Imperial College London

BEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Bayesian Optimisation on Graphs

Author:
Vladimir Volgin

Supervisor:
Roberto Bondesan

June 23, 2024

Abstract

Searching for new molecules is a task that requires us to solve a difficult black-box function optimization problem. Many studies tackle this problem in different ways like optimizing for a string representation of a molecule or using iterative approaches that approximate the black-box function. However, there are only a few that look at Bayesian optimisation (BO) for solving the issue and in general the area of BO on discrete structures is underexplored. BO has the potential to be very effective as it can operate on graph representations of molecules without the need to approximate cost function, thus, being fast and accurate. This project capitalizes on previous studies about graph BO and combines it with the performance-improving techniques for regular BO to introduce the implementation that pushes the speed of the optimization to new levels, without sacrificing accuracy.

Acknowledgments

First, I want to thank my supervisor Dr. Roberto Bondesan for providing valuable insight into an area that was completely new to me and guiding me to the end. Additionally, thanks to the MSc group that worked on implementing Bayesian optimisation on graphs before (especially Brian Cregan) for navigating me through their code base and giving helpful advice.

Contents

Chapter 1

Introduction

The main problem that inspired this project is formulated like this: how do we find new molecules with specific needed properties? This is a crucial question as understanding an answer to it would allow us for example to create medical drugs to cure diseases. The problem is that it is an extremely challenging question as searching for new valid molecules is not an easy task considering that an estimated range of searches is between 10^{23} and 10^{60}[1]. Moreover, molecular properties are highly complex and hard to separate.

Such a task can be formulated as an optimization problem; we have a measurement of how fitting a molecule is and want to find a molecule that maximizes this measurement. Additionally, we know that this measurement would be a very complicated function that would be costly for us to compute. This is because checking the molecular properties is a complicated task that might require running an expensive experiment or a computer simulation that might take hours to perform. Therefore, one of the key aspects of molecular search is minimizing the number of calls to this measurement function.

This study explores the use of Bayesian Optimization (BO) over graphs for solving the problem of searching for novel molecules. This method potentially allows us to find the desired molecules quickly. BO over graphs is a complex method with many moving parts, so the work has been done in synergy with the MSc group that developed the software that implements the algorithm. The focus of this project has been on improving the program's performance and pushing the speed of BO iterations as high as possible by implementing the GPU acceleration and investigating the options of using scalable kernel approximations to reduce the time complexity of the calculations.

1.1 Related Work

There are plenty of researches that use different approaches and aim to solve the problem of searching for molecules with specific properties. One example is a study[2] that uses iterative graph generation for molecular search. The disadvan-

tage of such an approach is that it requires the cost function to be called often which makes it necessary to approximate it as, otherwise, such calls would need too much computational resources. This can lead to less accurate results in evaluating new molecules. Another research[3] implements a neural network to optimize over latent spaces, that operates directly on string (SMILES) representation of the molecules. The problem with this method and with other methods (e.g. using de novo molecular generation[4]) that optimize over string representations is that for a given string it is computationally costly to find out if it does correspond to a valid molecule or not.

Bayesian optimization on the other hand combines the advantages of having molecules as graphs, thus, not requiring costly computations of checking the validity of the molecule, and minimising the number of calls of the cost function for evaluating molecules. There are a few quite recent papers, that investigate the BO on graphs for molecular search. For instance, the first study[5] suggests using the combination of the graph kernel and a manually created kernel (more on kernels in the preliminaries section) that focuses on specific graph features with the weights of these kernels being the hyperparameters of the BO model. Another research[6] introduces the library that allows the use of graph kernels for graph BO. However, these studies do not consider the performance side of the methods, conducting the experiments with an extremely small number of iterations. But, performance in this area is very crucial, if we want to search through as many molecules as possible to have the potential to find the most fitting one. The problem is that graph kernels for the Gaussian process limit options for utilization of GPU acceleration as well as scalable kernel approximations that can only work with stationary kernels.

In this project, the graph kernels were not used for the Gaussian process itself, instead, the classic stationary kernel was used, paired with a way to translate graphs into tensors, which opened many options for speeding up the BO iterations by many times. This allowed it to scale the BO to a much larger set of points.

Chapter 2

Preliminaries

2.1 Bayesian Optimization

Bayesian optimization is an effective tool for optimizing black-box functions that are costly to evaluate. The problem we are trying to solve is

$$\max_{x \in A} f(x) \tag{2.1}$$

Where A is a set of feasible solutions and f is a function with an unknown structure which does not allow us to optimize it using other methods easily. Moreover, f takes a lot of time to evaluate which limits our ability to query it too often, encouraging us to carefully choose candidate points to be evaluated.

There are two main parts of the BO algorithm. The first one is creating a Bayesian statistical model for modelling the function f from already acquired data. The second is using the acquisition function to choose the next point to evaluate. We evaluate f at this new point, add the result to our data and repeat the process until we run out of the computational capabilities. The result of BO would be the point with the maximum value of f at it.

2.2 Gaussian Process

For modelling our function we will use the Gaussian process (GP) which describes the distribution over functions giving us the distribution of potential values of f at points in A.

Definition 2.2.1. *A Gaussian process is a collection of random variables any finite number of which have a joint Gaussian distribution[7].*

The GP is defined by the mean function $m(x)$ and a positive-definite covariance function (kernel) $k(x, x')$ of a function $f(x)$ like this:

$$m(x) = \mathbb{E}[f(x)]$$

$$k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x')]$$

and so we can write

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')).$$

Without loss of generality, we can take the $m \equiv 0$. The kernel we will discuss later in this section. If we have several inputs X_* from the candidate set $\mathcal{X}$ we can write

$$f_* \sim \mathcal{N}(0, K(X_*, X_*))$$

where f_* is a distribution of outputs and $K(X_*, X_*)$ is a covariance matrix of inputs. Now consider, we have already evaluated training inputs x_i with corresponding outputs f_i with $i \in 1...n$. Then from the prior, we get that the training outputs f and n_* test outputs f_* have a joint distribution:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}(0, \begin{bmatrix} K(X, X) & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix})$$

where $K(X, X_*)$, $K(X_*, X)$ and $K(X_*, X_*)$ are $n \times n_*$, $n_* \times n$ and $n_* \times n_*$ covariance matrices respectively, evaluated at corresponding pairs of points.
To get the posterior distribution we need to condition the prior Gaussian distribution on our observations [7]

$$f_*|X_*, X, f \sim \mathcal{N}(K(X_*, X)K(X, X)^{-1}f, K(X_*, X_*) - K(X_*, X)K(X, X)^{-1}K(X, X_*)).$$

It is important to note that we only looked at the case where our observations are noise-free which means that the observations are exactly the values of the function. Now assume we have a system noise with variance σ^2 and our observations are $y_i \sim \mathcal{N}(f_i, \sigma^2)$ for $i \in 1...n$. Then we have

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}(0, \begin{bmatrix} K(X, X) + \sigma^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix})$$

and posterior distribution therefore would be

$$f_*|X_*, X, f \sim \mathcal{N}(\mu(X_*), \sigma(X_*)),$$

where

$$\begin{aligned} \mu(X_*) &= K(X_*, X)(K(X, X) + \sigma^2 I)^{-1} y \\ \sigma(X_*) &= K(X_*, X_*) - K(X_*, X)(K(X, X) + \sigma^2 I)^{-1} K(X, X_*). \end{aligned}$$

Thus, for a set of potential points (or for one point as well, X_* being 1×1 matrix), we can use the equation above to evaluate their posterior predictive mean and variance.

2.3 Training Gaussian Process

The Gaussian process has a set of hyperparameters θ, which may include for example likelihood noise or the locations of the inducing points (landmarks for predictions) if they are used. The parameters are learned as usually happens with neural networks, by minimizing the marginal log-likelihood function:

$$L(\theta|X,y) = \log|K(X,X)+\sigma^2 I| - y^T(K(X,X)+\sigma^2 I)^{-1}y,$$
$$\frac{dL}{d\theta} = y^T(K(X,X)+\sigma^2 I)^{-1}\frac{d(K(X,X)+\sigma^2 I)}{d\theta}(K(X,X)+\sigma^2 I)^{-1}y+$$
$$\mathrm{Tr}((K(X,X)+\sigma^2 I)^{-1}\frac{d(K(X,X)+\sigma^2 I)}{d\theta}).$$

2.4 Acquisition Function

There are many different acquisition functions but for this project, we will just use the expected improvement (EI) acquisition function, which is a common and consistent way of deriving the next candidate point. The principle behind EI is straightforward, we want to find a point that on average would increase our observed maximum of f by most. We can think of it as if we had only one more observation left and we wanted to get the best solution at this point. So, EI can be written like this

$$\mathrm{EI}_n(x_*) = \max(0, \mu(x_*) - y_{max}),$$

where y_{max} is the maximum observed value. This can be further rewritten using integration by parts[8] as

$$\mathrm{EI}_n(x_*) = (\mu(x_*) - y_{max})\Phi(\frac{\mu(x_*) - y_{max}}{\sigma(x_*)}) + \sigma(x_*)\phi(\frac{\mu(x_*) - y_{max}}{\sigma(x_*)}),$$

where $\Phi(x)$ and $\phi(x)$ are the cumulative distribution function and probability density function of the standard normal distribution, respectively. Then using the EI acquisition function we can select the next point to evaluate

$$x_{n+1} = \mathrm{argmax}_{x_*\in\mathcal{X}}(\mathrm{EI}_n(x_*)).$$

We continue to iteratively choose new candidate points using the equation above until we exceed the computation time.

2.5 Kernel

The important question left is what kernel function to use for our GP. In fact, this is a very challenging question. There is an endless amount of different kernels, including graph kernels as well. We will discuss the chosen solution in more detail in the implementation section.

The kernel itself is a positive-definite function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that measures how similar are the two points from the candidate set. In general, if two points are similar their covariance will be greater, which is based on the assumption that points closer to each other are more likely to have similar values. For example, if we were working with points in $\mathbb{R}^D$ we could use the squared exponential covariance function

$$k(x, x') = a \exp(-||x - x'||^2).$$

Here we can see that if two points are the same, covariance would be equal to 1, else it would decrease if the points were further apart.

2.6 Graph Kernel

The squared exponential kernel shown above works well for continuous spaces, but we will be dealing with molecules represented as graphs which are discrete structures, so we need another way to measure the similarity between them. Fortunately, there is a large variety of graph kernels that operate directly on graphs, for example, a random walk graph kernel that calculated the number of common random walks between two graphs[9]. Another example is the shortest-path kernel that transforms graphs into shortest-path graphs and compares them[10]. The transformation is done by connecting all the vertices of the original that are in the same connectivity component and giving each edge a value equal to the length of the shortest paths between the corresponding vertices of the original.

The more recent and advanced graph kernel is the deep graph kernel[11]. This kernel is inspired by natural language processing and deep learning; it decomposes a graph into substructures, puts them in a list and treats it like a sentence of words. Then this ”sentence” goes through a feed-forward neural network (layers of neurons with one direction of information flow) that maps similar ”words” to the nearby places in the vector space. Such an approach allows the kernel to consider the similarities between the graphs as a whole as well as between the substructures which leads to better accuracy in graph classification.

2.7 Molecule as a Graph

Initially, the molecules on which the BO will be operating are represented using SMILES (Simplified Molecular Input Line Entry System) which is a string of symbols that corresponds to a three-dimensional molecular structure. Then SMILES strings are converted to graphs that will be stored as network objects in the Python program.

We will represent the molecules as graphs G where atoms would become nodes and bonds between atoms would become the edges. Graph G with n nodes would be a tuple (A, E, F), where A is an adjacency matrix $n \times n$ with values $\{0, 1\}$, A_{ij} being 1 if there is an edge between nodes i and j and 0 otherwise. E is an edge tensor $\{0, 1\}^{b \times n \times n}$ where entry $E_{kij} = 1$ if there is an edge between nodes i and j and it is

of type k, the total number of edge types being b. Therefore, $A_{ij} = \sum_{l=1}^{b} E_{lij}$. Finally, $F \in \mathbb{R}^{n \times d}$ is a feature matrix, where each node has d features represented in real numbers.

Chapter 3

Implementation

The implementations of this project are based on an already existing program developed by a group of MSc students as a part of their group project. In the initial state, the software is able to conduct BO on molecules represented as graphs using the graph kernels from grakel python library. The program is complex with many different parts that depend on each other, so one of the challenges is to identify how the BO flow works and what parts should be changed and added in order to implement the GPU-related optimizations and kernel approximation. The final goal would be to compare the results which include the accuracy and speed of the code with GPU optimization and with or without kernel approximation with the results of the initial program.

3.1 GPU Utilization

The first thing to notice is that the BO algorithm requires a lot of matrix computations, so intuitively it should work significantly faster on GPU rather than CPU. However, graph kernels did not allow for the use of CUDA, a platform for fast and parallel GPU computation. The problem was that graph kernels operated on graph objects not on tensors, so to utilize GPU, it was needed to use a different approach.

In order to avoid using graph kernel for covariance matrix it would be necessary to convert the graph objects into tensors and use for example a classical squared exponential covariance function (its also called radial basis function (RBF)). This means that some accuracy might be sacrificed as instead of having a special kernel for graphs we would have a kernel operating on graph embeddings. However, such an approach would lead to many more iterations done in the same time limit which is potentially a much more significant thing. Additionally, the RBF kernel is stationary, so it depends only on the distance between the points and not on the values themselves[12]. This is another advantage of this approach as the use of stationary kernels allows for the later implementation of an advanced kernel approximation.

If we are set on using the RBF kernel on tensors, a way of converting the graphs to tensors is needed. This is possible to do manually, but there is a risk of RBF kernel

not being able to extract some of the correlations from tensor forms as such kernel would have no idea we are working with graphs. Another option that was chosen for this project is to use the deep graph kernel to get the tensor embeddings from graphs. Such an option is easier to implement and it utilises the deep graph kernel's ability to find correlations between substructures using the neural network. There could be an argument against it as creating such embeddings is not easy computationally. However, we do not need to convert every molecular graph we have into a tensor, we only need to convert the initial sample and then each new candidate after they have been chosen by the acquisition function, which should not significantly impact the performance.

After implementing the above it was finally possible to implement GPU utilization using CUDA. It was important to ensure that all the tensors, the model, the loss function, and the acquisition function were on GPU. But then occurred a difficult problem to solve - the GPU was running out of memory. After doing a bit more than a thousand iterations the program was using all 12GB of GPU memory. A combination of many small things made memory usage much more effective. It had to be ensured that there was minimal movement of data between CPU and GPU, there was a minimal amount of calls to the function that created the graph embeddings, and that all used data, models, and optimizers were deleted. This made it possible to do six times more iterations before running out of memory.

3.2 Matrix-Matrix Inference

Running BO on GPU not only increases the performance by itself but also allows for the utilization of gpytorch's (python library) blackbox matrix-matrix (BBMM) Gaussian process inference with GPU acceleration[13]. The equations, discussed in the preliminary section that are by far the most computationally costly are the $(K(X,X)+\sigma^2 I)^{-1}y$ of the predictive distribution, $\log|K(X,X)+\sigma^2 I|$ of the marginal log-likelihood, and $\mathrm{Tr}((K(X,X)+\sigma^2 I)^{-1}\frac{d(K(X,X)+\sigma^2 I)}{d\theta})$ of its derivative. The default method to compute these would be to use the Cholesky decomposition of $(K(X,X)+\sigma^2 I)$ which has a time complexity of $O(n^3)$, so it scales badly to a large number of points. BBMM uses a modification of the conjugate gradients algorithm[14] that allows for precise approximation of the above equations making the use of GPU's ability for parallel computations. Therefore, using BBMM allows us to reduce the time complexity to $O(pK)$, where K is the time complexity to multiply $(K(X,X)+\sigma^2 I)$ by a $n \times t$ matrix (t is data dimension), which is $O(n^2 t)$ for a standard matrix, and p is the number of iterations of the modified conjugate gradient algorithm that are done (if $p = n$ we get the exact result). This means that BBMM works significantly faster than the Cholesky decomposition, especially with the growing number of points. The requirement for space is $O(nt)$.

3.3 Kernel Approximation

Implementing the RBF kernel on tensors also gives an option to use a scalable kernel approximation. Such an approach was tested for regular continuous space Gaussian processes, but not on graphs, so one of the aims of this project is to evaluate it for use in graph Bayesian optimization.

Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)[15] is a framework for kernel approximation for fast computations through kernel interpolation. It places several inducing points on the grid and interpolates to create a kernel approximation in near-linear time, with the error decaying cubically. However, KISS-GP turned out to be extremely ineffective when operating on tensors that represented graphs. The problem was that these tensors were of high dimensionality to accurately store the molecular graphs, which led to the algorithm being slow due to the grid construction scaling exponentially with the dimensionality. This led to the consideration of Scalable Kernel Interpolation for Product Kernels (SKIP)[16].

SKIP broadens KISS-GP to higher dimensional data by abusing specific kernels' product structure, leading to a linear rather than exponential scale of runtime from the dimensionality.

Chapter 4

Experimental Results

In this section, we will investigate and compare the results of running the baseline solution that used a deep graph kernel with the results of running the newly implemented solution that utilizes the RBF kernel with GPU acceleration and the deep graph kernel for creating the graph tensor embeddings. Additionally, we will look into the outcome of using the kernel approximation for the RBF kernel.

4.1 Experimantal Setup

All the experiments were done with the same setup and parameters to fairly reflect the performance differences. There was the same number of 10 initial sample points and refitting of the Gaussian process model was done every 10 iterations. The program was run on the Imperial College London GPU cluster, using Tesla A30 12GB Mig GPU Devices and AMD Epyc CPU. Two different oracle (the black box) functions were used for the experiments. The first one is ”troglitazone rediscovery”, which aims to discover the troglitazone molecule used for diabetes, treatment[17]. The other one is a function searching for thiothixene, the antipsychotic that helps with the treatment of schizophrenia[18].

The experiments utilising the GPU were running until the 12GB of GPU memory ran out, which was approximately 6500 iterations without the scalable kernel approximation and 350 with the approximation. The baseline solution was operating for 1000 iterations as it was becoming much slower after and it would not make sense to spend computational power on it.

4.2 Quantitative Results

Firstly we will look at how fast we can do the Bayesian optimization iterations. The difference in times for iterations between two of the oracle functions turned out to be very insignificant, which is expected due to the BO algorithm itself being the same, therefore, the results presented in this section are only from the ”troglitazone rediscovery” tests. Table 4.1 shows the number of minutes it takes for a particular

implementation to reach the different numbers of iterations.

The implementation	first 100	first 350	first 1000	first 6500
Baseline	6	43	428	-
GPU	8	34	105	747
GPU + Approximation	9	56	-	-

Table 4.1: Time (minutes) for reaching the iteration counts

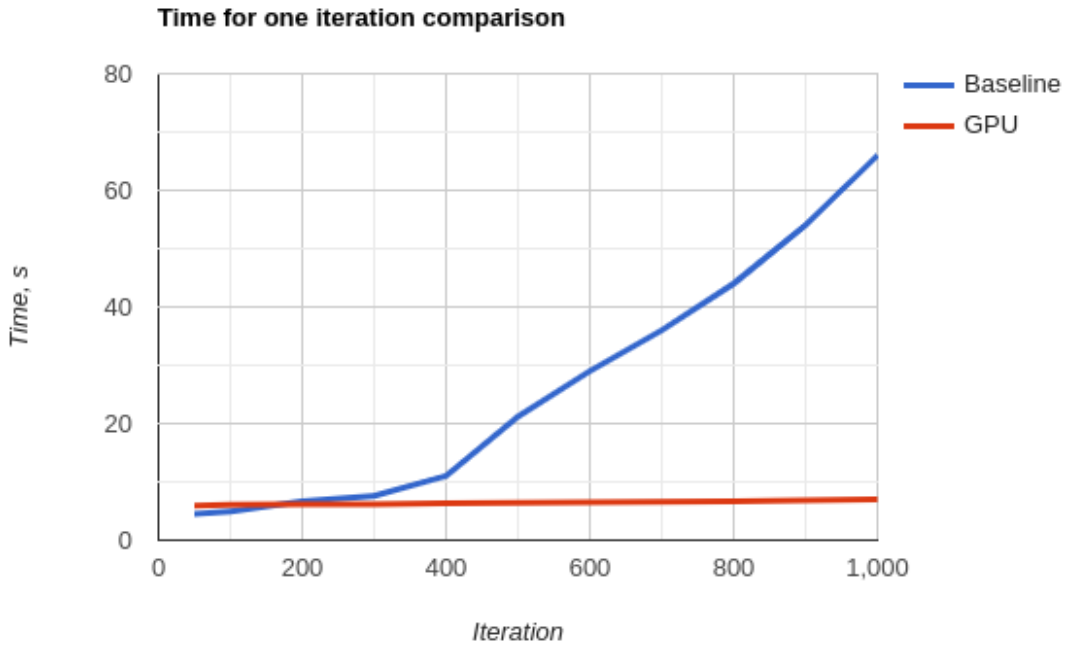


Figure 4.1: Average iteration time scaling

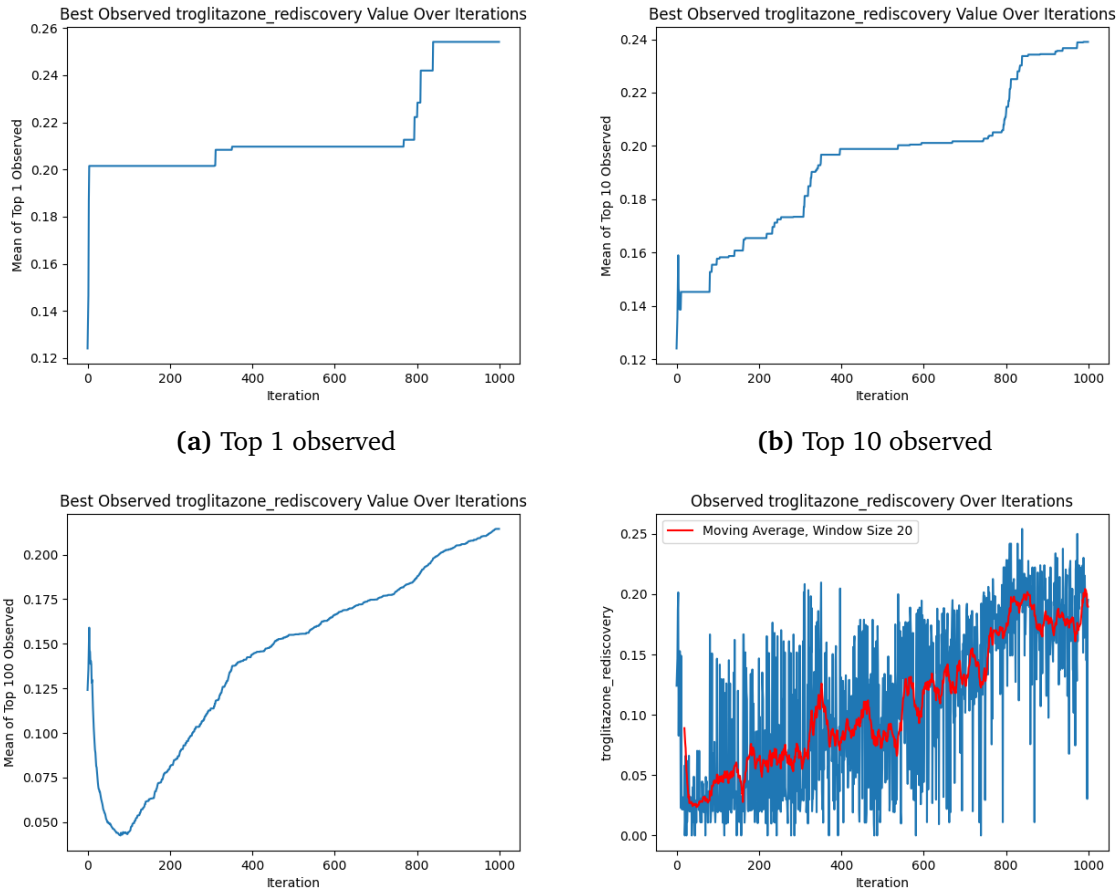
Figure 4.1 shows how much time a single iteration takes when the number of iterations grows. The points by which the graphs for the baseline solution and for the implemented solution without kernel approximation are plotted are taken to be average over 20 iterations because the time of iterations, especially for the baseline solution, varies significantly depending on how many iterations have passed since fitting. The graph depicts only the first 1000 iterations, but for the rest of the 5500 iterations of the GPU-utilizing solution, the time to do one rarely goes over 7 seconds.

The results above, show how well the implementation that utilises the RBF kernel with GPU acceleration scales to a larger number of points, while the baseline solu-

tion with graph kernel makes it very computationally costly to search for more and more molecules. Even if we had much greater computational power, it is clear that that would not help much as the rate with which the computations become more expensive for the baseline solution grows rapidly.
The solution with the kernel approximation could not run for many more than 350 iterations due to running out of GPU memory. Additional experiments with kernel approximation were done on the CPU to compare the results to the initial solution, however, the iterations were taking so much time that those results can be considered irrelevant.

4.3 Qualitative Results

In this section, we look at the accuracy and predictive power of different models. Obviously, the greater speed would be useless, unless it could be capitalized on for getting better results in the same time frame.



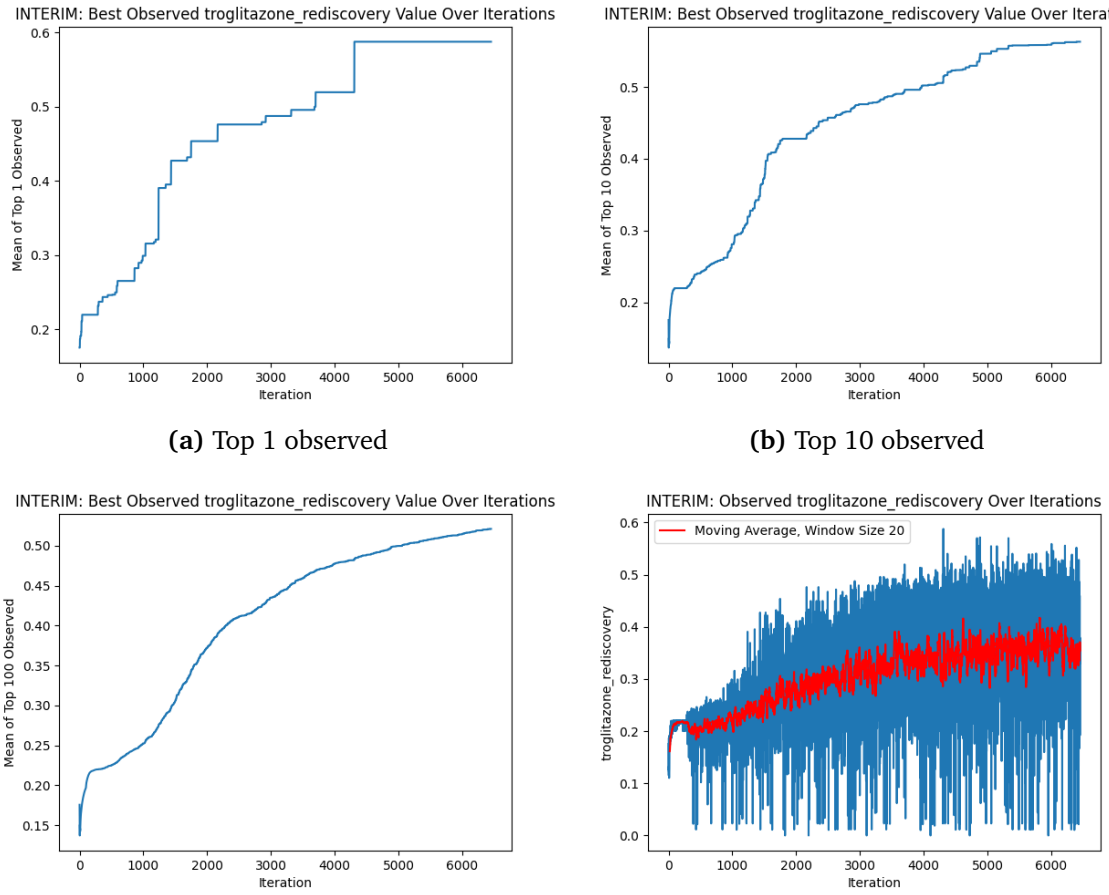
(a) Top 1 observed

(b) Top 10 observed

(c) Top 100 observed

(d) Average score

Figure 4.2: Baseline results for troglitazone rediscovery



(a) Top 1 observed

(b) Top 10 observed

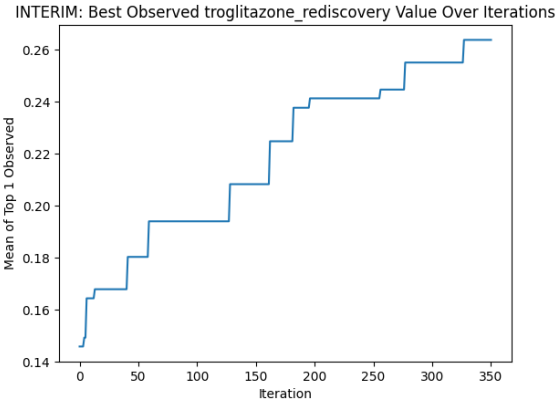
(c) Top 100 observed

(d) Average score

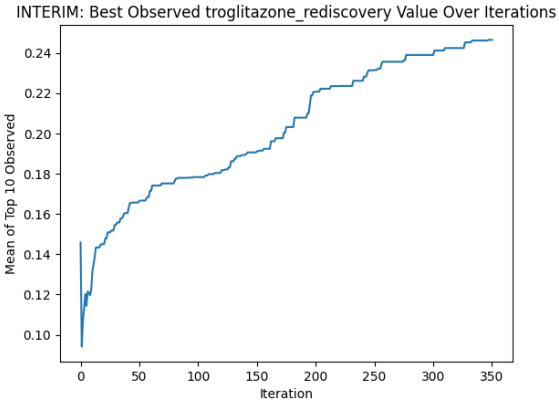
Figure 4.3: GPU results for troglitazone rediscovery

We can look at figures 4.2, 4.3, and 4.4, which show how the best scores grow with the number of iterations of three solutions for the troglitazone rediscovery oracle function. It's easy to notice that the corresponding numbers of iterations of the first two models produce very similar results and the implementation with kernel approximation reaches a higher score a bit faster, getting above 0.26 with its 350 iterations. Another significant thing to note is that after the 1000th iteration, the GPU implementation without approximation keeps steadily improving, showing that the final score that the baseline solution produces is far from optimal.

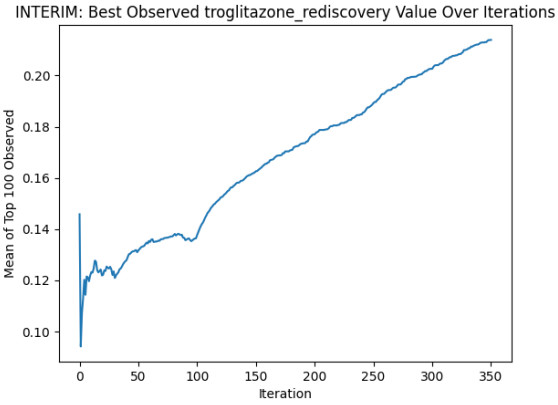
Figures 4.5, 4.6, and 4.7 show the outcomes for the thiothixene rediscovery oracle. These results validate the exact points, introduced above, confirming that the GPU implementation with and without kernel approximation does not lose in accuracy to the initial program.



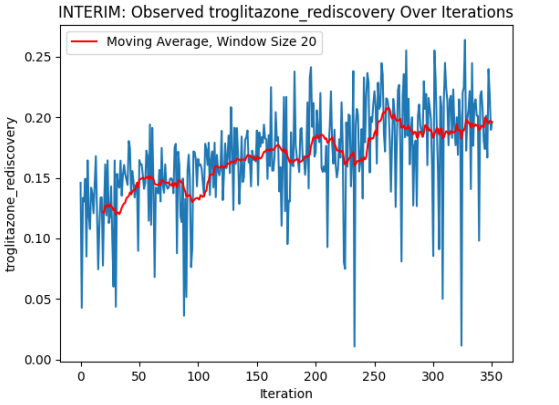
(a) Top 1 observed



(b) Top 10 observed

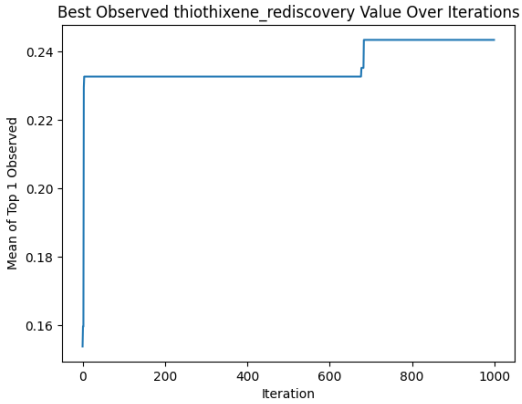


(c) Top 100 observed

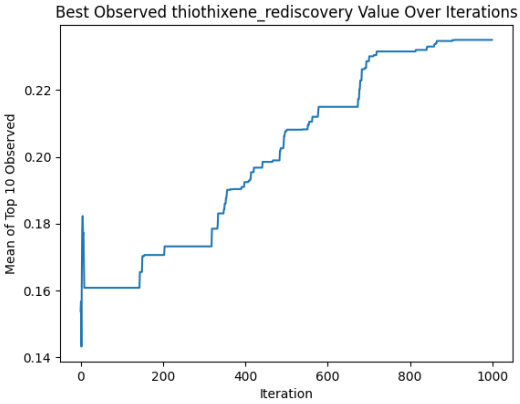


(d) Average score

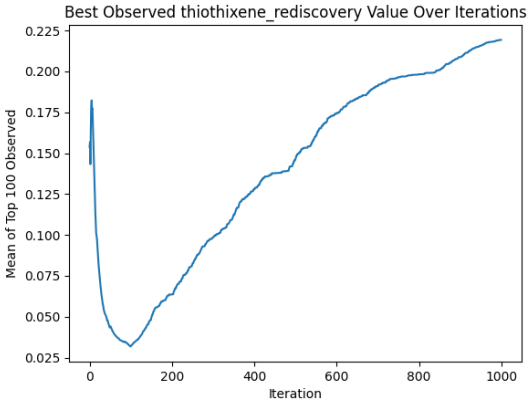
Figure 4.4: GPU + Approximation results for troglitazone rediscovery



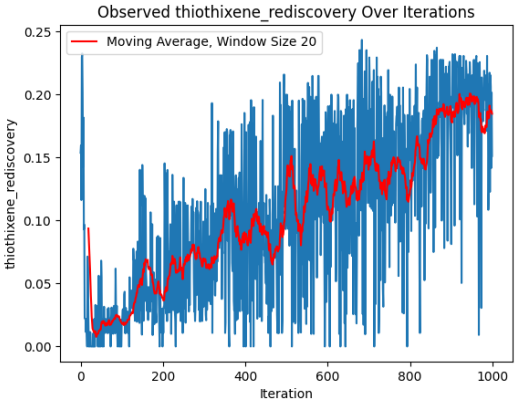
(a) Top 1 observed



(b) Top 10 observed

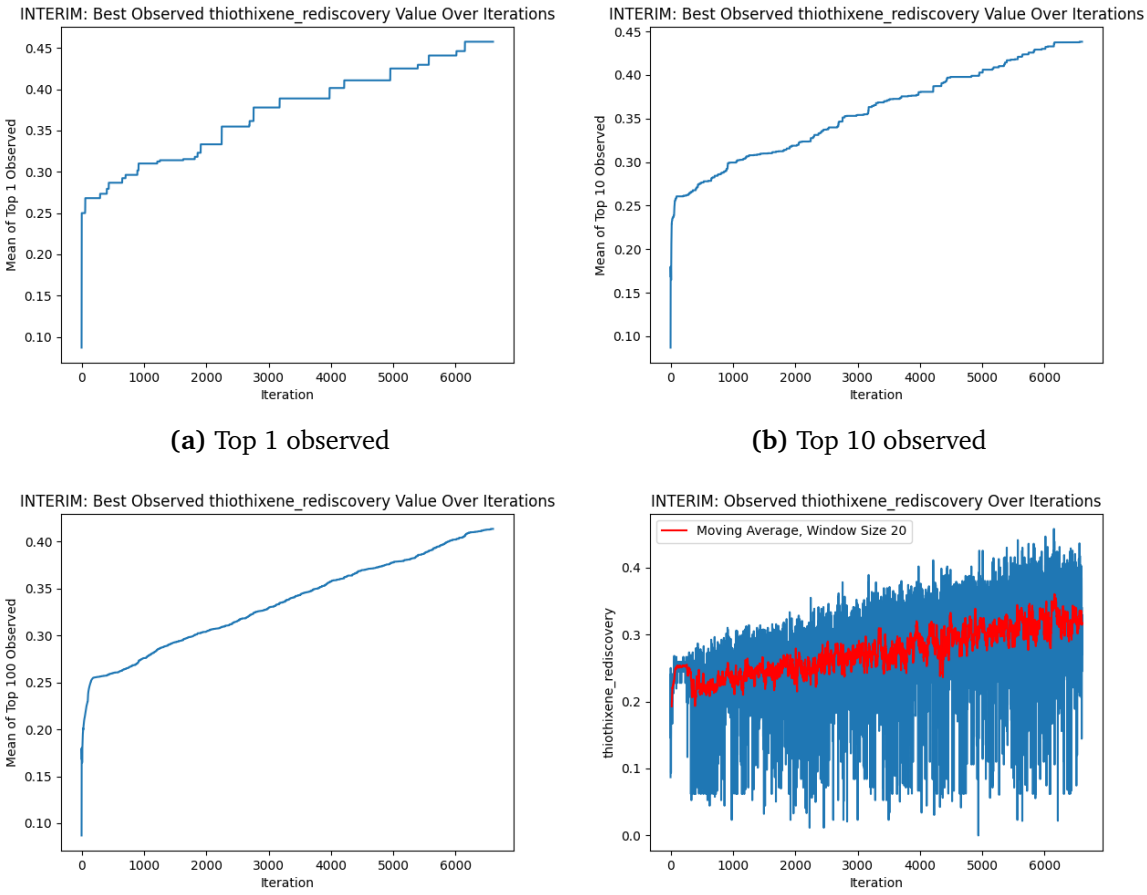


(c) Top 100 observed



(d) Average score

Figure 4.5: Baseline results for thiothixene rediscovery



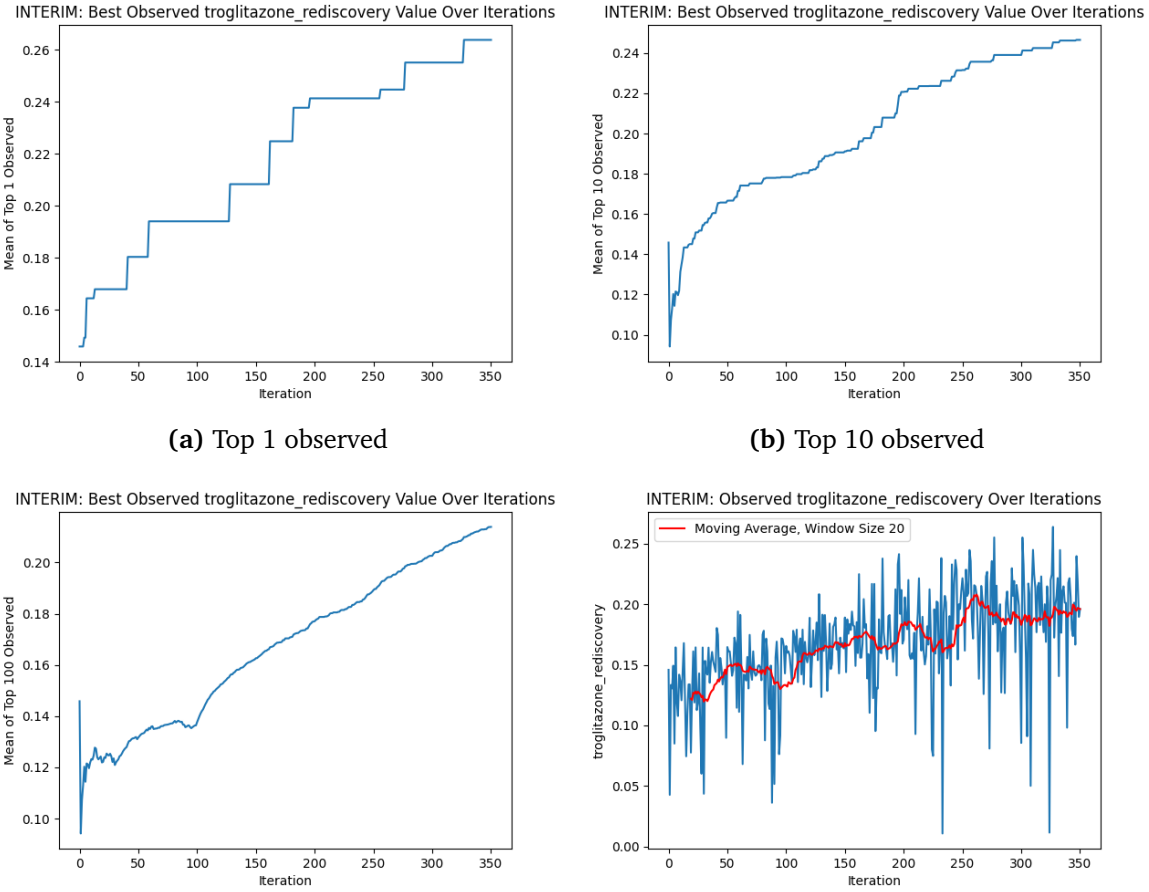
(a) Top 1 observed

(b) Top 10 observed

(c) Top 100 observed

(d) Average score

Figure 4.6: GPU results for thiothixene rediscovery



(a) Top 1 observed

(b) Top 10 observed

(c) Top 100 observed

(d) Average score

Figure 4.7: GPU + Approximation results for thiothixene rediscovery

Chapter 5

Conclusion

5.1 Evaluation

The qualitative results demonstrate that the implementation with RBF kernel that utilizes GPU allows conducting the Bayesian optimization on graphs with many more points, not requiring much more time at the stages where the base solution reaches too long to produce even one iteration. This gives an option to scale the molecular search drastically.

In our case, the higher search speed does not mean worse accuracy. On the same iteration, all the solutions produce comparable results, but we just get those results much faster if we use the implementation introduced in this project. The initial program will struggle to find a good molecule settling on a non-optimal result before reaching the computational time limit. On the other hand, with more GPU memory the RBF implementation could go on, iterating through many more points and improving the best score, which makes this approach clearly superior.

The kernel approximation, on the other hand, did not show good performance in terms of speed or memory, however figures 4.4 and 4.7 show potential in terms of accuracy as the BO had reached high scores (¿0.26 in both) in its 350 iterations. But, in the current form, it is not worth running the Bayesian optimisation on the molecular graphs, which are very complex and require high dimensional data to store them, with the scalable kernel approximation for product kernels.

5.2 Future Work

There is still much to explore in the field of graph Bayesian optimization for molecules. First of all, for this work, the RBF kernel was used, because it has proven itself in machine learning and is considered to be quite universal. However other kernels and their combinations could be considered, while still being able to utilize the GPU's ability to compute in parallel. Another area to explore further is the approximations that can be used to reduce computational costs. For example, the lower rank approximation for the graph tensors could be considered, to then make it easier to

implement scalable kernel approximation. Moreover, LanczOz variance estimates (LOVE)[19] can potentially improve the speed of predictive variances and posterior sampling.

Bibliography

[1] Madzhidov T. I. Varnek A. Polishchuk, P. G. Estimation of the size of drug-like chemical space based on gdb-17 data. *Journal of computer-aided molecular design*, 27(8):675–679, 2013. URL https://doi.org/10.1007/s10822-013-9672-4. pages 4

[2] Jiaxuan You, Bowen Liu, Rex Ying, Vijay S. Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. *CoRR*, abs/1806.02473, 2018. URL http://arxiv.org/abs/1806.02473. pages 4

[3] Rafael Gómez-Bombarelli, David Duvenaud, José Miguel Hernández-Lobato, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *CoRR*, abs/1610.02415, 2016. URL http://arxiv.org/abs/1610.02415. pages 5

[4] Kazuki Yoshizoe Kei Terayama Xiufeng Yang, Jinzhe Zhang and Koji Tsuda. Chemts: an efficient python library for de novo molecular generation. *Science and Technology of Advanced Materials*, 18(1):972–976, 2017. doi: 10.1080/14686996.2017.1401424. URL https://doi.org/10.1080/14686996.2017.1401424. PMID: 29435094. pages 5

[5] Jiaxu Cui and Bo Yang. Graph bayesian optimization: Algorithms, evaluations and applications. 2018. URL https://arxiv.org/abs/1805.01157. pages 5

[6] Ryan-Rhys Griffiths, Leo Klarner, Henry B. Moss, Aditya Ravuri, Sang Truong, Samuel Stanton, Gary Tom, Bojana Rankovic, Yuanqi Du, Arian Jamasb, Aryan Deshwal, Julius Schwartz, Austin Tripp, Gregory Kell, Simon Frieder, Anthony Bourached, Alex Chan, Jacob Moss, Chengzhi Guo, Johannes Durholt, Saudamini Chaurasia, Felix Strieth-Kalthoff, Alpha A. Lee, Bingqing Cheng, Alán Aspuru-Guzik, Philippe Schwaller, and Jian Tang. Gauche: A library for gaussian processes in chemistry. 2023. URL https://arxiv.org/abs/2212.04450. pages 5

[7] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*, volume 2. MIT Press, 2006. ISBN 978-0-262-18253-9. pages 6, 7

[8] Donald Jones, Matthias Schonlau, and William Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 12 1998. doi: 10.1023/A:1008306431147. URL https://www.researchgate.net/publication/235709802_Efficient_Global_Optimization_of_Expensive_Black-Box_Functions. pages 8

[9] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. pages 129–143, 2003. URL https://link.springer.com/chapter/10.1007/978-3-540-45167-9_11#citeas. pages 9

[10] K.M. Borgwardt and H.P. Kriegel. Shortest-path kernels on graphs. pages 8 pp.–, Nov 2005. ISSN 2374-8486. URL https://ieeexplore.ieee.org/document/1565664. pages 9

[11] Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1365–1374, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336642. doi: 10.1145/2783258.2783417. URL https://doi.org/10.1145/2783258.2783417. pages 9

[12] Marc G. Genton. Classes of kernels for machine learning: a statistics perspective. 2, 2002. ISSN 1532-4435. URL https://dl.acm.org/doi/10.5555/944790.944815. pages 11

[13] Jacob R. Gardner, Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration, 2021. URL https://arxiv.org/abs/1809.11165. pages 12

[14] John P. Cunningham, Krishna V. Shenoy, and Maneesh Sahani. Fast gaussian process methods for point process intensity estimation. page 192–199, 2008. doi: 10.1145/1390156.1390181. URL https://doi.org/10.1145/1390156.1390181. pages 12

[15] Andrew Gordon Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp), 2015. URL https://arxiv.org/abs/1503.01057. pages 13

[16] Jacob R. Gardner, Geoff Pleiss, Ruihan Wu, Kilian Q. Weinberger, and Andrew Gordon Wilson. Product kernel interpolation for scalable gaussian processes. 2018. URL https://arxiv.org/abs/1802.08903. pages 13

[17] Janice Parker. Troglitazone: The discovery and development of a novel therapy for the treatment of type 2 diabetes mellitus. *Advanced drug delivery reviews*, 54:1173–97, 12 2002. doi: 10.1016/S0169-409X(02)00093-5. URL https://www.researchgate.net/publication/11069934_Troglitazone_The_discovery_and_development_of_a_novel_therapy_for_the_treatment_of_Type_2_diabetes_mellitus. pages 14

[18] K Davison, P Leyburn, and H A McClelland. Thiothixene in schizophrenia. *BMJ*, 4(5577):489–489, 1967. ISSN 0007-1447. doi: 10.1136/bmj.4.5577.489-b. URL `https://www.bmj.com/content/4/5577/489.1`. pages 14

[19] Geoff Pleiss, Jacob R. Gardner, Kilian Q. Weinberger, and Andrew Gordon Wilson. Constant-time predictive distributions for gaussian processes. 2018. URL `https://arxiv.org/abs/1803.06058`. pages 23