OPTIMISATION AND VERIFICATION OF SYSTEMS REPRESENTED USING MACHINE LEARNING MODELS

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Boning Cui

Supervised by Dr Konstantin Korovin

Department of Computer Science

Contents

Word Count: 10763

List of Tables

List of Figures

Abstract

Optimisation and verification towards machine learning models are interesting yet challenging research areas. This project focuses on the implementation of a hybrid algorithm combining Bayesian optimisation and Satisfiability Modulo Theories (SMT) solvers, which can compute a formally verified optimum region of machine learning models with certain constraints being satisfied, and then uses it to reason various types of ML algorithms and compares their performance.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank Dr Konstantin Korovin for his guidance when I was uncertain about my objectives, and his constructive feedback during our meetings that deepened my understanding of the core algorithm for this project throughout my research journey.

Chapter 1

Introduction

1.1 Motivation

Machine learning models, such as neural network, decision tree or random forest, have proven to be effective for capturing patterns in complex data that typically cannot be modelled directly. These models offer efficient and automated procedures for building an approximation of an unknown function using historical data, which can achieve high predictive accuracy if trained and tuned properly.

Given a well-trained machine model, it is then usual and natural to analyse and reason this model to find its optimal output(s). Consider a single-output regression task with input feature vector $\mathbf{x}$ as numerical control signals for a machine and output y as a measurement of its performance divided by energy cost. It is ideal to find input features that yield a global maximum output. However, unlike function that represents directly modelled systems and is expressed explicitly as mathematical formulas, whose optimal value(s) can be found and verified using its 1^{st} and 2^{nd} derivatives, machine learning models define complex and various relations between features and output that usually can neither be fully expressed as formulas nor be differentiated. Therefore, an alternative is desired for optimisation and verification problems towards functions represented as machine learning models.

In practice, it is quite common that input values have fluctuations – the actual input value can fluctuate within a small range around the theoretically ideal value. Moreover, machine learning models are only approximations for the real, unknown function no matter how accurate they can predict unseen data, which provides no guarantees that an optimum point found in a machine learning model is indeed the optimum or even near-optimum point for the real function. Taking those two facts into consideration, finding an optimum region in which the minimal value is optimised, instead of just an optimum point, is a more robust and realistic approach. Furthermore, there may also be additional constraints on specific features or relations between different features that cannot be modelled by machine learning models. An optimiser capable of taking those additional constraints into account while doing optimisation and verification of machine learning models has the flexibility to adapt to various usage scenarios. An algorithm called $GEAROPT_\delta - BO$, combining Bayesian optimisation and SMT solver to achieve such purpose has been proposed in [4], which takes a machine learning model and finds a formally verified global optimum feature space, whose size is determined by user-specified radius for each feature (e.g. the inevitable fluctuation interval of a feature variable), such that the minimal output value that can be obtained by input features inside this region is maximised, while satisfying all the additional constraints.

This project focuses on the implementation of this hybrid algorithm, the comparison of its performance between different machine learning models, potential efficiency improvements for certain machine learning model and usage extension to other machine learning models.

1.2 Aims and Objectives

As briefly described in the motivation section, this project has the following aims and objectives:

1. Code implementation of the algorithm, $GEAROPT_\delta - BO$, referring to the pseudocode presented in [4].

2. Performance comparison between different machine learning models (currently it supports polynomial model, neural network with ReLu/linear activation, decision tree, random forest), including: training and fine-tuning of those models, essential translations from machine learning models into their equivalent SMT formulas, comparison between different models on their prediction accuracy against algorithm solving times for different types of data sets.

3. For decision tree model, explore how different SMT representations influence the algorithm's solving time, as the complexity of the tree structure increases.

4. Investigate the potential for extending the algorithm's application to other machine learning models.

1.3 Report Structure

This report contains five chapters:

- Chapter 1 presents an introduction to the project.
- Chapter 2 presents the background behind the project, mainly regarding the two essential components of the hybrid algorithm, Bayesian optimisation and SMT solving technique, with introductions to the machine learning models suitable for the algorithm.
- Chapter 3 presents methodology and actual implementations of the project, including the construction of the algorithm, training and tuning strategies for different machine learning models, their equivalent SMT representations and variations, and extension to other machine learning models.
- Chapter 4 presents experiments and evaluations regarding performance comparison between different models, time usage comparison between different types of SMT representation of decision trees.
- Chapter 5 discusses conclusion, limitations and potential improvements of the project.

Chapter 2

Background

This chapter presents the background of the project:

In Section 2.1 and Section 2.2, the two key components, Bayesian optimisation and SMT (satisfiability modulo theories) solver, are introduced, together with their general procedural, strengths and limitations. The combination of those two techniques forms the core algorithm used in this project, aiming to find the verified global optimum of machine learning models.

In Section 2.3, related research regarding optimisation for machine learning models is presented.

2.1 Bayesian Optimisation

2.1.1 Overview

Bayesian optimisation is a popular technique for finding global optimum of an objective function. Since it requires neither any known special structures such as concavity or linearity for efficient optimisation nor the 1^{st} or 2^{nd} derivatives (typically used in optimisation methods like gradient descent and Newton's method) of the objective function, it is especially suitable for black-box functions of which the relationship between its input variables and output is not explicitly known or is complex, and the evaluation is time-consuming or costly[12]. For example, a deep neural network or decision tree.

Bayesian optimisation consists of two parts: 1) a probabilistic surrogate model approximating the objective function, using Gaussian process regression to estimate uncertainty at unseen points; and 2) an acquisition function that utilises the surrogate model, giving suggestion on the next sample point that is more likely to yield optimum output[12]. Bayesian optimisation iterates over these two steps: acquisition function makes suggestion based on the current surrogate model and the surrogate model gets updated using this suggested data point, till max iteration times have been reached.

2.1.2 Gaussian Process

The goal of the surrogate model is to approximate the black-box function and make predictions on unseen data as accurate as possible. Two common approaches for this problem are 1) to define the surrogate from a restricted class of functions, such as considering only linear relation between input and output, which has an obvious disadvantage – predictions may be poor due to the inability to capture data patterns using only a limited class of functions; or 2) to assign a prior probability to every possible function and make prediction as a weighted combination based on their probabilities, which also has a shortcoming because it is unrealistic to compute probability of infinitely many possible functions in finite time[25]. Gaussian process comes as a solution for those two problems.

For a single dimensional input x and output y, x follows a Gaussian distribution with mean μ and variance σ^2 if its probability density function can be expressed as[35]:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

In the context of machine learning problems, many input data has more than one feature variable. The multivariate Gaussian distribution is thus introduced as a generalisation of the (univariate) Gaussian distribution for data with multiple input variables. A random vector $\mathbf{x} \in \mathbb{R}^n$ follows a multivariate Gaussian distribution in n dimensions with mean vector μ and covariance matrix Σ if its probability density function is[35]:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} exp\left(-\frac{(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}{2}\right)$$

A Gaussian process is a generalisation of the Gaussian probability distribution, which arises when the multivariate Gaussian is extended to infinite dimensions, depicting a group of random variables, each of which has a multivariate Gaussian distribution[25]. It can then be used for modeling input-output mapping in supervised machine learning tasks like regression and classification, with a quantified estimation of uncertainty. With respect to this project, only Gaussian process regression is considered, which models probability distribution over possible functions with a continuous domain.

Back to the surrogate model used in Bayesian optimisation, the aim here is to model the true function, utilising Gaussian process regression and sequentially observed sample points. Initially, Gaussian process regression (GPR) defines a prior distribution before any data observations, displaying possible functions that are expected to be sampled by prior understandings. The prior distribution is described by a particular mean vector and covariance (kernel) matrix[12]:

- The mean vector (or function, if thinking function as a very long vector, each entry of which referring to the function value at a particular input point[25]) determines the expected value of the function being modelled at each point in the input space, before observing any data. Usually it is initialised to zero, meaning that the averaged function value at any input point is zero. Taking the prior distribution of a 1-dimension input space (shown in 2.1a) as an example, here only four random functions are sampled and the expected function value at point $x \approx 6$ is not 0, despite a zero-initialised mean function. However, if more and more functions are sampled, function values will gradually converge to zero. Moreover, since there are no actual constraints on the mean vector, it can be modified according to any prior knowledge about the target function, such as linearity, to give better prediction and fast convergence.

- The covariance matrix or the kernel function measures the pairwise similarity in the input space and is constructed by applying a covariance function at each pair of points, following an intuition that the closer (distance calculated by any norm) these two points are to each other in input space, the more similar they are and thus they should have more similar function values than points far away from each other[12]. It turns out that the choice of kernel function plays a vital role in characterising the behaviour of a Gaussian process and deciding which functions are more likely to be sampled. There are many pre-defined kernel functions, capturing different kinds of correlations between data points. For example, the Radial Basis Function (RBF) prefers to sample functions that are smooth and differentiable, while the Periodic Kernel Function is appropriate if the target function is believed to be periodic. One can even use customised kernel to fulfill specific requirements based on smoothness, sparsity, drastic changes or differentiability[11]. Additionally, kernel functions also have hyperparameters that can be tuned, offering even more flexibility to adapt to a particular type of GPR task.

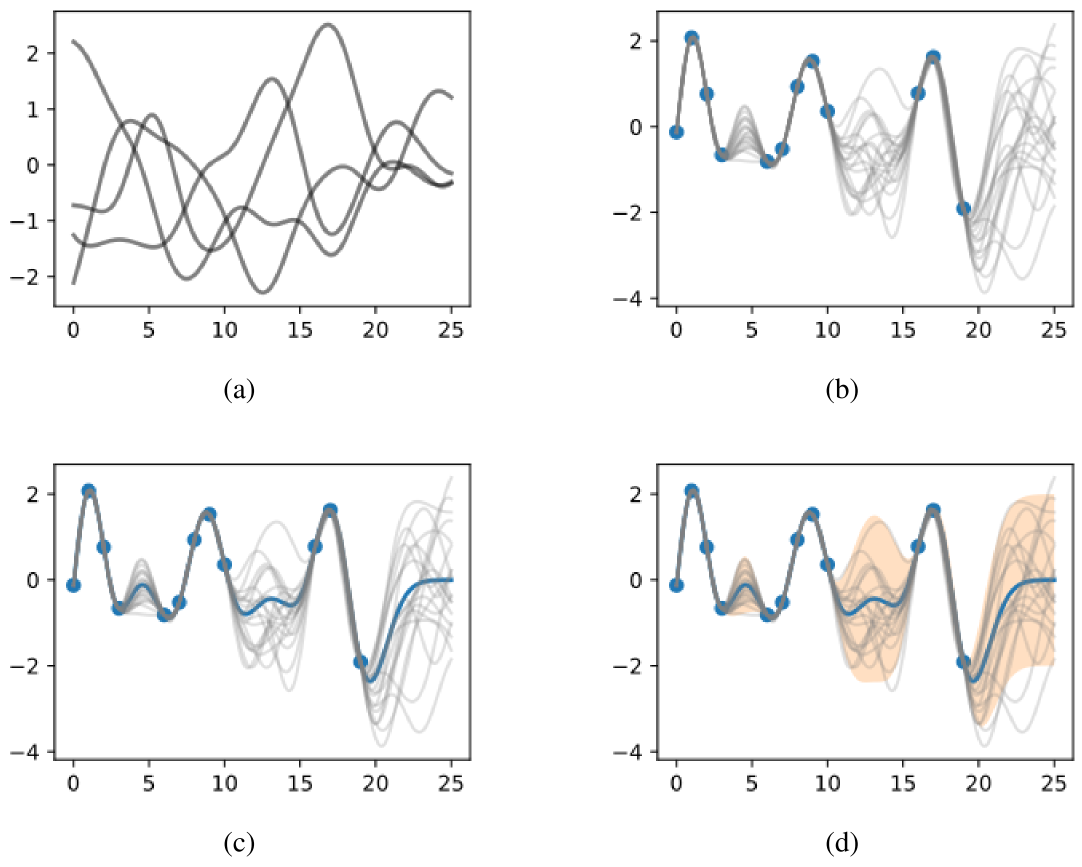


Figure 2.1: Example Gaussian process regression visualisation[10].

The characteristic of the prior distribution is important, as it fixes the properties of the functions considered for prediction[25]. Next, after observing some data points, the prior distribution is updated to yield a posterior distribution over functions that match the data best. For an one-dimensional example shown in Figure 2.1, the prior sample functions in 2.1a are updated in 2.1b to match observed data represented as blue dots. Obviously, there is no uncertainty at the observed data points, and the further some input areas are away from the known data points, the more unpredictable the function values are within those areas. As more and more points are observed, the uncertainty will reduce and the probabilistic distributed function will approximate the target function better, if the mean and kernel functions are appropriately chosen. In 2.1c, a surrogate model is built by combining the prior beliefs defined in the prior distribution and the new information from the observed data. Figure 2.1d colours the uncertainty with a 95% confidence interval.

Compared to the two approaches of approximating target function mentioned before, Gaussian process regression does have advantages as it neither fixes itself within a restricted class of functions, nor tries to assign prior probability to all possible functions. Instead, it uses a flexible setting of prior beliefs, determined by its mean vector and kernel function and hyperparameters within, which can all be adjusted and tuned with experiments to fit a specific type of data set, and generates posterior beliefs according to sequentially observed data points. Next, the surrogate model utilising GPR is used by an acquisition function to suggest a potential optimum input point.

2.1.3 Acquisition Function

In the iteration loop of Bayesian optimisation, acquisition function serves as the second component, trying to propose an optimum point based on the current surrogate model. It is a heuristic function that is cheaper to evaluate and also balances exploitation, which focuses on the current best results, and exploration, which pushes the search towards regions that have not yet been explored. Making use of the mean and variance of the current posterior distribution, the problem is transferred from optimising the black-box function to optimising the acquisition function.

Most commonly used acquisition functions are probability of improvement, expected improvement and entropy search:

- Probability of Improvement (PI) is considered as the first acquisition function designed for Bayesian optimisation. Suppose n points are observed so far, f_n^* is the maximum function value among those points (optimisation as maximisation here for convenience) and $\mu_n(x)$ and $\sigma_n^2(x)$ are the mean and variance generated by the current posterior distribution respectively, PI measures the probability that a candidate point yields function value larger than the current optimum f_n^*, expressed as[13]:

  $$\alpha_{PI}(x) = P(f(x) \geq f_n^*)$$

  An algorithm for PI has been proposed in [17], with its simplified expression:

  $$\alpha_{PI}(x) = \Phi\left(\frac{f_n^* - \mu_n(x)}{\sigma_n(x)}\right),$$

  which utilises properties of the current posterior distribution. Naturally, the next point to be evaluated can be computed by optimising the acquisition function:

  $$x_{n+1} = argmax\, \alpha_{PI}(x),$$

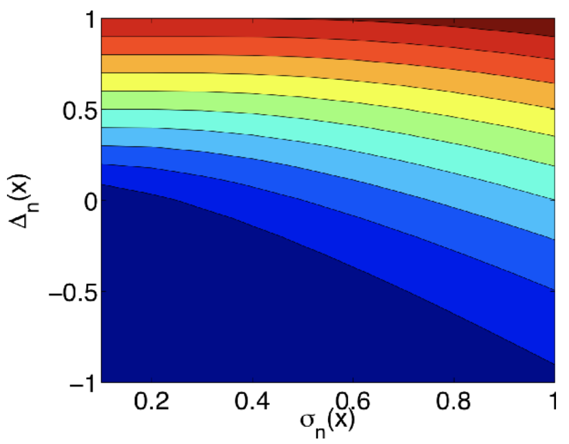


Figure 2.2: Contour plot of EI[12]. Blue indicates smaller values and red larger values.

which is the point with the highest probability to give a larger function value than the current best. This is feasible because the acquisition function is inexpensive to evaluate compared to the target function and allows continuous 1^{st} and 2^{nd} order derivatives to solve optimisation problems[12].

- One problem of PI acquisition function is that it only considers the probability of a point to produce a better optimum, not how much it is improved from the current optimum. This could make the search stuck in local optimum regions without sufficient exploration. An alternative acquisition function trying to resolve this problem is the Expected Improvement (EI). EI measures the extent of improvement a point can offer compared to the current best, whose ideal expression is:

$$\alpha_{EI}(x) = max(0, f(x) - f_n^*)$$

Since $f(x)$ is unknown, the expected improvement is instead computed using the posterior distribution as[12]:

$$\alpha_{EI}(x) = max(0, \Delta_n(x)) + \sigma_n(x)\phi(\frac{\Delta_n(x)}{\sigma_n(x)}) - |\Delta_n(x)|\,\Phi(\frac{\Delta_n(x)}{\sigma_n(x)}),\ \Delta_n(x) = \mu_n(x) - f_n^*$$

In this equation, high $\Delta_n(x)$ appears near previously observed points with high function values, representing the exploitation term – how to make use of good observations, while high $\sigma_n(x)$ refers to regions with high uncertainty due to insufficient exploration so far. As shown in Figure 2.2, the EI value increases as both $\Delta_n(x)$ and $\sigma_n(x)$ increase, balancing the trade-off between exploitation and exploration[12].

- Another common approach is the Entropy Search (ES) acquisition function, which makes suggestion based on the current information about the global optimum according to its differential entropy, seeking point to sample next that would reduce its differential entropy (i.e. uncertainty) most[12]. Suppose x^* yields the optimum f_n^* in the n observations so far, the posterior distribution will have a probability distribution over x^*. What the ES function measures is the decrease of the entropy of the posterior distribution over x^* from time n to time $n+1$ after a point x is observed, with its ideal formula[12]:

$$\alpha_{ES}(x) = H(P_n(x^*)) - H(P_n(x^*|f(x)))$$

  The more decrease a point can cause with respect to the entropy of posterior distribution over the current optimum x^*, the more likely the point will be sampled next. Since the target function $f(x)$ is unknown, the expected entropy is approximated using mean $\mu_n(x)$ and variance $\sigma_n^2(x)$. Unlike PI and EI, ES selects points by measuring how they change the posterior over the whole input space instead of only the improvement they can bring compared to f_n^*, which is more useful in some certain kinds of tasks[12].

All of those acquisition functions utilise the posterior distribution from the Gaussian process, generated by previous observations, and choose the most promising point to sample according to different heuristics. The choice of acquisition function is indeed important as it affects accuracy, convergence rate and time consumption. In the next chapter, the construction of the main algorithm uses a hybrid acquisition function in its Bayesian optimisation component, which employs among a set of standard acquisition functions randomly.

Together with Gaussian process regression, Bayesian optimisation now has its core components and is able to iterate over:

- Update the posterior distribution in GPR to fit data better, utilising new observations;
- Suggest next sample point based on the current posterior distribution and the selected acquisition function.

The effectiveness to locate near-optimal point makes Bayesian optimisation a widely used technique for optimisation problem in the context of artificial intelligence and machine learning.

2.1.4 Strengths and Limitations

Bayesian optimisation is good at finding near-optimal points for functions that are expensive to evaluate, usually with only a small number of observations. It can also encode prior knowledge about the target function, such as linearity, smoothness or periodicity, through the prior distribution in the Gaussian process. The ability to experiment, choose and even customise the suitable mean function, kernel function and hyperparameters within gives it the flexibility towards various objective functions[25]. Moreover, the GPR enables uncertainty estimation, which is usually preferred in AI-related tasks. Last but not the least, Bayesian optimisation performs better than many gradient-based optimisation techniques with respect to local optimum escaping, with the help of an acquisition function well balancing exploitation and exploration.

However, there are also limitations. Its performance is hugely influenced by the continuity and the number of dimensions of the objective function – preferably a continuous input space with dimensions less than twenty[12]. The flexibility of prior knowledge encoding can also be a double-edge, which makes the performance highly depend on the choices of the mean and the kernel function and also sensitive to hyperparameter settings, requiring more time to tune and test.

More importantly, Bayesian optimisation by its nature is a probability-based optimisation technique, meaning it can neither provide formal guarantees on the suggested point (after several observations) being the real optimum or even near-optimum, nor can it ensure that the region closely around this suggested optimum point also has near-optimum function values[4]. Such requirements for formal verification and near-optimum regions are essential in the practical scenarios in which perturbation of inputs is inevitable yet safety considerations are strict. SMT solving technique comes as a rescue for this problem.

2.2 Satisfiability Modulo Theories

2.2.1 Overview

Satisfiability Modulo Theories, or SMT, is the problem of checking whether a first-order formula is satisfiable with respect to some background theory, within which the interpretations of certain predicate and function symbols, such as $<$, $+$, $\neg$, are fixed[2]. For example, a background theory interested only in integer arithmetic does not concern other interpretations of $+$ other than as the function symbol of integer addition.

SMT generalises the SAT problem over boolean variables to the satisfiability problem of complex formulas that may contain integers, real numbers or even data structures such as lists, arrays and strings. The motivation behind is the growing demand for formal verification and effective constraints satisfaction checking in many areas, such as artificial intelligence and robotics, software and hardware development, computer-aided design and information security. In these areas, the problem's formula usually requires a more expressive representation that supports various types of variables and incorporated theories, rather than just propositional formulas with boolean variables[2].

2.2.2 Components

For most Satisfiability Modulo Theories solvers, the satisfiability of a SMT formula is reasoned using a combination of: 1) a SAT solver that performs efficient satisfiability checks (via algorithm like DPLL) for propositional logic by representing the original formula as a conjunction of disjunctions of atomic sub-formulas or their negations with respect to one or several background theories and treating them as boolean variables without worrying their internal structures; and 2) theory solvers that handle those SMT formulas and try to find a model satisfying the boolean assignment generated from the SAT solver[8].

2.2.2.1 SAT solver

A SAT solver solves the satisfaction problem of a propositional formula, expressed by boolean variables combining the fundamental logic connectives: negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$ and bidirectional implication $\leftrightarrow$[9]. By applying a series of standard steps, such as implication elimination, De Morgan's Law, the distributive law, etc., any propositional formula can be translated into its equivalent conjunctive normal form (CNF), expressed as a conjunction of clauses. Each clause is then a disjunction of literals that are either a boolean variable or its negation form.

A technique widely used in SAT solvers, called the DPLL (Davis-Putnam-Logemann-Loveland) algorithm, utilises this CNF equivalence. To make the formula evaluate to true, or satisfiable, at least one boolean variables' assignment must exist, such that every clause has at least one literal element that evaluates to true without any contradictions. To seek such an assignment or to prove unsatisfiability, the DPLL algorithm has three main operations[8]:

- **Propagation** is the process to eliminate any (obtained) unit clause containing only one literal and deal with the resulting consequences in other clauses. Since the CNF is true if and only if all its clauses are true and an unit clause contains only one literal, then this literal must be true to satisfy the unit clause, indicating a compulsory assignment for an variable. This new assignment can be used to analyse other clauses and it is the case that either a literal in some clause evaluate to true due to the assignment and thus the whole clause is also true, or the literal is false and can be removed from its clause, as it will not contribute to the satisfiability of its clause. The final result of applying propagation iteratively is one of: 1) a CNF containing only non-unit clauses that needs further exploration; 2) a satisfied CNF with all its clauses evaluate to true; 3) a CNF containing empty clause (obtained from removing all its literals due to their falsehood) that cannot be satisfied by the current variable assignments following the path.

- **Selection** is called when a CNF with non-unit clauses is obtained after propagation. What the selection does is to choose a literal according to a heuristic and add it as a new unit clause to the CNF, which will then trigger a new propagation. There are many heuristics for selection, such as *Maximum Literal Frequency* (MLF), which selects the most frequently occurring literal in all clauses, *Maximum Occurrences in Minimum Size Clauses* (MOMS), which prioritises literals that occur most frequently in the smallest clauses.

- **Backtracking** is called when the iteration over the first two steps results in any empty clause, meaning all the literals in that clause are evaluate to false and it is thus unsatisfiable. This operation then backtracks from the dead end to the nearest splitting node that has not yet been fully expanded and branches to a new path by trying a different assignment and repeat the first two steps.

By applying the DPLL algorithm on a CNF, one can either get a satisfied CNF at the end of one search path and a model assignment constructed by collecting all the selected literals on the path, or the occurrence of empty clauses in every branch, which shows the unsatisfiability of the original formula.

However, the DPLL algorithm does not scale well with respect to the number of boolean variables. Since every variable has two possible assignments, the time complexity in the worst case for n variables is $O(2^n)$. The space usage can also be huge for storing the modified CNF and the current partial assignments, in case backtracking is called.

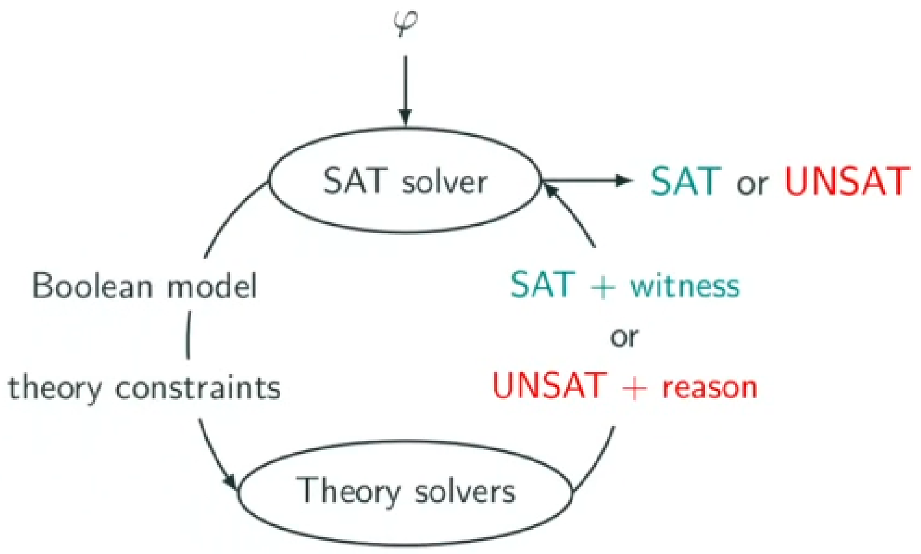


Figure 2.3: General solving cycle of a SMT solver[16].

2.2.2.2 Theory solvers

A background theory refers to a specific domain, such as integer/real arithmetic, bit vectors or arrays, in which only SMT formulas that satisfy the specified representation and data types are considered. For instance, the commonly used difference arithmetic (DA) theory deals with constraints of the form[9]:

$$x - y \leq c,$$

where x, y are variables and c is an integer constant. A theory solver then tries to either find a model assignment for a SMT formula specified by this theory or prove its unsatisfiability.

Compared to propositional formulas constructed by boolean variables and basic logic connectives, the SMT formulas consist of atomic formulas (i.e. boolean expressions) with respect to a particular theory domain or a combination of various theories, such as $x - y \leq c$ in the DA theory, together with basic logic connectives and possibly quantifiers in first-order logic as well, despite the extra computational complexity they introduce.

2.2.2.3 Combination

With the help of the SAT solver and the theory solvers, the general solving procedural of a SMT solver can be described as shown in Figure 2.3, where ϕ is the SMT formula to be examined. The basic idea is[8]:

- First, use the DPLL-based SAT solver to propose a possible (boolean) assignment, which is achieved by translating the SMT formula into CNF form and creating an abstraction that maps all the atomic formulas in this SMT formula into fresh boolean variables. For instance,

  $$\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5) \ \textit{is translated into} \ \neg p_1 \wedge (p_1 \vee p_2)$$

  The SAT solver does not care what is inside those new boolean variables but only treats them as a CNF in propositional logic. Obviously, if it proves unsatisfiability, then the original SMT formula is unsatisfiable. Otherwise, a model assignment is returned. In this example, $\{p_1 \rightarrow false,\ p_2 \rightarrow true\}$.

- Next, the theory solvers take a look inside those boolean variables and check whether the atomic formulas they represent are still satisfiable according to the assignment from the SAT solver. If so, then the assignment or solution produced by the theory solvers is a model for the original SMT formula. Otherwise, there is a disagreement between the SAT solver and theory solvers, such as in this example, where $\{\neg(a \geq 3),\ a \geq 5\}$ is unsatisfiable in its theory domain. What is important next is that the theory solvers need to inform the SAT solver that the (boolean) assignment it returns is invalid to prevent the SAT solver from proposing the same assignment again. Since the SAT solver treats atomic SMT formulas as boolean variables and does not understand their internal structure, the communication between the two solvers is achieved by adding the negation of the proposed assignment, called theory lemma, into the original formula[8]. The resulting formula in this example is:

  $$\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5) \wedge (a \geq 3 \vee \neg(a \geq 5)),$$

  which is then translated into:

  $$\neg p_1 \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2)$$

Consequently, the SAT solver is forced to try different assignments other than the previous invalid one.

These two steps are repeated and terminate until either the SAT solver shows unsatisfiability, meaning that the original SMT formula is unsatisfiable, or the theory solvers output an assignment, which is a model for the original SMT formula.

2.2.3 Strengths and Limitations

The ability of deciding satisfiability of formulas that consist of rich expressions representing various theory domains makes Satisfiability Modulo Theories technique a useful tool in areas that require formal verification and guarantees, such as CAD and information security. For instance, to guarantee that a system defined by some rules is indeed safe and stable, a SMT solver can be used to show the unsatisfiability of any violation of the rules. Moreover, a SMT solver can integrate multiple theories into a single problem framework, making it a versatile tool for various types of real world problems. The Z3, which is used in the code implementation of the hybrid algorithm, is such a solver developed by Microsoft Research.

However, SMT also has two main limitations. One is the bad scalability and high time and space complexity. Even the worst time complexity of the SAT component grows exponentially with respect to the input, not to mention the time and space needed for the integrated theory solvers to run, especially when the problem involves complicated interactions across multiple theories. This makes SMT solving a time consuming operation in practical use.

Another shortcoming is the limited number of well-developed theory solvers. A SMT solver can only solve formulas defined within the domain of its integrated theory solvers. There are still certain types of problems that cannot be reasoned, despite the increasing expressiveness of popular SMT solvers. One example related to this project is the non-linear activation functions, such as sigmoid, used in neural network models.

Now the two components of the algorithm have been introduced, Chapter 3 will show the implementation of the algorithm and its supported machine learning models.

2.3 Related Work

Optimisation problems regarding machine learning models are a fascinating yet challenging area. One reason is the basic requirement for a feasible optimisation procedural from an increasing number of systems represented mostly or even exclusively by machine learning models, against the uninterpretable and obscure nature of most machine learning models – they cannot be analysed the way as functions given in explicit and mathematical forms. Moreover, many systems modelled by machine learning algorithms have strict standards on certain constraints in the system to be always satisfied regardless of what the input is, such as security requirements for autonomous vehicles or subtle considerations that should be taken by large language models for generating texts suitable for all users. This has introduced another requirement, the ability to give formal verification, into machine learning related systems.

Song et al. wrote a review that focused on the intricate and dual interactions between machine learning and optimisation techniques, highlighting the potential of their integration to enhance AI-driven applications[30]. They discussed various optimisation methods used in machine learning represented systems, such as data preprocessing strategies as an optimisation problem and optimisation used in hyperparameter tuning of machine learning models, especially Bayesian optimisation. They also emphasised the role of machine learning in optimisation algorithm improvement, such as hyper-heuristics enhancement.

Xiang et al. focused on verification techniques for autonomous systems, particularly safety-critical and cyber-physical systems enabled by machine learning and artificial intelligence, especially looked at auto-driving applications[36]. They demonstrated the verification of even simple properties in neural networks is an NP-complete problem, and proposed to use activation functions in piecewise linear forms, such as Rectified Linear Unit (ReLU), to circumvent the difficulties due to the non-linearities in the neural networks.

Song et al. introduced QNN-Verifier, an open-source tool that enhances the verification of quantised neural networks through the application of SMT solvers[31]. They addressed the unique challenges posed by neural networks, particularly those influenced by finite word-length effects in fixed and floating point representations. By integrating SMT-based techniques, QNN-Verifier offered a robust solution for verifying the real-world implementations of neural networks, focusing on ensuring their reliability and accuracy in safety-critical applications.

Chapter 3

Implementation

This chapter presents the implementation part of the project:

In Section 3.1, the hybrid algorithm combining Bayesian optimisation and SMT solver, $GEAROPT_{\delta} - BO$, is introduced and implemented.

In Section 3.2, different machine learning models that are currently supported by the algorithm are discussed, including their training and tuning strategies and their equivalent SMT representations.

In Section 3.3, different methods of translating decision tree into SMT representations are investigated and their corresponding influences on the solving time of the SMT solver are compared.

In Section 3.4, efforts trying to extend the algorithm to other machine learning models are made.

3.1 Construction of the Algorithm

Recall the motivation behind, in which an optimisation algorithm running in feasible time that can reason systems represented as machine learning models and compute optimum points in input space which ensure safety, accuracy and stability with formal verification is desired, where[3][4]:

- Safety means that the computed points yield outputs satisfying pre-defined system constraints;
- Accuracy requires that the computed points yield outputs that are at least near the real optimum value;

- Stability refers to the robustness of safety and accuracy for any input points inside the regions, which are defined by the computed optimum points being stretched by user-specified perturbation in each feature dimension as well as in both directions.

3.1.1 Overview

Such an algorithm, $GEAROPT_\delta - BO$, is introduced in [4].

The *GEAR* in the name refers to the reflexively guarded $\exists^*\forall^*$ fragment, and can be expressed (with respect to the problem that this project concerns) as[3]:

$$\exists \mathbf{x_i} \forall \mathbf{x_j} \left(\theta(\mathbf{x_i}, \mathbf{x_j}) \to f(\mathbf{x_j}) \geq T \right),$$

where:

- $\mathbf{x_i}$, $\mathbf{x_j}$ are input points;
- $\theta(\mathbf{x_i}, \mathbf{x_j})$ is a guard defining a reflexive relation between $\mathbf{x_i}$ and $\mathbf{x_j}$. It typically measures the distance between the two points under p norm and decides whether they can be considered as neighbours accordingly: $||\mathbf{x_i} - \mathbf{x_j}||_p \leq r$, where r is the maximum distance allowed between two neighbours, defined by the user;
- $f(\mathbf{x_j})$ is the function value at point $\mathbf{x_j}$;
- T refers to the threshold value.

Thus, the formula can be read as: find a point $\mathbf{x_i}$ in the input space, such that the function value of any of its neighbours is no less than the threshold value T.

The *OPT* in the name then specifies that this is an optimisation process, whose goal is to maintain the satisfiability of the *GEAR* formula while maximising the threshold value T.

The δ corresponds to an extra tolerance of the neighbourhood radius, modifying the distance measurement to: $\theta_\delta(\mathbf{x_i}, \mathbf{x_j}) = ||\mathbf{x_i} - \mathbf{x_j}||_p \leq r + \delta$[3]. This will reject solutions containing counter-examples that are inside the extended safe region even if they are not inside the original region, and thus increase the strictness of accepted solutions.

The *BO* in the name refers to Bayesian optimisation, which is an essential building block of the algorithm.

As discussed in Section 2.1 and Section 2.2, the Bayesian optimisation technique is good at suggesting optimum points efficiently for black-functions that are expensive to evaluate, yet there are no formal guarantees on the suggested points being the real optimum or even near the real optimum, while the SMT solver can provide a formal verification by solving the satisfiability problem of SMT formulas, despite the high computational costs in both time and space. The $GEAROPT_{\delta} - BO$ algorithm combines the two techniques together, maintaining their strengths while complementing each other's limitations, and thus can satisfy the requirements listed at the beginning[4]:

- The feasible running time is achieved by delegating the points search part to the fast and inexpensive Bayesian optimisation process and the formal verification part to the SMT solver.
- The safety requirement can be satisfied by encoding the constraints, such as system specifications and restrictions on input features, into the SMT solver.
- The accuracy is ensured by precisely reasoning the function value over the whole input space, utilising the SMT component.
- The stability can be verified by asking the SMT solver whether any point in the region defined by the perturbation of the proposed optimum point violates either the safety or the accuracy properties.

3.1.2 Components

The algorithm tries to find the maximised threshold value T^* and its corresponding safe region. However, this aim is unrealistic because the input space is continuous (for regression tasks), and even a small interval is infinitely divisible. If such an approach is taken, the algorithm would run forever to try to find such a threshold, yet there will always be a threshold that is slightly better than the current one, until maybe it reaches the max accuracy level of the hardware. Since the optimum threshold value has to be an interval rather than a single value, the desired level of accuracy for the interval should be defined first.

To do this, the algorithm first defines a lower bound l and upper bound u which are initialised to $-\infty$ and $+\infty$ respectively, and computes the threshold value by taking the middle of its lower and upper bounds. One thing to notice is that this does not mean to calculate $(-\infty + \infty)/2$ at the beginning. Instead, arbitrary finite lower and upper bounds l_0 and u_0 satisfying $l_0 < u_0$ are used to compute T at first, and l or u is updated if a lower or upper bound has been found and verified following the execution of the algorithm[4]. As the algorithm runs, the lower and upper bounds will keep getting closer to each other and a satisfying interval or solution is obtained if the distance between them is less then an user-specified value, ε. Such an interval is then called a ε-solution and the ideal threshold value T^* has been proved to be bounded by $[l,\ l+\varepsilon)$[4].

With the specifications of solution, the general procedural of the algorithm can be described as an iteration over the two main steps[4]:

- First, try to find a potential candidate point whose value is greater than or equal to the current threshold value. If such a point cannot be found, then the current threshold is set too high, thus set the upper bound u to this value and compute a smaller new threshold: $T_{new} = (l + u_{new})/2$, and execute this step again. Otherwise, take this candidate point as input and perform the next step.

- Given a candidate point, try to find a counter-example inside the region that is around this candidate point and defined by user-specified perturbation or radius in each input dimension of the point, whose value is less than the current threshold. If such a counter-example cannot be found, then the candidate point indeed defines a region in which the function value is greater than or equal to the current threshold everywhere. Record this point and its corresponding threshold value. Next, try a larger threshold value by setting the lower bound to the current threshold and computing: $T_{new} = (l_{new} + u)/2$, and start from the first step. Otherwise a counter-example does exist, meaning that the proposed candidate point is not a real candidate, then return to the first step with the unchanged threshold value and the region around this counter-example being excluded from the search space.

The algorithm terminates if a real candidate point, $\mathbf{x}^*$, is found with lower and upper bound satisfying: $u - l < \varepsilon$, representing a satisfying solution with accuracy ε that defines a safe region centered at $\mathbf{x}^*$ in which all values are no less than the near-optimum value l, as the ideal threshold T^* is bounded in $[l,\ l+\varepsilon)$.

Both of the two steps use a reasoning block consisting of a Bayesian optimiser and a SMT solver[4]:

- The Bayesian optimiser is implemented using the $Optimizer$ class in the SKOPT package[29]. The $Optimizer$ runs a Bayesian optimisation loop, searching for minimum within user-specified input space and dimensions. It has built-in functions $tell()$ for observations and updates on posterior distribution and $ask()$ for point suggestion, and the acquisition function it uses in this project is a hybrid one, *gp_hedge*, which probabilistically selects one heuristic to use among Probability of Improvement (PI), Expected Improvement (EI) and Lower Confidence Bound (LCB) at every iteration;

- The SMT part uses the Z3 solver, developed by Microsoft Research, which is a state of the art SMT prover that supports various logic theories and programming interfaces[22].

3.1.2.1 Searching for Counter-Examples

This part of the algorithm assumes a candidate point has been proposed and tries to find a counter-example around it whose value is less than the current threshold, thus a Bayesian optimiser for minimisation is suitable here. To make fully use of the ability of efficient optimum points suggestion, the Bayesian optimiser is first used to make suggestions up to a max number of times that is set by the user. If any point found during the process evaluates to a value smaller than the threshold T, then the whole operation stops and returns this point as the counter-example.

Otherwise such a point still cannot be found after a max number of attempts using Bayesian minimisation, the job is then delegated to the SMT component, which solves the problem in a much stricter fashion – as formal satisfiability verification, despite a high resulting computational cost. The SMT solver then either provides a satisfying assignment for each dimension in the input space as the counter-example, or returns unsatisfiable if it cannot find a model solution.

The complete steps are as follows:

1. Define the search space around the input candidate point for both the Bayesian optimiser and the SMT solver, using the max allowed perturbation in each feature dimension pre-specified by the user. Given a candidate $\mathbf{x} = [x_1\ x_2\ ...\ x_n]$ of n dimensions and a perturbation vector $\mathbf{r} = [r_1\ r_2\ ...\ r_n]$, the region is then restricted as:
$$\mathbf{R} = [\ [x_1 - r_1,\ x_1 + r_1]\ [x_2 - r_2,\ x_2 + r_2]\ ...\ [x_n - r_n,\ x_n + r_n]\ ],$$
in which each entry represents the allowed interval for a point to be seen as inside the region with respect to the dimension corresponding to the index of the entry. Here, only the l_1-norm is used to measure distance, as it has the lowest computational cost among other norms. Thus, a point $\mathbf{x}' = [x'_1\ x'_2\ ...\ x'_n]$ is considered as a potential counter-example in the valid search space if and only if it

satisfies:

$$\bigwedge_{i=1}^{n} x_i - r_i \leq x'_i \leq x_i + r_i$$

For the BO component, **R** can be directly passed into SKOPT.*Optimizer* to define the search space. For the SMT component, this is achieved by adding extra constraints on the model assignment in each dimension as inequalities into the Z3 prover.

2. Initialise the Bayesian optimiser with a batch of sample points selected based on some heuristic before utilising it for points suggestion, as the Bayesian optimiser without any observations almost always performs poorly and makes bad predictions. Given a user-defined number, n, sample points to be selected and used to warm up the optimiser, there are many heuristics to choose, such as random sampling which randomly picks n points from the input space, or grid sampling which divides the space into n parts and samples once in each sub-region. In my implementation, the Latin Hypercube Sampling (LHS) proposed in [21] is used, which can generate sample points that are evenly distributed across all intervals of the multidimensional input space so that the Bayesian optimiser can see a relatively more general picture of the objective function with those n samples.

3. The warmed-up Bayesian optimiser can then propose promising minimum points, whose value will be evaluated by passing itself into the trained machine learning models, and compared with the threshold T. If the value is less than T, output the point as counter-example. Otherwise, input this point together with its evaluation value as a new observation into the Bayesian optimiser and let it make the next suggestion. After this process iterating over a specific number of times, the Bayesian optimiser gives up and passes the job to the SMT solver.

4. The Z3 prover then tries to solve the problem and either outputs a model assignment as counter-example or returns unsatisfiable.

Moreover, the Bayesian optimiser used in this part is newly created and initialised every time the counter-example searching is called, and the constraints (regrading the search space) that have been encoded into the Z3 porver include only the original domain and the perturbation interval around the candidate point, with respect to each feature dimension. This will not be the case in the candidate searching part.

3.1.2.2 Searching for Candidates

This part also consists of a Bayesian optimiser and a SMT solver. What is different from the counter-example searching process is that:

- The goal is to find a candidate point whose value is greater than or equal to a given threshold T, thus a Bayesian optimiser for maximisation is desired here. However, the SKOPT.*Optimizer* is for minimisation by default and cannot be modified[29]. Since maximising a function is equivalent to minimising its negation, an alternative way to obtain a Bayesian optimiser for maximisation is to use the SKOPT.*Optimizer* on the negation form of the objective function, which can be achieved by letting the *Optimizer* witness $(\mathbf{x_i}, -f(\mathbf{x_i}))$ instead of $(\mathbf{x_i}, f(\mathbf{x_i}))$, for any observed point $(\mathbf{x_i}, f(\mathbf{x_i}))$;

- The SMT solver also constructs constraints in a different manner, which will be discussed later.

Recall the general procedural of the algorithm, in which the threshold T is explored following a binary search manner using the lower and upper bound. For each threshold value, the algorithm decides whether this value is a tighter lower bound or a tighter upper bound for the real optimum T^*, then updates the old lower or upper bound and computes a new threshold accordingly. The Bayesian optimiser (for maximisation) and the SMT solver in this part are globally used during the whole reasoning process of a specific threshold T, while those in the previous part are only defined with respect to a candidate point.

One thing important in the algorithm is how to learn from previous found counter-examples. Given a particular threshold T, the algorithm would iterate over candidate searching and counter-example searching until:

- Either the former returns unsatisfiable, meaning T a tighter upper bound;

- Or the latter returns unsatisfiable, meaning T a tighter lower bound.

Thus, the occurrence of the (i+1)-th iteration means that the candidate searching procedural at the i-th iteration proposed a bad candidate point as a counter-example around it was found. To learn from these counter-examples and increase the quality of candidate points proposed at future iterations, an intuitive approach is to simply exclude the regions centered at those counter-examples from the search space. This is easy to achieve for the SMT solver. For a counter-example point $\mathbf{d} = [d_1\ d_2\ ...\ d_n]$ with perturbation vector $\mathbf{r} = [r_1\ r_2\ ...\ r_n]$, and an arbitrary point $\mathbf{x} = [x_1\ x_2\ ...\ x_n]$ in the input space, the constraint that the search space should be inside the region around this counter-example can be represented as a conjunction:

$$\bigwedge_{i=1}^{n}(d_i - r_i \leq x_i) \quad \wedge \quad \bigwedge_{i=1}^{n}(d_i + r_i \geq x_i)$$

Its negation, representing the constraint that the search space should exclude the region around the counter-example, is expressed as a disjunction:

$$\bigvee_{i=1}^{n}(d_i - r_i > x_i) \quad \vee \quad \bigvee_{i=1}^{n}(d_i + r_i < x_i),$$

which can easily be encoded into the Z3 prover.

However, things get tricky for the Bayesian optimiser. Since the Gaussian process regression (GPR) requires the input space to be continuous[12], the region around a counter-example cannot be directly excluded from the features domain. An alternative way is to penalise the bad candidate point (around which counter-example is found) by forcing the Bayesian optimiser to treat this candidate as the worst (i.e. the smallest) point among all known counter-examples around it[4]:

- The algorithm maintains a list of counter-example points found previously, with respect to various candidate points and under different threshold values during the execution.

- Each time the Bayesian maximiser makes a suggestion, the point is first compared against elements in the counter-examples list before being evaluated. If there are counter-examples around the suggested point that also have values less the current threshold T being reasoned, then the value of this suggested point is set to the smallest value among all its known counter-examples and observed by the Bayesian maximiser, which then makes the next suggestion. This step avoids meaningless evaluation, as point having counter-example(s) nearby cannot be a valid candidate no matter how large an evaluation value it has.

- Only if no known counter-examples nearby can be found will the suggested point be evaluated. If the value is no less than T, return it as a candidate point. Otherwise the point itself is a counter-example with respect to the current threshold T, then append it to the list and record an observation in the Bayesian maximiser.

Additionally, the initialisation of the Bayesian optimiser is also different from that in the previous part. Since the algorithm maintains a list of counter-examples found so far and also records the verified real candidates (obtained by the counter-example searching operation returning unsatisfiable) for previous reasoned thresholds, the potentially helpful information they stored is then used to warm up the Bayesian optimiser at the start of the reasoning process of a particular threshold T[4].

Apart from the goal and the specifications on its Bayesian optimiser and SMT solver, the candidate searching part runs similarly to the counter-example searching part.

3.1.2.3 The Whole Picture

With the help of the two key reasoning blocks, the algorithm is now able to compute a safe, accurate and stable optimal region for a function by iterating over these two blocks to gradually tighten the bound of the real optimum until it satisfies the user, while fully utilising the Bayesian optimisation and the SMT solver techniques to keep the computational complexity within a reasonable range.

The Algorithm 1 below shows a simplified wordy version of the pseudocode of the algorithm from[4].

3.1.3 Usage Conditions

There are also preconditions that need to be satisfied for this algorithm to run.

First, the domain of each input feature must be a continuous space, which is required by the Gaussian process regression in the Bayesian optimisation.

Second, the objective function being reasoned must have an equivalent representation in the SMT solver. This means that the evaluation value of any legal input vector should be the same as the model assignment returned by the SMT solver, if all input features have been encoded as equality constraints into the solver.

In the next section, several commonly used machine learning models will be introduced and their unique ways of equivalent SMT representations translation will be discussed.

Algorithm 1 (*$GEAROPT_\delta - BO$*) l_0 and u_0 are initial bounds satisfying $l_0 < u_0$

```
 1: $P \leftarrow \varnothing$                                  ▷ initialise list for known real candidates
 2: $N \leftarrow \varnothing$                                  ▷ initialise list for known counter-examples
 3: $l \leftarrow -\infty$, $u \leftarrow +\infty$
 4: loop                                        ▷ outer loop, searching for $T^*$ with $\varepsilon$-accuracy
 5:     if $u = +\infty$ then
 6:         $(T, u_0) \leftarrow (u_0, 2u_0 - l_0)$
 7:     else if $l = -\infty$ then
 8:         $(T, l_0) \leftarrow (l_0, 2l_0 - u_0)$
 9:     else
10:         $T \leftarrow (l + u)/2$
11:     end if
12:
13:     Initialise Bayesian Optimiser in Candidate Searching with $P$ and $N$
14:     loop                                    ▷ inner loop, analysing a specific threshold $T$
15:         if Candidate Searching returns UNSAT then
16:             $u \leftarrow T$                            ▷ current threshold $T$ is too large
17:             break
18:         end if
19:         if Counter-Example Searching returns UNSAT on proposed point then
20:             $l \leftarrow T$                            ▷ the point is indeed a solution w.r.t $T$
21:             append to $P$; break
22:         end if
23:         append the counter-example found to $N$
24:         BO and SMT in Candidate Searching learn from this counter-example
25:     end loop
26:
27:     if $u - l < \varepsilon$ then                       ▷ check if $\varepsilon$-accuracy satisfied
28:         break
29:     end if
30: end loop
```

3.2 Supported Machine Learning Models

Machine learning models are widely used in real life problems due to their ability to efficiently build a reasonable approximation of the real yet unknown function under the problem. The motivation behind Algorithm 1 is to optimise such models, representing systems that require formal verification on miscellaneous constraints being satisfied, and also have inevitable perturbation in their input space. This has an implicit requirement on the quality of the models, as the verified optimum output would not make any use in practice if the model itself is a bad approximation to the real function. Thus, this section discusses not only the SMT equivalence for different models, but also their general training methods for higher accuracy.

The aim here is to compare different models (polynomial model, neural network and decision tree) between their accuracy and time taken for the algorithm to run, based on such a standard: for each type, build a model that is as accurate as possible and test the algorithm on it.

3.2.1 Polynomial Model

Polynomial model is an important and widely used approach to identify the relationship between the input and output of a dataset using a polynomial equation[24]. By adjusting the number of degree, coefficients and terms included in the equation, polynomial model can capture non-linear and complex patterns in a dataset and fit a wide variety of curves, as shown in Figure 3.1.

The main drawback of the polynomial model is the underfitting or overfitting on test data due to an improper degree being set, which can be eased by selecting the degree with the help of a validation set that is different from the training data.

3.2.1.1 Training and Tuning

The implementation of the polynomial model uses:

- *sklearn.linear_model.LinearRegression*[26], which builds a linear model on the dataset and makes predictions by taking a weighted sum of all input features plus a bias term to reduce overfitting;

- *sklearn.preprocessing.PolynomialFeatures*[27], which generates a list of polynomial features, given the number of dimensions of input features and the max polynomial degree allowed. For instance, a two-dimensional input space $\mathbf{x} =$

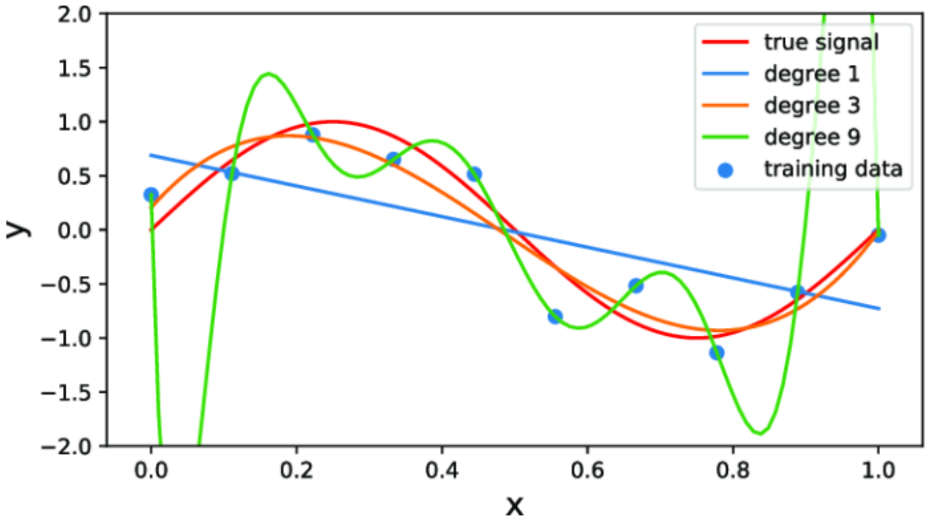


Figure 3.1: Example 1D polynomials with various degrees[1].

$[x_1\ x_2]$ with a desired degree of 2 would produce a polynomial features list: $[1,\ x_1,\ x_2,\ x_1x_2,\ x_1^2,\ x_2^2]$.

The two steps will build a linear regression model on the polynomial features, which can also be seen as the polynomial model with respect to the original input features.

The hyperparameter to be tuned here is mainly the degree of the polynomial, which is selected by following the steps:

1. Given the training dataset, it is first shuffled and a k fold cross validation is performed for each candidate degree within range $[1,15]$, where k is set to 5;

2. Each candidate degree is then trained using 80% of the training data and tested against the rest for five times, and the average root mean squared error (RMSE) is recorded;

3. After all candidates being evaluated, the one with the smallest average RMSE is chosen to build the polynomial.

The final polynomial model is then built on the whole training data using the selected degree, and tested against the test dataset, using the RMSE metric to quantify its performance.

3.2.1.2 SMT Translation

After the polynomial model being trained and tuned, it then needs to be translated into its SMT representation for the algorithm to run. For a problem in n-dimensional input

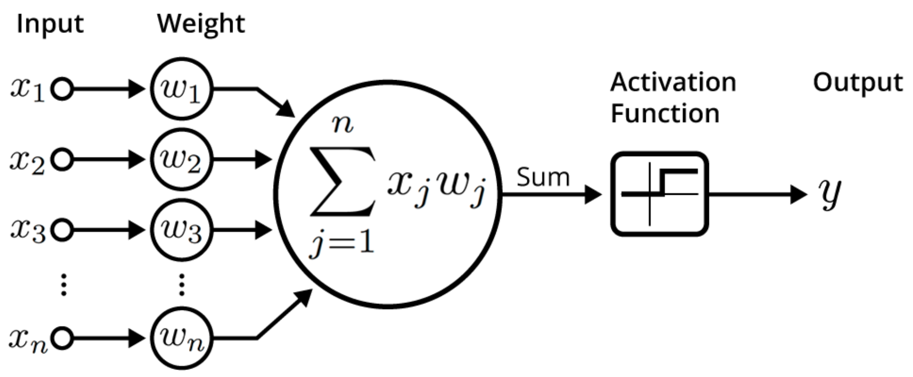


Figure 3.2: Internal structure of a single neuron[20].

space containing features $[x_1\ x_2\ ...\ x_n]$ and a degree of d, any arbitrary i-th term t_i in its polynomial equation can be expressed as: $c_i x_1^{i_1} x_2^{i_2} x_3^{i_3} ... x_n^{i_n}$, satisfying $i_1 + i_2 + ... + i_n \leq d$, where c_i is the trained weight for this term. Given an input vector $\mathbf{x} = [x_1\ x_2\ ...\ x_n]$, a d-degree polynomial model with m terms thus outputs:

$$y = \sum_{i=1}^{m} c_i x_1^{i_1} x_2^{i_2} ... x_n^{i_n}\ ,\ where\ \ i_1 + i_2 + ... + i_n \leq d$$

Since the terms are decided during training and can be extracted together with the corresponding weights from the model, and also the operations used in the equation are just addition and multiplication which are both supported by the theory solvers in Z3, the polynomial model can thus be directly encoded into the SMT solver.

However, since the polynomial model contains multiplications between variables, the computational complexity for the SMT solver is considerably huge, especially when the degree or the number of input dimensions is also high.

3.2.2 Neural Network with Rectified Linear Activation Function

Neural network is one of the most popular machine learning models and is used extensively in many areas, such as image recognition, natural language processing and components placement in chip design, due to its versatility in capturing any complex patterns and relationships under datasets and generalising to new data.

It is inspired by the biological neural networks in animal brains, constructed by the basic units called neurons. In the context of computer science, a neuron can be considered as a multi-input single output linear regression model, whose value can be computed by taking a weighted sum of all its inputs (plus an independent bias term)

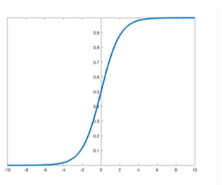
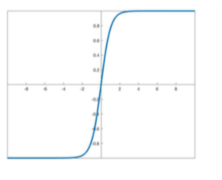
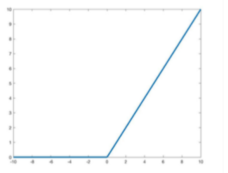
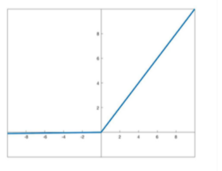
Activation function	**Equation**	**Graph**
Sigmoid	$S(x) = \dfrac{1}{1+e^{-x}}$	
Tanh	$\tanh x = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$	
ReLU	$RELU(x) = \begin{cases} 0 \ if \ x < 0 \\ x \ if \ x >= 0 \end{cases}$	
Leaky ReLU	$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$	

Figure 3.3: Four commonly used activation functions[7].

and then applying an activation function, as shown in Figure 3.2. Activation functions are utilised to introduce non-linearity into the model by deciding how should a neuron be activated, offering neural network the ability to model complex patterns. The commonly used ones are displayed in Figure 3.3, which explains why the activation function of the neural network to be reasoned by the algorithm is specified as ReLu or its variant – the SMT solver does not support non-linear relationships such as *Sigmoid* and *Tanh*.

Such neurons then combine and form a layer. The neural network consists of an input layer, an output layer and various number of hidden layers in between, each neuron in which is decided by all neurons in the previous layer together with the selected activation function, and also contributes to the calculations of all neurons in its subsequent layer, as shown in Figure 3.4.

The rich selection of weights for each neuron, hidden layer structures and activation functions gives neural network high flexibility for different types of date patterns and tasks, despite its opaque nature from a human's point of view and difficulties in training

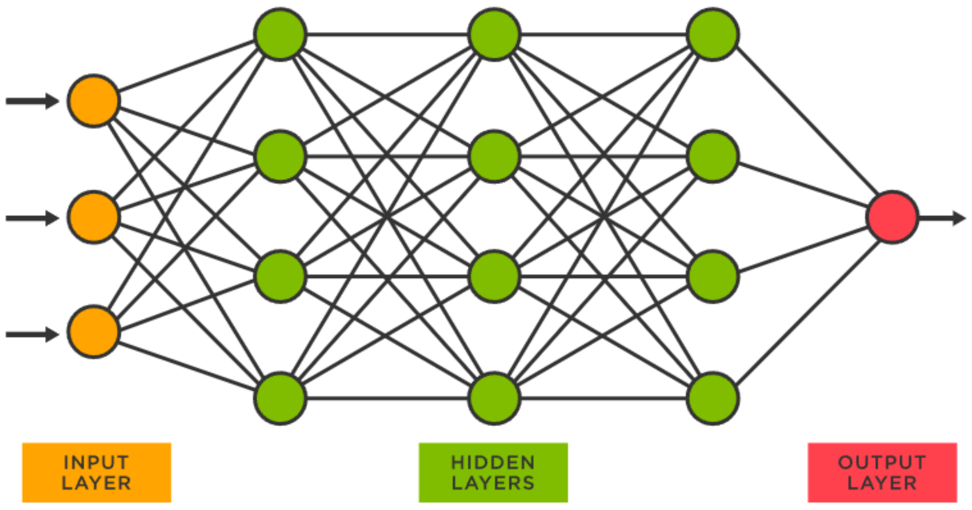


Figure 3.4: A multi-input neural network with single output[33].

due to a large amount of parameters.

3.2.2.1 Training and Tuning

The neural network is implemented and tuned using:

- The *Sequential* class from *Keras*[15], which can stack layers sequentially with specified number of neurons and activation function;
- The neural network is tuned using the *Hyperband* method from *KerasTuner*[23], with the number of hidden layers within $[1,2]$, the number of neurons in each layer ranging from $[8,64]$ and *max_epochs* set to 300 to control complexity.

The trained neural network is then tested using the same test dataset as in polynomial model, and measured using RMSE.

3.2.2.2 SMT Translation

As shown in Figure 3.4, the final output of the neural network is a weighted sum of all neurons in the last hidden layer (activation function typically not applied for final output), each of which is also an activated weighted sum of all the neurons in its previous layer. Thus, given n hidden layers with their corresponding numbers of neurons: $k_1, k_2, \ldots, k_n$ and $h_{i,j}$ representing the j-th neuron in the i-th hidden layer and $w_{i,j,m}$ as its weight for the m-th neuron in the subsequent layer, the neural network model can be recursively written as:

$$
\begin{aligned}
y &= \sum_{j=1}^{k_n} w_{n,j,1}\, h_{n,j} && \text{; 1 refers to the single output neuron} \\
&= \sum_{j=1}^{k_n} w_{n,j,1}\, (\phi \sum_{s=1}^{k_{n-1}} w_{n-1,s,j}\, h_{n-1,s}) && \text{; } \phi \text{ is the selected activation function} \\
&= \sum_{j=1}^{k_n} w_{n,j,1}\, (\phi \sum_{s=1}^{k_{n-1}} w_{n-1,s,j}\, (\phi \sum_{r=1}^{k_{n-2}} w_{n-2,r,s}\, h_{n-2,r})) \\
&= \dots\dots,
\end{aligned}
$$

until the input layer is reached, whose value is known. Since:

- The number of hidden layers, the number of neurons in each hidden layer and its weights can be extracted from the model;
- The linear regression calculation (i.e. the weighted sum) is supported by the Z3 prover;
- The ReLu activation function is equivalent to: $if\ value > 0\ then\ value\ else\ 0$, which can be directly implemented in the solver.

Thus, the SMT representation of a neural network (with ReLu) can be obtained by encoding the equation of every neuron from those in the first hidden layer to the one in the output layer into the solver.

3.2.3 Decision Tree

Decision tree is another popular model used in supervised machine learning tasks, which is renowned for its simplicity to be understood by humans compared to other models as well as its ability to handle complex data patterns, following a top-down method called recursive splitting[18, 34].

A decision tree has a binary tree structure, in which each non-terminal node runs a constraint check on input variables and makes a split accordingly, until a leaf node is reached, representing the output for the given input. The constraints on split nodes and output of leaf nodes can take various forms to fit particular types of tasks. With respect to the project, only decision tree for regression is discussed, in which conditions are inequality and outputs are real numbers, as the example shown in Figure 3.5.

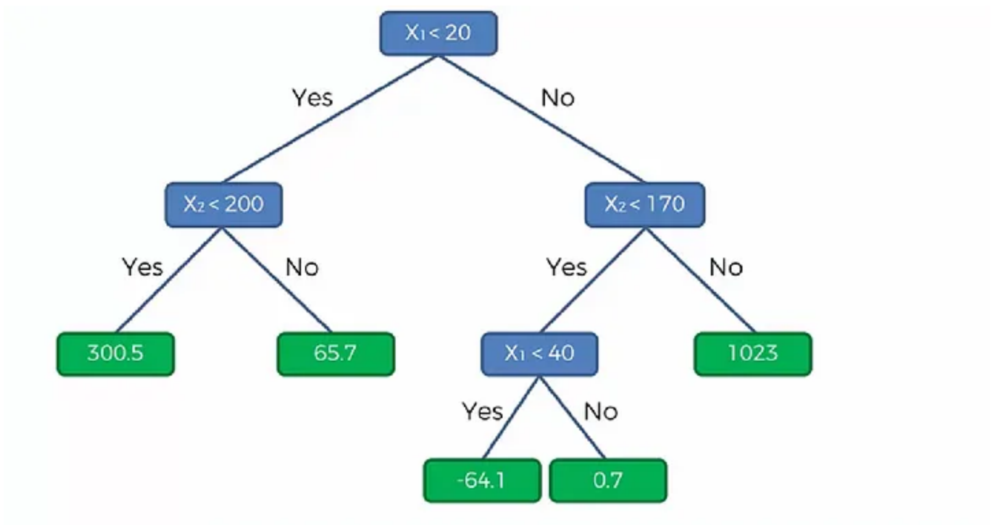


Figure 3.5: A decision tree regressor in 2D space[32].

3.2.3.1 Training and Tuning

The decision tree model is implemented using *sklearn.tree.DecisionTreeRegressor*[28], which offers user the freedom to specify various properties of the tree.

For a decision tree model, the most influential hyperparameters are:

- *max_depth*, which decides how deep the tree can be. The deeper the tree is, the more complex data patterns it is able to capture, despite the risk of overfitting;
- *min_samples_split*, representing the minimum number of samples needed to make a split;
- *min_samples_leaf*, which sets the minimum number of samples needed that have the same value as the leaf node;
- *max_features*, controlling the number of features to be considered when looking for the best split;
- *max_leaf_nodes*, deciding the number of all possible outputs.

These hyperparameters are tuned with the help of *sklearn.model_selection.GridSearchCV*, which selects each hyperparameter from a user-defined list containing their commonly used settings of different scales, then performs a k-fold cross validation and records its performance measured using mean squared error (MSE). After all possible combinations of these parameters have been tried, the model with the lowest MSE is used as the final model.

3.2.3.2 SMT Translation

The SMT representation of decision tree is easy to implement, since the two node types in decision tree can be directly encoded into the SMT solver, with the split node as an *if-then-else* structure and the leaf node as an equation that assigns value to output.

By recursively traversing through the tree (starting from the root node), the whole decision tree can be encoded as a nested *if-then-else* structure into the solver.

3.3 Variants of SMT Representations of Tree Models

Given the same machine learning model, it is obvious that how it is represented in the SMT solver has a huge influence on the algorithm's solving time. This section discusses two ways of translating decision trees into SMT formulas, and tries to compare the resulting difference between their solving times.

3.3.1 Nested *if-then-else*

The nested *if-else* statements shown in Figure 3.6 may be easy and intuitive to be encoded into SMT formulas, but it could add unnecessary complexity to the solving process as the depth of the tree grows, and increase solving difficulties. Thus, an alternative is then presented to make a comparison with this approach.

```
if ... then:
    if ... then:
        ...
    else:
        ...
else:
    if ... then:
        ...
    else:
        ...
```

Figure 3.6: Nested if-then-else structure

3.3.2 Conjunction of Implications

The decision tree can also be seen as a trip from the root node to any of the leaf nodes, between which there are many possible paths to take, determined by conditions check on input variables. A path P_i can then be expressed as an implication:

$$c_1 \wedge c_2 \wedge ... \wedge c_n \ \rightarrow \ y = y_i \ ,$$

where $c_1, c_2, ..., c_n$ are conditions on split nodes required to be satisfied to take path P_i, typically atomic formulas as inequality of input features, and y_i is the output returned by P_i. This is equivalent to a clause, in which literals are either atomic SMT formulas – equality and inequality under the real arithmetic theory or their negations:

$$\neg c_1 \vee \neg c_2 \vee ... \vee \neg c_n \vee y = y_i$$

All possible paths of a decision tree can be found in an iterative manner by running a depth-first search on the root node. Suppose there are m possible paths, then the tree is equivalent to:

$$\bigwedge_{i=1}^{m} P_i$$

which is in a conjunction normal form (CNF). Recall the mechanism behind the SMT solver in Section 2.2, in which the formula is first converted to its CNF equivalence and solved by a DPLL-based SAT solver to propose a potential satisfying assignment, the conversion step can thus be omitted for CNF-represented decision trees, which could potentially improve solver's efficiency.

The comparison is measured in Chapter 4.

3.4 Extension to Other Machine Learning Models

Now that decision tree has its complete SMT representation, other advanced tree-based models can also be encoded into SMT solver and reasoned by the algorithm. This section looks at two machine learning models built on decision trees.

3.4.1 Random Forest

Random forest is a machine learning technique that consists of multiple decision trees, each of which is trained using a randomly selected subset from the training data and

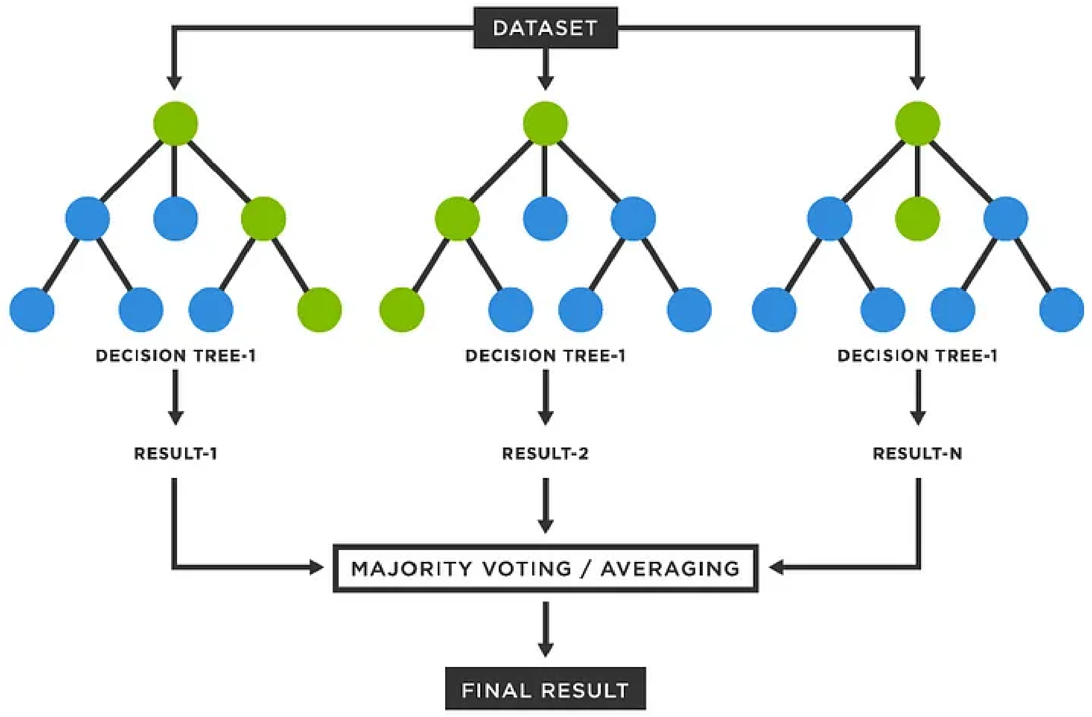


Figure 3.7: Structure of a simple random forest[14].

then contributes to the final output independently based on a heuristic (typically the mean), resulting in a system that is robust to overfitting and has generally good accuracy[5], as shown in Figure 3.7.

For regression tasks, the final output of a random forest is mostly calculated by taking:

- the mean, which can be easily implemented in SMT solver;
- the weighted mean, easy to implement (as linear regression) in SMT, despite additional complexity introduced for weights training;
- the trimmed or truncated mean that ignores trees of extreme values, requiring sorting among all trees, which is also supported by most SMT solvers.

Thus, the algorithm is also able to reason systems represented using random forests.

3.4.2 Boosted Tree

Boosted tree is another technique that is built with individual trees. Unlike random forest that builds each tree independently using a subset from training data, individual

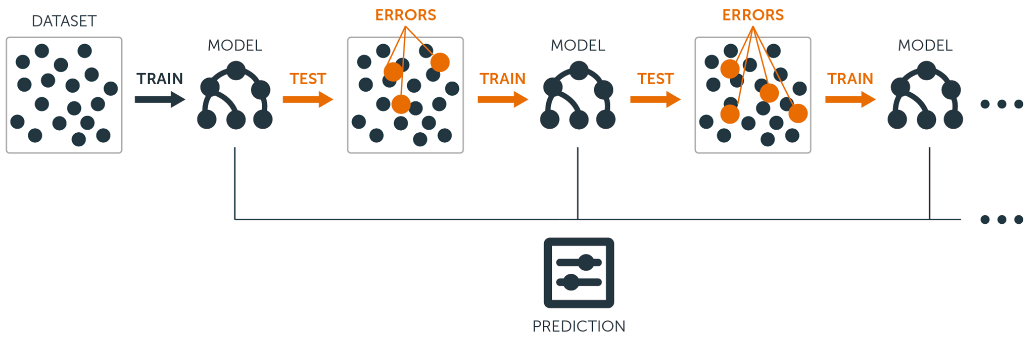


Figure 3.8: Internal structure of boosted trees[19].

trees in boosted tree are sequentially connected as shown in Figure 3.8, each learning from the errors of the previous ones, trying to reduce overall errors in the speed of a specified learning rate and both contributing to the final output, which makes boosted trees good at reducing bias and variance in predictions[6].

The main complexity here is about model training and tuning, while the SMT translation of boosted trees is straightforward and similar to that of random forests – as linear combination of values of individual decision trees.

The usage extension to other machine learning models enables the optimisation algorithm to reason a wider range of systems.

Chapter 4

Evaluation

This chapter presents the testing and evaluation part of the project:

In Section 4.1, multiple datasets with various number of feature dimensions have been used to train polynomial models, neural networks with ReLu and decision trees, and comparison has been made between their prediction accuracy and algorigthm solving time.

In Section 4.2, datasets of various feature dimensions have been used to train a decision tree model. Given the same tree model, two ways of SMT translations have been applied and their corresponding solving times have been recorded as the complexity of tree increased.

4.1 Comparison between Different Models

Both of the three models were trained and tested using the same training and testing dataset, respectively. The prediction accuracy is measured using Normalised Root Mean Square Error (NRMSE) as a scale-invariant error metric:

$$NRMSE = \frac{RMSE}{y_{max} - y_{min}}\ ,\ \ where\ RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

Three trials were performed, NRMSE and time spent (in minutes) of each trial were recorded and then averaged to represent the overall performance.

	t_1	t_2	t_3	t_{avg}	$NRMSE_{avg}$
Polynomial	0.83	0.64	0.73	0.73	0.24%
Neural Network	3.01	4.36	4.04	3.80	1.50%
Decision Tree	0.73	0.68	0.72	0.71	0.83%

Table 4.1: Results for 1D dataset

	t_1	t_2	t_3	t_{avg}	$NRMSE_{avg}$
Polynomial	timed out	timed out	timed out	timed out	N.A.
Neural Network	timed out	49.73	129.67	89.7	6.37%
Decision Tree	1.07	1.08	1.03	1.06	4.89%

Table 4.2: Results for 2D dataset

4.1.1 One-Dimensional Dataset

This dataset was obtained by randomly sampling 2000 (80% for training) points from function $y = -x_1^4 + 10x_1^3 - 25x_1^2 - 5x_1 + 10$, and all input features were then normalised to $[-1, 1]$ and output normalised to $[0, 1]$ using *minmax*. Each feature had a 5% perturbation (± 0.1) and ε-accuracy was set to 5% (0.05).

Results are shown in Table 4.1.

4.1.2 Two-Dimensional Dataset

The dataset was sampled from $y = -x_1^4 - x_2^4 + 10x_1^2 + 10x_2^2$, while the other configurations were unchanged. Results are shown in Table 4.2.

4.1.3 Four-Dimensional Dataset

The dataset was sampled from $y = -0.1x_1^4 + 0.5x_1^3 - x_2^2 + x_2x_4^2 + x_3^2 - x_3x_4^3 + 0.1x_4^4 - x_1x_2 + x_2x_3 - x_3x_4 + x_1x_4 + 150$ with the rest unchanged. Results shown in Table 4.3.

	t_1	t_2	t_3	t_{avg}	$NRMSE_{avg}$
Polynomial	timed out	timed out	timed out	timed out	N.A.
Neural Network	timed out	timed out	timed out	timed out	N.A.
Decision Tree	1.11	1.04	1.08	1.08	5.22%

Table 4.3: Results for 4D dataset

4.2 Comparison between Different SMT representations of Decision Trees

Two representations of SMT equivalence for decision trees discussed in Section 3.3 were implemented and tested by applying the algorithm on various depths of trees and on various-dimensional datasets.

Since the objective here was only to compare solving times for different SMT representations and accuracy was not concerned, thus the decision tree is trained casually and all hyperparameters were set to the values for resulting in deeper trees.

4.2.1 One-Dimensional Dataset

The same dataset was used and only the *max_depth* parameter was changed during decision trees creations, ranging from: $[1,2,5,10,15,20,25,30,35,40,45,50]$. The time spent for the two trees using different SMT representations with respect to a specific depth was recorded and plotted in Figure 4.1.

4.2.2 Two-Dimensional Dataset

Same settings were used and the results were compared in Figure 4.2.

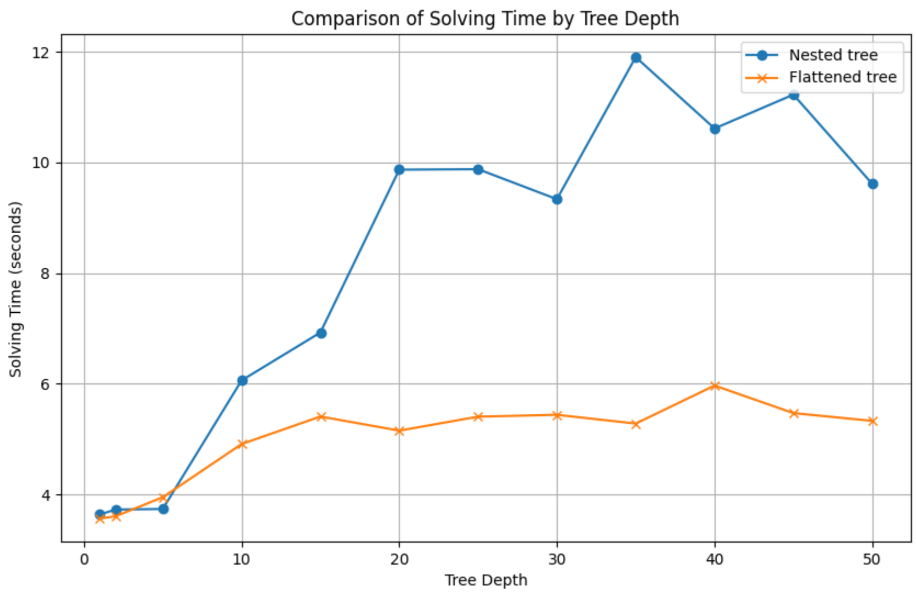


Figure 4.1: Comparison on solving time using 1D dataset.

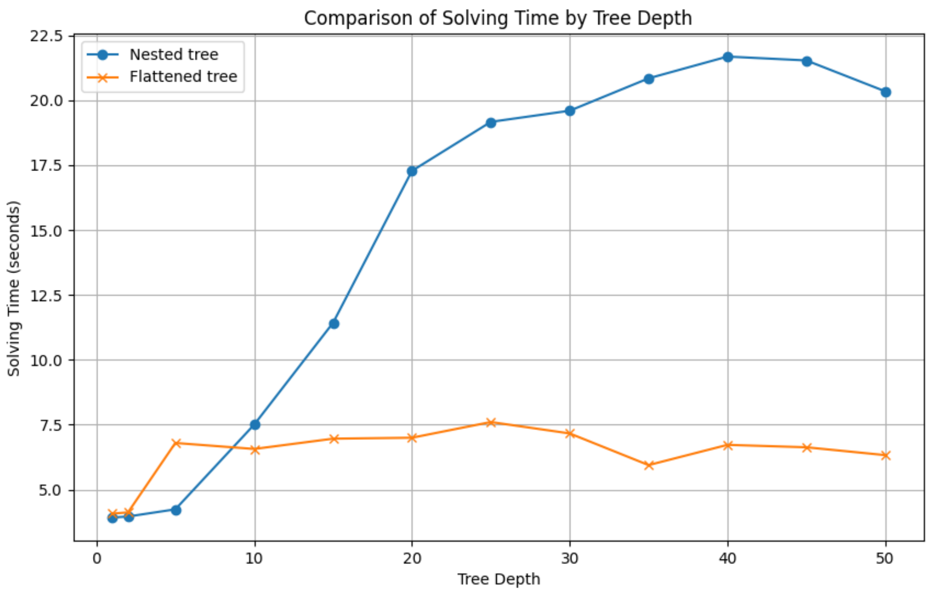


Figure 4.2: Comparison on solving time using 2D dataset.

Chapter 5

Conclusion

This chapter analyses the results from Chapter 4 and discusses limitations and future improvements of the project.

5.1 Results

As shown in Section 4.1, the polynomial model can only terminate in the one-dimensional dataset and always timed out in multi-dimensional datasets. This is understandable as polynomial models for multi-dimensional inputs introduce multiplications between variables and even powers of variables, which would significantly enlarge the searching space and increase huge complexity.

However, the several timed-outs for neural networks are confusing, since it only introduce weighted sum of variables into the SMT solver yet had such a huge influence on the algorithm's reasoning time. This may be caused by the inefficient SMT representations of neural networks discussed in Section 3.2 and deserved further exploration.

Moreover, decision tree is the fastest model for the algorithm to run, while maintaining a good balance in accuracy and solving time. This is because it does not do any calculations in the SMT solver, only values comparison or assignment.

As for the comparison between different SMT representations of decision trees shown in Section 4.2, the tree using flattened path conjunction shows an invariant property to the change of tree depths while the solving time for trees represented as nested *if-then-else* increases as the tree grows deeper.

5.2 Limitations

There several main limitations for this project:

- Only single-out regression task is considered, despite the possibility to the extend the algorithm into classification problems;
- Few datasets are used for restricting the running time of the algorithm within a short range, making the observations and conclusions less representative;
- The hyperparameter tuning strategies chosen for each models are rather simple, which may limit their potential in prediction accuracy.

5.3 Future Improvements

Potential improvements that can be done in the future for the algorithm may include:

- Try to support more machine learning models, as long as they can be efficiently encoded into the SMT solver;
- Try to use dynamic *Max_iteration* values in Bayesian optimiser instead of hard coded values, which could waste time on hopeless searching if many counter-examples have been excluded from the search range and the BO is unable to propose a candidate;
- Try different heuristics for BO's initialisation.

Bibliography

[1] M. J. Bianco, P. Gerstoft, J. Traer, E. Ozanich, M. A. Roch, S. Gannot, and C.-A. Deledalle. Machine learning in acoustics: Theory and applications. *The Journal of the Acoustical Society of America*, 146(5):3590–3628, Nov. 2019.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsch. *Handbook of Satisfiability*, chapter 12. IOS Press, 2008.

[3] F. Brauße, Z. Khasidashvili, and K. Korovin. Selecting stable safe configurations for systems modelled by neural networks with relu activation. In *Formal Methods in Computer-Aided Design*, 2020.

[4] F. Brauße, Z. Khasidashvili, and K. Korovin. Combining constraint solving and bayesian techniques for system optimization. In *the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, 2022.

[5] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 10 2001.

[6] Y. Coadou. *Boosted Decision Trees*, page 9–58. WORLD SCIENTIFIC, Feb. 2022.

[7] Datawow. *Interns Explain Basic Neural Network.* Available: `https://www.datawow.io/blogs/interns-explain-basic-neural-network`. Accessed: 5-Apr-2024.

[8] L. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[9] L. de Moura, B. Dutertre, and N. Shankar. A tutorial on satisfiability modulo theories. In *Computer Aided Verification*, pages 20–36. Springer Berlin Heidelberg, 2007.

[10] Dive into Deep Learning. *Introduction to Gaussian Processes*. Available: https://d2l.ai/chapter_gaussian-processes/gp-intro.html. Accessed: 15-Mar-2024.

[11] D. K. Duvenaud. *Automatic Model Construction with Gaussian Processes*. PhD thesis, University of Cambridge, 2014.

[12] P. I. Frazier. A tutorial on bayesian optimization, 2018.

[13] W. Gan, Z. Ji, and Y. Liang. Acquisition functions in bayesian optimization. In *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*, pages 129–135, 2021.

[14] D. Gunay. *Random Forest*. Available: https://medium.com/@denizgunay/random-forest-af5bde5d7e1e. Accessed: 5-Apr-2024.

[15] Keras 3 API documentation. *The Sequential class*. Available: https://keras.io/api/models/sequential/. Accessed: 15-Mar-2024.

[16] G. Kremer. *Satisfiability Modulo Theories: Using OS to solve hard problems*. Available: https://media.ccc.de/v/froscon2023-2873-satisfiability_modulo_theories#t=1. Accessed: 15-Mar-2024.

[17] H. J. Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86:97–106, 1964.

[18] Y. Lu, T. Ye, and J. Zheng. Decision tree algorithm in machine learning. In *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*, pages 1014–1017, 2022.

[19] mariajesusbigml. *Introduction to Boosted Trees*. Available: https://blog.bigml.com/2017/03/14/introduction-to-boosted-trees/. Accessed: 5-Apr-2024.

[20] N. McCullum. *Deep Learning Neural Networks Explained in Plain English*. Available: https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/. Accessed: 5-Apr-2024.

[21] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[22] Microsoft Research. *The Z3 Theorem Prover*. Available: `https://github.com/Z3Prover/z3`. Accessed: 15-Mar-2024.

[23] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, et al. Kerastuner. `https://github.com/keras-team/keras-tuner`, 2019.

[24] E. Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48:500–506, 2012. Modelling of Mechanical and Mechatronics Systems.

[25] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. the MIT Press, 2006.

[26] Scikit-learn Documentation (Ver.1.4.2). *sklearn.linear_model.LinearRegression*. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html`. Accessed: 15-Mar-2024.

[27] Scikit-learn Documentation (Ver.1.4.2). *sklearn.preprocessing.PolynomialFeatures*. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html`. Accessed: 15-Mar-2024.

[28] Scikit-learn Documentation (Ver.1.4.2). *sklearn.tree.DecisionTreeRegressor*. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`. Accessed: 15-Mar-2024.

[29] Scikit-Optimize Documentation (Ver.0.8.1). *skopt.Optimizer*. Available: `https://scikit-optimize.github.io/stable/modules/generated/skopt.Optimizer.html`. Accessed: 15-Mar-2024.

[30] H. Song, I. Triguero, and E. Özcan. A review on the self and dual interactions between machine learning and optimisation. *Progress in Artificial Intelligence*, 8(2):143–165, 6 2019.

[31] X. Song, E. Manino, L. Sena, E. Alves, E. de Lima Filho, I. Bessa, M. Lujan, and L. Cordeiro. Qnnverifier: A tool for verifying neural networks using smt-based model checking, 2021.

[32] M. Soni. *What is Decision Tree Regression?* Available: `https://maniksonituts.medium.com/what-is-decision-tree-regression-dcd0ea40a323`. Accessed: 5-Apr-2024.

[33] Spotfire. *What is a neural network?* Available: `https://www.spotfire.com/glossary/what-is-a-neural-network`. Accessed: 5-Apr-2024.

[34] B. Talekar. A detailed review on decision tree and random forest. *Bioscience Biotechnology Research Communications*, 13:245–248, 12 2020.

[35] J. Wang. An intuitive tutorial to gaussian process regression. *Computing in Science & Engineering*, 25(4):4–11, July 2023.

[36] W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson. Verification for machine learning, autonomy, and neural networks survey, 2018.