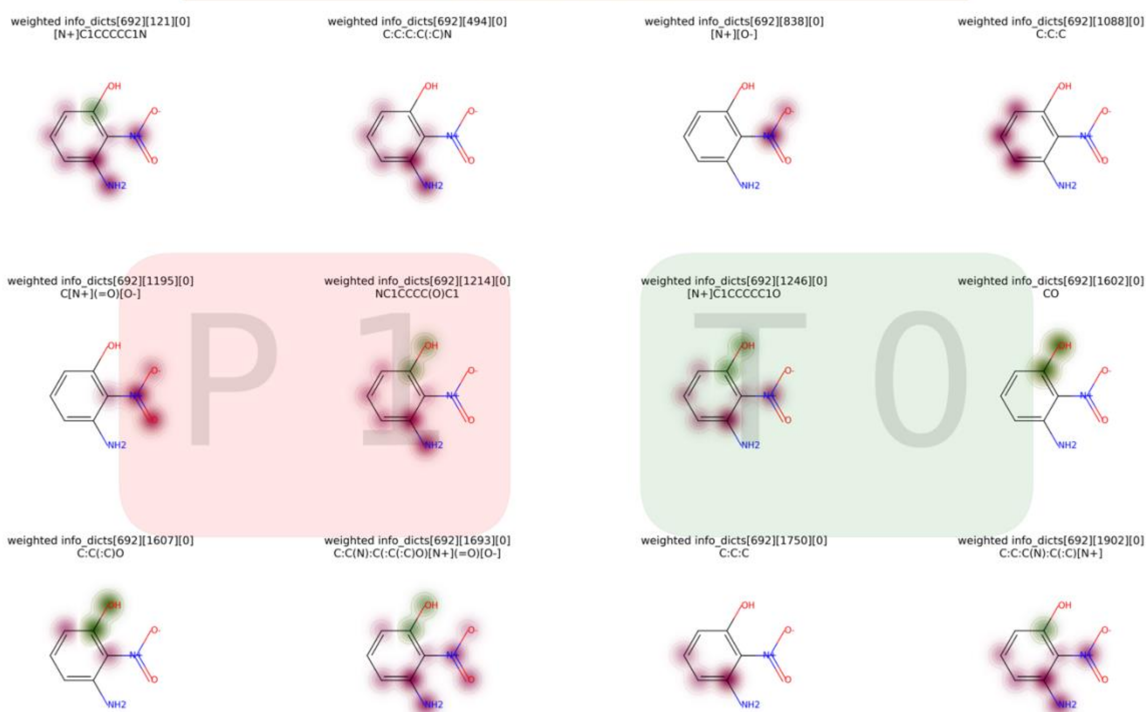


Figure 19: weighted attributions of 12 randomly selected fingerprints found in molecule 692 in the test set. The attributions show that the network understood the molecule as mutagen due to the presence of an aromic nitro. However, the Ames test labeled the molecule as 'non-mutagen'.

$$\frac{\textit{positively attributed structures that include a kown toxicophore}}{\textit{all structures that include a known toxicophore}} = \textbf{\textit{rediscovery rate}}$$

One remark should be made here about the rediscovery rate: having a toxicophore in the molecular structure does not immediately turn the molecule into a mutagen. Therefore, in order to investigate how successful IG is in attributing molecular structures, we would have to find out the number of failures of IG too. The first ratio alone does tell anything about the failures of IG. It is not sufficient to learn that IG is sensitive, but it has to be specific too

$$\frac{\textit{positively attributed strauctures that include no known toxicopohres}}{\textit{all structures that do not include no known toxicophore}}$$

However, the problem is that the second ratio might be a bit more difficult to calculate, since we do not know all potential toxicophores. It is possible that IG recognizes a new toxicophore in a substructure, but only because we are not aware of the toxicophore, the specificity of IG decreases according to the second ratio. Therefore, double checking this decision and further statistical testing is required in order to make a final statement about the performance of IG.

Table 7: rediscovery rates of all four tested baselines. Zero vector had the highest rediscovery rate (**0.634**). The random baseline, not surprisingly, had the worst rediscovery rate.

	Zero vector	Average	Modal	Random
True positives	1374			
TPs with toxicophores	1050			
Num. identifiable toxicophores	1558			
Num. Toxicophores occurrences	8756			
Num. positively attributed toxicophores	5549	5508	5514	2891
Rediscovery Rate	0.63373687	0.62905436	0.62973961	0.3301736

For demonstration purposes, we do not consider all these possible scenarios but only the first ratio so that we can compare the performance of the method using different baselines. In Table 7 the following is reported:

- **True positives** (first row): number of true positives in the test set
- **TPs with toxicophores**: the number of molecules containing at least one toxicophore in their structure.
- **Num. identifiable toxicophores**: unique occurrences of toxicophores in the test data set, regardless of the number of occurrences within one molecule. I.e. if the same toxicophore is detected twice in the structure, it is been identified only once
- **Num. Toxicophores occurrences**: the total number of occurrences of all toxicophores (including duplicates)
- **Num. positively attributed toxicophores** (fifth row): number of positively attributed substructures, where a toxicophore has been detected, using the attribution matrix calculated by each baseline. Correctly attributing a toxicophore multiple times within the same molecule, increases the baseline's rediscovery rate.
- **Rediscovery Rate**: for evaluation of baseline performance

The zero baseline performed the best with a rediscovery rate of ~0.634. In the second place comes the modal baseline. The random baseline on the other hand performed the worst (~0.330). Furthermore, we include a structural comparison of our findings with already existent toxicophores. In Figure 20 we list the structures of some of the most common toxicophores

from the literature [7]. Figure 21 shows a structural comparison of 6 randomly selected toxicophores and 4 different matches found in the test set. See how the network is misled in some cases such as with the aromatic methylamine (fourth example). To investigate the attributions a little more, we wanted to understand the correlation between the attributions and the target label. For every molecule, we calculated the average attribution we calculated the average attribution, where the average attribution is sum of all attributions in the attribution vector over the number of present fingerprints in a molecule (number of bits 1):

$$\tilde{a} = \frac{\sum_i a_i}{m_{bit}}$$

where a_i the attribution at position i in the attribution vector, and m_{bit} is the number of present fingerprints in a molecule. We then obtain a vector of equal length to the size of the data set (3315) containing the average attributions. Every value in the vector describes whether a molecule was on average positively or negatively attributed. Viewed this way, let us understand what relation these values have with the confusion matrix constructed using the network's output. Much like we can see in Figure 17, Figure 18 and Figure 19, we expect the average attribution to cohere with/correspond to the network's output, if integrated gradients is functionally correct. I.e. a true positive showed that the average attribution was correspondingly highly positive too (e.g. 1.7318 for molecule 1731). Furthermore, a lower positive molecular attribution (e.g. 0.0489 for molecule 692) corresponds to either a true or false positive, because the molecule is indicative of toxicity, but the confidence level is lower in this case. Likewise, a true negative received a highly negative "molecular attribution" (e.g. -3.7203 for molecule 341). This correspondence is straightforward but only as long as the molecular structure is big enough such that we can detect several fingerprints in its structure (100 – 200 fps), because we can 'explain' more fingerprints. Also, more fingerprints the better the estimate of the average. It gets harder for the network to extract information if the molecular structure is small and encodes only 2-10 fingerprints. In the latter case, there is too little information to extract from the molecule, and the network is therefore more likely to make a false prediction. It would be particularly interesting to see if some toxicophores are more indicative of toxicity than others,

[7] A tabular representation of the toxicophores and their SMARTS strings is available on the GitHub repository too. See `toxicophores.csv` and `additional_toxicophores.csv` on https://github.com/kareemjeiroudi/molecules_and_ml/tree/master/data.

Table 8: number of molecules where toxicophore i is found vs. number of molecules where toxicophore i is positively attributed vs. number of true positives, where toxicophore i is positively attributed. The reported results are all correspondent to the attributions calculated using the zero-vector baseline. Highest 3 values in a column are highlighted in green. Lowest 3 are highlighted in red. Those toxicophores that were not detected in any molecules are highlighted with a red border.

Tox. Index	Toxicophore Name	Found	TP	TP & Positiv. Attr.	Successful Attribution Rate
0	Specific aromatic nitro	2391	1842	1370	0.7437568
1	Specific aromatic amine	2719	1463	887	0.6062884
2	aromatic nitroso	95	75	57	0.7600000
3	alkyl nitrite	7	7	5	0.7142857
4	nitrosamine	731	602	394	0.6544850
5	epoxide	1258	762	439	0.5761155
6	aziridine	182	160	95	0.5937500
7	azide	249	229	167	0.7292576
8	diazo	5	5	4	0.8000000
9	triazene	0	0	0	
10	aromatic azo	0	0	0	
11	aromatic azoxy	0	0	0	
12	unsubstituted heteroatom-bonded heteroatom	760	420	242	0.5761905
13	hydroperoxide	50	26	19	0.7307692
14	oxime	123	43	25	0.5813953
15	1,2-disubstituted peroxide	158	40	24	0.6000000
16	1,2-disubstituted aliphatic hydrazine	14	12	5	0.4166667
17	aromatic hydroxylamine	329	196	111	0.5663265
18	aliphatic hydroxylamine	581	301	174	0.5780731
19	aromatic hydrazine	66	55	26	0.4727273
20	aliphatic hydrazine	125	89	45	0.5056180
21	diazohydroxyl	2	2	0	0.0000000
22	aliphatic halide	2288	1199	741	0.6180150
23	carboxylic acide halide	12	3	2	0.6666667
24	nitrogen or sulphur mustarg	0	0	0	
25	aliphatic monohalide	1380	661	372	0.5627837
26	α-cholorthioalkane	127	117	74	0.6324786
27	β-halo ethoxy group	0	0	0	
28	chloroalkene	80	57	33	0.5789474
29	1-chloroethyl	42	15	9	0.6000000
30	polyhaloalkene	0	0	0	
31	polyhalocarbonyl	0	0	0	
32	bay-region in Polycyclic Aromatic Hydrocarbons	0	0	0	
33	K-region in Polycyclic Aromatic Hydrocarbons	0	0	0	
35	sulphonate-bonded carbon (alkyl alkane sulphonate or dialkyl sulphate)	0	0	0	
36	aliphatic N-nitro	10	10	5	0.5000000
37	α,β unsaturated aldehyde (including α-carbonyl aldehyde)	211	93	38	0.4086022
38	diazonium	25	25	15	0.6000000
39	β-propiolactone	4	2	2	1.0000000
40	α,β unsaturated alkoxy group	229	214	155	0.7242991
41	1-aryl-2-monoalkyl hydrazine	0	0	0	
42	aromatic methylamine	64	30	18	0.6000000
43	ester derivative of aromatic hydroaxylamine (including original specific toxicophore)	38	15	6	0.4000000

but we could not conclude to a final result due to lack of knowledge of the toxicity index, but there might be a correlation between the toxicity index of a toxicophore and the average attribution.

Lastly, we repeated the same steps as in the rediscovery rate for every toxicophore individually. For every toxicophore we compared the number of molecules containing toxicophore i vs. number of true positives containing toxicophore i. Moreover, the number of true positives where toxicophore i is positively attributed. For that purpose, Table 7 was produced. The table also shows that some toxicophores are more frequent in the test set than others. According to

the literature, the aromatic nitro and aromatic amine are well known toxicophores for mutagenicity [1, 2, 52]. In our analysis, the specific aromatic amine was found in 2719 molecules from the test set making it the most frequent toxicophore in the data set. The specific aromatic nitro is found in 2391 molecules. In the third place comes the aliphatic halide with 2288 molecules. All of the previously mentioned toxicophores were more often than accurately classified (1463, 1842, 1199) respectively. Around two of these molecules, the toxicophores were positively attributed. The successful attribution rate of for these toxicophores does not suggest that toxicophores such as the aromatic nitro and aliphatic halide are more indicative of toxicity than others. The reason for IG to attribute them positively more often, is probably due to their abundance in the data set. On the other hand, some toxicophores such as diazo hydroxyl that is found only two times in the test set was never positively attributed although the network successfully classified both molecules as mutagen. One reason for that could be that the fingerprinting fails to capture some structural patterns [9]. The results of using IG are promising, nevertheless, explaining mutagenicity using IG is not insightful. Because IG successfully attributed only 0.634 of all toxicophore occurrences, it is unlikely that we can derive new knowledge with this low performance. Better evaluation methods and statistical testing is required to come to make a final conclusion about IG.

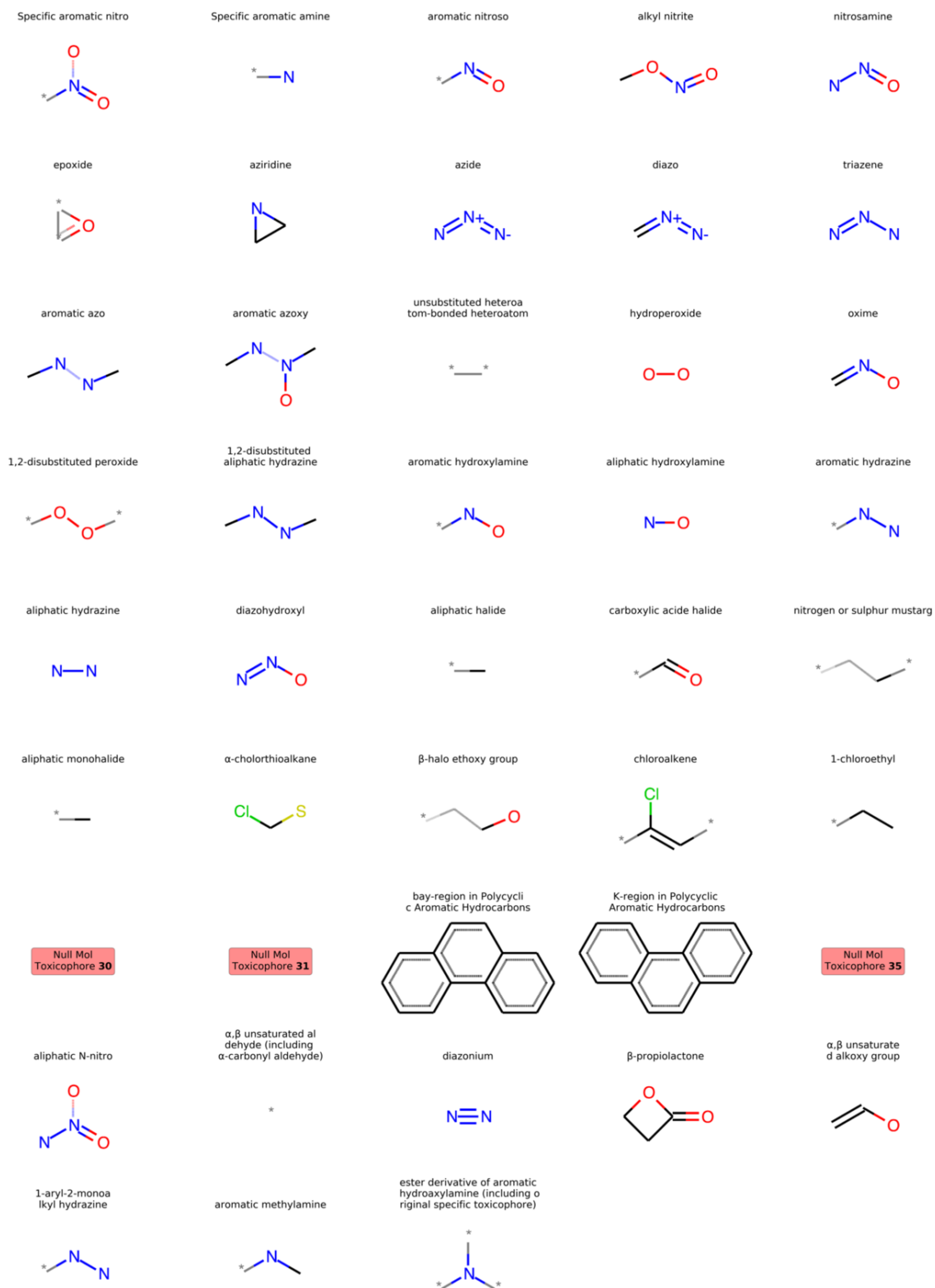


Figure 20: structural representation of some of the most common toxicophores listed in the literature. Some of these could not be plot due to technical errors. The original document provides a toxicity index. Equivalently, our network suggests that some are more indicative of toxicity of others. Comparing that against that toxicity level would have been of knowledge.

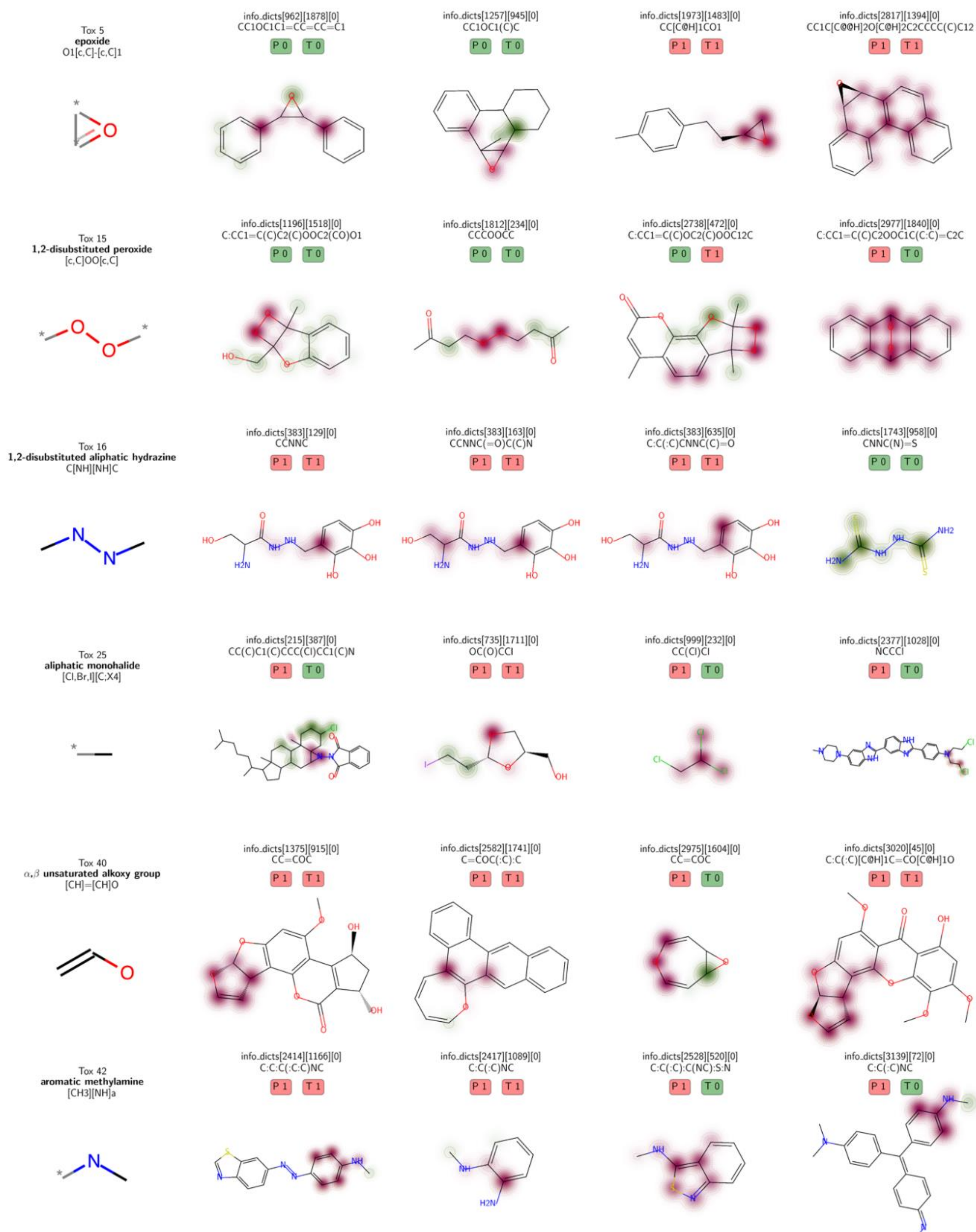


Figure 21: few examples of the toxicophores detected in the test set. On the left side we show the toxicophores formula. On the right side we show 4 examples per toxicophore together with their attributions. Above every example in the plot is a comparison of the true label (T) vs. the predicted label (P). Observe how areas where toxicophores are present are indicating toxicity.

Conclusion

Neural networks are highly accurate computationally-driven predictors in the field of drug design. We have successfully developed and evaluated a number of networks using different hyperparameter settings to obtain a high AUC score on the test. As discussed earlier, boosting the AUC score could be done by a finer hyperparameter tuning or using a different kind of network architectures. We also suggest using a more promising encoding algorithm such graph-based methods, where atom arrangements are encoded in the input too. Additionally, having larger training sets could help neural networks make more accurate predictions on unseen data. Taking the patterns identified by the tested method and the available data set into account makes it challenging to avoid false predictions. Integrated gradients is a great means to explain a neural network's decision. Using the attributions obtained by IG, we have successfully related that network's output to its patterns of interest in the input. As noted earlier, molecules with similar substructures are functionally similar too. In that regard we have highlighted structural alerts in the input data and associated them with their mutagenicity. Looking at these molecules, our attribution method identified a number of the known toxicophores and attributed them positively for being mutagenic compounds. In that regard, we managed to rediscover a number of these toxicophores. However, in order to derive more knowledge from IG, further experimentation is required. On the other hand, IG fails to explain how atoms arrangement is of importance to form a functional group, or how combining multiple functional groups steers a chemical reaction in a different direction, because it attributes features in the input independently of one another. The reason for the network to make a false prediction such as the one demonstrated in Figure 19 cannot be explained using only integrated gradients. One could hypothesize that the mutagenic site of the molecule is inhibited in the presence of some enzymes. And that we can only conclude to that using a bacteria assay such as in the Ames test. These are all some of the weaknesses of IG. Lastly, the choice of parameters is of significance in order for IG to deliver better results - just like any other parameterized algorithm. Also, an assessment index might be necessary to evaluate the significance of a

substructure before comparison. Using an assessment method such as enrichment factor makes it easier to rank the substructures and compare them against the toxicophores [9]. This work has demonstrated the potential of deep learning and attribution methods in Drug design but leaves a lot of room for improvement, all of which is aimed is save chemists lab tedious experiments.

References

1. Martin, Y. C., Kofron, J. L., & Traphagen, L. M. (2002). Do structurally similar molecules have similar biological activity? Journal of Medicinal Chemistry. https://doi.org/10.1021/jm020155c
2. Kazius, J., McGuire, R., & Bursi, R. (2005). Derivation and validation of toxicophores for mutagenicity prediction. Journal of Medicinal Chemistry, 48(1), 312–320. https://doi.org/10.1021/jm040835a
3. Maron, D. M., & Ames, B. N. (1983). Revised methods for the Salmonella mutagenicity test. Mutation Research/Environmental Mutagenesis and Related Subjects. https://doi.org/10.1016/0165-1161(83)90010-9
4. Garrett, R. H., & Grisham, C. M. (1997). Biochemistry, Fifth Edition. Journal of Chemical Education. https://doi.org/10.1021/ed074p189.2
5. Mortelmans, K., & Zeiger, E. (2000). The Ames Salmonella/microsome mutagenicity assay. Mutation Research - Fundamental and Molecular Mechanisms of Mutagenesis, 455(1–2), 29–60. https://doi.org/10.1016/S0027-5107(00)00064-6
6. Hengstler, J. G., & Oesch, F. (2001). Ames Test. In S. Brenner & J. H. Miller (Eds.), Encyclopedia of Genetics (pp. 51–54). https://doi.org/10.1006/rwgn.2001.1543
7. Ridings, J. E., Barratt, M. D., Cary, R., Earnshaw, C. G., Eggington, C. E., Ellis, M. K., ... Yih, T. D. (1996). Computer prediction of possible toxic action from chemical structure: An update on the DEREK system. Toxicology. https://doi.org/10.1016/0300-483X(95)03190-Q
8. Klopman, G. (1992). MULTICASE 1. A Hierarchical Computer Automated Structure Evaluation Program. Quantitative Structure-Activity Relationships. https://doi.org/10.1002/qsar.19920110208
9. Yang, H., Li, J., Wu, Z., Li, W., Liu, G., & Tang, Y. (2017). Evaluation of Different Methods for Identification of Structural Alerts Using Chemical Ames Mutagenicity Data Set as a Benchmark. Chemical Research in Toxicology, 30(6), 1355–1364. https://doi.org/10.1021/acs.chemrestox.7b00083
10. Ivanciuc, O. (2007). Applications of Support Vector Machines in Chemistry. https://doi.org/10.1002/9780470116449.ch6
11. Mayr, A., Klambauer, G., Unterthiner, T., & Hochreiter, S. (2016). DeepTox: Toxicity prediction using deep learning. Frontiers in Environmental Science. https://doi.org/10.3389/fenvs.2015.00080
12. Hansen, K., Mika, S., Schroeter, T., Sutter, A., Laak, A. Ter, Thomas, S. H., ... Müller, K. R. (2009). Benchmark data set for in silico prediction of Ames mutagenicity. Journal of Chemical Information and Modeling, 49(9), 2077–2081. https://doi.org/10.1021/ci900161g
13. Benchmark Data Set for In Silico Prediction of Ames Mutagenicity. (n.d.). Retrieved February 6, 2020, from http://doc.ml.tu-berlin.de/toxbenchmark/index.html
14. Preuer, K., Klambauer, G., Rippmann, F., Hochreiter, S., & Unterthiner, T. (2019). Interpretable Deep Learning in Drug Discovery. https://doi.org/10.1007/978-3-030-28954-6_18
15. Rogers, D., & Hahn, M. (2010). Extended-connectivity fingerprints. Journal of Chemical Information and Modeling. https://doi.org/10.1021/ci100050t
16. Bender, A., Mussa, H. Y., Glen, R. C., & Reiling, S. (2004). Molecular Similarity Searching Using Atom Environments, Information-Based Feature Selection, and a Naïve Bayesian Classifier. *Journal of Chemical Information and Computer Sciences*. https://doi.org/10.1021/ci034207y
17. Glen, R. C., Bender, A., Arnby, C. H., Carlsson, L., Boyer, S., & Smith, J. (2006). Circular fingerprints: Flexible molecular descriptors with applications from physical chemistry to ADME. *IDrugs*.
18. Landrum, G. (n.d.). Getting Started with the RDKit in Python. Retrieved from http://www.rdkit.org/docs/GettingStartedInPython.html#fingerprinting-and-molecular-similarity
19. Chollet, F. et al. (2015). Keras. https://keras.io
20. Werbos, P. J. (1974). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD Thesis, Harvard U. https://doi.org/10.1.1.41.8085

21. Claesen, M., & De Moor, B. (2015). Hyperparameter Search in Machine Learning. 10–14. Retrieved from http://arxiv.org/abs/1502.02127

22. Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011, 1–9

23. Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13, 281–305

24. Aljeiroudi, A. A., & Preuer, K. (2019). Optimizing Keras ANNs for predicting molecules mutagenicity. Retrieved from https://github.com/kareemjeiroudi/molecules_and_ml/blob/master/doc/old_(Grid_Search_optimization)/project_report.pdf

25. Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. Advances in Neural Information Processing Systems, 4, 2951–2959

26. Wikipedia, T. F. E. (2019). Hyperparameter optimization. Retrieved February 10, 2020, from https://en.wikipedia.org/w/index.php?title=Hyperparameter_optimization&oldid=925073211

27. Brochu, E., Cora, V. M., & de Freitas, N. (2010). A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. Retrieved from http://arxiv.org/abs/1012.2599

28. Wilson, J. T., Hutter, F., & Deisenroth, M. P. (2018). Maximizing acquisition functions for Bayesian optimization. Advances in Neural Information Processing Systems, 2018-December (NeurIPS), 9884–9895

29. Nogueira, F. (2014). Bayesian Optimization: Open source constrained global optimization tool for Python Retrieved from https://github.com/fmfn/BayesianOptimization

30. Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. 1–15. Retrieved from http://arxiv.org/abs/1412.6980

31. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Proceedings of the IEEE International Conference on Computer Vision. https://doi.org/10.1109/ICCV.2015.123

32. Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010

33. Guvenir, H. A., & Kurtcephe, M. (2013). Ranking instances by maximizing the area under ROC curve. IEEE Transactions on Knowledge and Data Engineering, 25(10), 2356–2366. https://doi.org/10.1109/TKDE.2012.214

34. Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic attribution for deep networks. 34th International Conference on Machine Learning, ICML 2017, 7, 5109–5118

35. Mudrakarta, P. K., Taly, A., Sundararajan, M., & Dhamdhere, K. (2018). Did the model understand the question? ACL 2018 - 56th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers), 1, 1896–1906. https://doi.org/10.18653/v1/p18-1176

36. Kazemi, V., & Elqursh, A. (2017). Show, Ask, Attend, and Answer: A Strong Baseline For Visual Question Answering. Retrieved from http://arxiv.org/abs/1704.03162

37. Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). Self-normalizing neural networks. Advances in Neural Information Processing Systems

38. Cuissart, B., Poezevara, G., Crémilleux, B., Lepailleur, A., & Bureau, R. (2016). Emerging patterns as structural alerts for computational toxicology. In Contrast Data Mining: Concepts, Algorithms, and Applications. https://doi.org/10.1201/b12986-25

39. Lavecchia, A. (2015). Machine-learning approaches in drug discovery: Methods and applications. Drug Discovery Today. https://doi.org/10.1016/j.drudis.2014.10.012

40. Ancona, M., Ceolini, E., Öztireli, C., & Gross, M. (2017). Towards better understanding of gradient-based attribution methods for Deep Neural Networks. 1–16. Retrieved from http://arxiv.org/abs/1711.06104

41. Unterthiner, T., Ceulemans, H., & Steijaert, M. (2014). Multi-task deep networks for drug target prediction. Advances in Neural Information Processing Systems

42. Li, X., Chen, L., Cheng, F., Wu, Z., Bian, H., Xu, C., ... Tang, Y. (2014). In silico prediction of chemical acute oral toxicity using multi-classification methods. Journal of Chemical Information and Modeling. https://doi.org/10.1021/ci5000467

43. Yang, H., Li, X., Cai, Y., Wang, Q., Li, W., Liu, G., & Tang, Y. (2017). In silico prediction of chemical subcellular localization via multi-classification methods. MedChemComm. https://doi.org/10.1039/c7md00074j

44. Ahlberg, E., Carlsson, L., & Boyer, S. (2014). Computational derivation of structural alerts from large toxicology data sets. Journal of Chemical Information and Modeling. https://doi.org/10.1021/ci500314a

45. Liu, P., Agrafiotis, D. K., & Rassokhin, D. N. (2011). Power keys: A novel class of topological descriptors based on exhaustive subgraph enumeration and their application in substructure searching. Journal of Chemical Information and Modeling. https://doi.org/10.1021/ci200282z

46. Sun, L., Zhang, C., Chen, Y., Li, X., Zhuang, S., Li, W., ... Tang, Y. (2015). In silico prediction of chemical aquatic toxicity with chemical category approaches and substructural alerts. Toxicology Research. https://doi.org/10.1039/c4tx00174e

47. Lepailleur, A., Poezevara, G., & Bureau, R. (2013). Automated detection of structural alerts (chemical fragments) in (eco)toxicology. Computational and Structural Biotechnology Journal. https://doi.org/10.5936/csbj.201302013

48. Floris, M., Raitano, G., Medda, R., & Benfenati, E. (2017). Fragment Prioritization on a Large Mutagenicity Dataset. Molecular Informatics. https://doi.org/10.1002/minf.201600133

49. Webb, S. J., Hanser, T., Howlin, B., Krause, P., & Vessey, J. D. (2014). Feature combination networks for the interpretation of statistical machine learning models: Application to Ames mutagenicity. Journal of Cheminformatics. https://doi.org/10.1186/1758-2946-6-8

50. Baehrens, D., Schroeter, T., Harmeling, S., Kawanabe, M., Hansen, K., & Müller, K. R. (2010). How to explain individual classification decisions. Journal of Machine Learning Research, 11, 1803–1831

51. Xu, C., Cheng, F., Chen, L., Du, Z., Li, W., Liu, G., ... Tang, Y. (2012). In silico prediction of chemical ames mutagenicity. Journal of Chemical Information and Modeling. https://doi.org/10.1021/ci300400a

52. Benigni, R., Giuliani, A., Franke, R., & Gruska, A. (2000). Quantitative structure-activity relationships of mutagenic and carcinogenic aromatic amines. Chemical Reviews. https://doi.org/10.1021/cr9901079

53. Taly, A. (2018). How to use Integrated Gradients (IG). Retrieved October 10, 2019, from https://github.com/ankurtaly/Integrated-Gradients/blob/master/howto.md

54. Tseng, G. (2018). Interpretable Neural Networks. Retrieved October 10, 2019, from https://towardsdatascience.com/interpretable-neural-networks-45ac8aa91411

55. Wikipedia contributors. (2019). Machine learning. Retrieved November 13, 2019, from https://en.wikipedia.org/wiki/Machine_learning

56. Feng, J., Lurati, L., Ouyang, H., Robinson, T., Wang, Y., Yuan, S., & Young, S. S. (2003). Predictive Toxicology: Benchmarking Molecular Descriptors and Statistical Methods. Journal of Chemical Information and Computer Sciences. https://doi.org/10.1021/ci034032s