Jonas Tørnes

Machine learning applied to GPU test generation

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Bjørn B. Larsen
Co-supervisor: Øystein Gjermundnes
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Jonas Tørnes

Machine learning applied to GPU test generation

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Bjørn B. Larsen
Co-supervisor: Øystein Gjermundnes
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Preface

This master's thesis has been written as the final assignment for the five-year master's program in Electronics Systems Design and Innovation at the Norwegian University of Science and Technology. It is submitted as the final delivery in the course "TTFE4940 1 Electronics Systems Design and Innovation, Master's thesis". It has been supervised by Prof. Bjørn B. Larsen from the Department of Electronic Systems and DEng. Øystein Gjermundnes at Arm Norway AS. All the work in this thesis has been carried out during the spring semester of 2023. It is a spiritual continuation of the specialization project conducted during the autumn of 2022 [1], where the problem was the same, but the solution proposed was different.

Acknowledgements

The work with this thesis has had its ups and downs, frustrating and rewarding simultaneously. But also, never have I learned so much from a project; it has been exciting to take a deep dive into machine learning and try and use it for something useful and new. Spending the last year thinking about this problem has been a privilege, and it is with a bit of sadness that the journey is finally over. I would first like to express my gratitude to my supervisors throughout this last year. Thank you, Bjørn, for our bi-weekly meetings and for listening and allowing me to share my thoughts and frustrations. An Øystein, you have constantly been encouraging and positive through this project, even when facing new problems or things did not go according to the plan. You have also always pushed me to write a better thesis; your input has been invaluable. I would also like to express my gratitude to Wade Walker for creating the idea which resulted in this thesis. Zhirong Yang from the Department of Computer Science has also been an enormous resource for anything machine learning related and helped by proposing the initial solution.

Five years is quite a long time, or at least I thought it would be that, but it has passed too quickly. That would not have been the case had it not been for all the amazing people I have met through these five years here in Trondheim; I am very grateful for that. Finally, I want to thank my family, Mom, Dad, and my sister Anne for supporting me through these five years and the 18 years prior. I could not have done this without you.

Abstract

This thesis has investigated how to produce and find efficient stimuli for simulation-based verification of digital circuits. The current state-of-the-art verification still relies heavily on random constrained verification and human involvement in the verification process. This results in tedious work or the need to simulate many tests to ensure good design coverage; this process is time-consuming and costly. Currently, very few methods exist, allowing this process to be optimized with machine learning.

This thesis proposes and implements a complete system for optimizing random-constrained verification. It utilizes Bayesian optimization to optimize the weights assigned inside the testbench. Because of the current limitations of Bayesian optimization, we have also implemented a way to scale the maximum number of weights the system can control. The system is tested on two devices to verify its capabilities. Both absolute coverage numbers and compute costs have been measured during testing for different scenarios and configurations.

Results show that the method generally performed as well as random verification on our test devices. However, it can be used with human input and produce a considerable increase over random while also doing slightly better than constrained random. It is just as efficient as random verification when not using any form of scaling to allow the system to control more weights.

The key finding of this thesis is that a machine learning system based on Bayesian optimization can improve over random verification. However, some downsides remain, such as not performing better than random without additional input or when used with the proposed scaling solution to allow for control of additional weights functions badly. The limited testing on two devices shows that improvement can be device-specific, depending on functionality. It can aid the verification process, yielding a slight coverage increase in its current form.

// Sammendrag

Denne oppgaven har undersøkt hvordan finne effektiv tester og stimuli for simulerings basert verifikasjons av digital kretser. Nåværende metoder er baseres på avgrenset tilfeldig simulering og menneskelig involvering. Dette resultere in tidkrevende og arbeids krevende prosess for å forsikre seg om at hele kretsen er testet. Det mangler vel etablerte metoder for å optimalisere denne prosessen med maskinlæring.

Denne oppgaven foreslår en ny metode for å optimalisere avgrenset tilfeldig simulering. Dette systemet er bygget på Bayesiansk optimalisering som skal optimalisere vekter inne i testbenken. På grunn av de nåværende begrensingen til Bayesiansk optimalisering har vi også laget en måte å skalere antall vekter systemet kan kontrollere. Systemet har blitt testet på to ulike enheter for ulike scenarier.

Resultatene viser at metoden ikke hadde noen store forbedringer over tilfeldig testing på de to enhetene når den er brukt fritt stående. Når brukt sammen med et menneske viste systemet en betydelig forbedring over tilfeldig stimuli og var også bedre en avgrenset tilfeldig simulering. Når ingen metode for skalering av antall vekter var brukt viste systemet at det ikke økte kjøretiden sammenlignet med tilfeldig stimuli.

Testingen viser at maskinlæring systemet basert på Bayesiansk optimalisering kan gi en forbedring over tilfeldig verifikasjon. Det finnes imidlertid ulemper slik som at det ikke gir noe forbedring over tilfeldig verifikasjon uten ekstra menneskelig hjelp. Resultater ved bruke av skalerings metoder viser at disse ikke fungere og gir like resultat med tilfeldig verifikasjon. Testingen på de to enheten viser at forbedringer kan varier fra enhet til enhet avhengig av funksjonaliteten til enheten. Uten videre utvikling av systemet kan det bare brukes som et hjelpe middel for en verifikasjon ingeniør.

TABLE OF CONTENTS

Table of Contents iv

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Report Outline 2

2 Problem Description 3

3 Theory 4

3.1 Bayesian statistics 4

3.2 Bayesian optimization 5

3.3 Multivariate normal distribution 6

3.4 Properties of the Gaussian distribution 7

3.5 Gaussian process 7

3.5.1 Covariance plot 9

3.5.2 Kernels 10

3.5.3 A practical example of Gaussian process 11

3.5.4 Acquisition function 13

3.6 A practical example of Bayesian optimization 14

3.7 Bayesian optimization in high dimensions 17

3.8 Preceptron 17

3.8.1 Curse of dimensionality 18

3.9 Coverage 19

3.9.1 Code coverage 19

4 Implementation **22**

4.1 Device under test (DUT) 22

4.1.1 Device A 22

4.1.2 Device B 23

4.2 SystemVerilog Testbench 23

4.2.1 Transaction item 23

4.2.2 Interface 24

4.2.3 Driver 25

4.2.4 Monitor 25

4.2.5 Environment 25

4.2.6 Test 25

4.2.7 Top-level testbench 26

4.2.8 Testbench for Device A 26

4.2.9 Testbench 1 for Device B 27

4.2.10 Testbench 2 for Device B 27

4.3 Machine learning framework 28

4.3.1 Training loop 28

4.3.2 Scikit-Optimize 29

4.3.3 Weight preprocessor 29

4.3.4 TensorFlow 30

4.3.5 Coverage converter 30

4.3.6 Training loop for device A 30

4.3.7 Training loop for device B testbench 1 31

4.3.8 Training loop for device B testbench 2 31

4.3.9 Inference 33

5 Results **34**

5.1 Testing methodology 34

5.1.1 Welch's t-test 35

5.1.2 Normally distributed 35

5.2 Determining the number of tests 39

5.2.1 Device A 39

5.2.2 Device B 40

5.3 Parameter tuning 41

5.3.1 How to read the results . . . 41
5.3.2 Device A . . . 42
5.3.3 Device B . . . 44
5.4 Comparison with other methods . . . 47
5.4.1 Device A . . . 47
5.4.2 Device B . . . 49
5.5 Computational cost . . . 51
5.5.1 Total Compute Time (TCT) . . . 51
5.5.2 Equivalent Compute Time (ECT) . . . 52
5.6 Scaling . . . 54
5.6.1 Device A . . . 54
5.6.2 Device B . . . 55

6 Discusion **57**
6.1 Coverage results . . . 57
6.2 Computational cost . . . 58
6.3 Problems with projection . . . 58
6.4 Golden coverage . . . 59
6.5 Limitation of the testing . . . 59

7 Future work **61**

8 Conclusion **62**

Bibliography **63**

LIST OF FIGURES

3.1 A covariance plot between two variables y_1 and y_2. 9

3.2 The resulting Gaussian distribution when conditioning on $y_1 = 2$. 10

3.3 Plot of a selection of kernels. 11

3.4 Plot of potential functions considered by using different kernels. 11

3.5 The covariance plot between the two cars based on the covariance matrix in Equation 3.21. 12

3.6 The normal distribution when conditioning on your friend's price. 13

3.7 The function $f(x)$ gives the value of a car based on its age. 15

3.8 The prediction over the function after two random initialization points. 15

3.9 The prediction over the function after four points with two random initialization points and two points with UCB. 16

3.10 The prediction over the function after eight evaluation points. 16

3.11 The Euclidean distance between each pair of points inside a unit hypercube. 19

4.1 The overview of the SystemVerilog test environment. 24

4.2 The figure shows an overview of the components used by the Machine learning framework. Colored boxes are bigger-picture functionality, while white boxes provide extra detail. 28

4.3 The figure shows the components needed to use the ML framework for inference. . 33

5.1 The distribution of hit coverage points for different coverage types over 1000 seeds for device A. 36

5.2 The distribution of hit coverage points for different coverage types over 1000 seeds for device B. 37

5.3 The approximation of standard deviation and mean value for expressional coverage for device A with different amounts of seeds. 38

5.4 The approximation of standard deviation and mean value for expressional coverage for device B with different amounts of seeds. 39

5.5 How coverage types scale with random testing for between 1 to 1000 transactions for device A. 40

5.6 How coverage types scale with random testing for between 1 to 1000 transactions for device B. 41

5.7 Total compute time measured for the two devices with 200 000 transactions. 52

5.8 Measured ECT for all coverage types on device A. 53

5.9 Measured ECT for all coverage types on device B. 54

5.10 The scaling for the different coverage types between 1 and 10 000 transactions for device A. 55

5.11 The scaling for the different coverage types between 1 and 10 000 transactions for device B. 56

LIST OF TABLES

3.1 The state of the cars. 12

3.2 Hypercube edge length containing the $k = 10$ nearest points for $n = 10000$ point uniformly distributed inside a unit hypercube. 18

4.1 The number of code cover point for block A. 23

4.2 The number of code cover point for device B. 23

5.1 Calculated mean and standard variance accuracy for devices A and B for 15 samples over the true values for 1000 samples. 39

5.2 The results of testing different bounds for Bayesian optimization on device A. . . . 43

5.3 The results of testing different acquisition functions for Bayesian optimization on device A. 43

5.4 The results of testing different bounds for Bayesian optimization on device B. . . . 44

5.5 The results of testing different acquisition functions for Bayesian optimization on device B. 45

5.6 The results of changing the number of values assigned uniform probability in the weighted probability distribution. 46

5.7 Results of preforming a meta-optimization to find an optimal projection. 47

5.8 Coverage results for different verification methods for 100 transactions on device A. 48

5.9 Coverage results for different verification methods for 1000 transactions on device A. 48

5.10 Coverage results for different verification methods for 100 000 transactions on device A. 49

5.11 Coverage results for different verification methods for 200 transactions on device B. 50

5.12 Coverage results for different verification methods for 2000 transactions on device B. 50

5.13 Coverage results for different verification methods for 100 000 transactions on device B. 51

CHAPTER 1 INTRODUCTION

The last year has, by some, been referred to as the AI revolution, thanks to the giant leaps made in the field. One of the biggest advances has been Chat-GPT, which has had one of the fastest adoption rates ever in terms of time to reach 100 million users [2]. This tells us something about the scale at which people are interested in this technology and how much impact it can have on our life. Even though Chat-GPT and other large language models seem pretty general purpose, there are still many problems they do not excel at. In this report, we look at the use of AI and machine learning to take a data-driven approach to verifying digital circuits.

One of the main drivers of this AI revolution is the extreme increase in hardware performance over the last decades. The hardware has been able to have this massive increase, in large parts, thanks to Moore's Law [3], which states that the number of transistors in a circuit doubles every 18 months. It has allowed more complex and large neural networks and machine learning models. However, this increase has come at the cost of making the designs harder to verify. As the complexity grows, verifying that a design is functioning correctly pre-silicon becomes increasingly harder. This difference between what can be designed and verified is called the verification gap, and it is slowly growing. To improve this, we need to optimize the verification process. Because fixing bugs after the chip is taped out very quickly becomes costly, much time is spent simulating the design up front to find and squash bugs. However, the design simulation is slow compared to the test on silicon, so having an efficient set of tests is paramount. Determining and finding efficient stimuli can be very complex because of the complexity of the underlying circuit.

The motivation for writing this thesis is to explore the potential to apply Machine Learning and AI to find a more data-driven approach to understanding the correlation between input stimuli and coverage. Our goal for the simulation is to have tested all the functionality, which the amount of coverage can measure. Since which functionality is triggered directly correlates to the inputs, an ML framework can control which parts get covered by guiding the input stimuli. So, this is an optimization problem of finding a set of inputs that gives the highest coverage.

A more data-driven approach to finding inputs will not replace verification engineers but rather complement them. Much guesswork goes into selecting and finding an optimized set of stimuli, together with some trial and error. By doing this with ML, there is a potential for saving many engineering hours that could be spent on writing properties for new features allowing companies to deliver more features in every product, producing an even faster development rate. There is also the potential to save money and the environment because of fewer CPU hours used for simulation, reducing the power consumption of data centers running these simulations. If ML and AI can optimize this part, the potential savings can be used in various ways.

1.1 Report Outline

Chapter 2 is the problem statement for the thesis.

Chapter 3 presents all the relevant theories necessary to understand the solution developed in this report to solve the problem presented in chapter 2. The sections introduce the fundamentals of Bayesian statistics before introducing Bayesian optimization. After introducing Bayesian optimization, we explain how the Gaussian process works together with other relevant components, which is the main step of Bayesian optimization. Throughout the chapter, there are some practical examples where the theory is applied to some smaller problems to give the reader a better understanding of how they work before using them on a much more complicated problem. Then we look at how we can potentially scale Bayesian optimization to higher dimensions.

Chapter 4 is the implementation part and goes into more detail about the solution. The chapter starts by introducing the two test devices. Then there is a look at the SystemVerilog testbench made to support the machine learning framework, which requires more interaction with the outside world than a standard testbench. The next part introduces how the machine learning framework interacts and functions with the testbench.

Chapter 5 presents the result found by using the system on the two test devices. The first part is concerned with finding the optimal parameters for the machine learning framework on each device. After finding the optimal solution, we can compare the result with other verification approaches, such as random and constraint random verification. We will also look at potential improvements in computing time.

Chapter 6 contains a discussion of the result, together with some of the limitations of the testing.

Chapter 7 contains some considerations and ideas for how future development should proceed.

Chapter 8 are concluding remarks and the conclusion of the thesis.

CHAPTER 2 PROBLEM DESCRIPTION

The high-level objective of this master thesis is to explore a more data-driven approach for simulation-based verification of digital circuits. Verification engineers will still be a hugely important part of the verification process, but machine learning might solve some tasks better. One task that machine learning systems, in general, might solve better than humans is the optimization of input stimuli to a device under test (DUT). Both data and control signals going to the DUT need to be controlled during simulation, but some stimuli might be redundant since it is not testing new parts of the design. The goal is to minimize the number of redundant stimuli and only use CPU hours to simulate stimuli which adds coverage.

Simulation-based verification methods are heavily centered around the principles of constrained random verification. Constrained random verification allows for the randomization of input stimuli within a set of constraints. Even inside the constraints, one might find more or less interesting stimuli that require more computation inside the device or test more functionality. Sitmuli, which increase coverage quickly, should be more likely to be chosen. One way to do this is to assign weight to such stimuli. By changing these weights, one can increase the likelihood that interesting events would happen in the circuit. The goal is to modify this process by allowing a machine-learning method to control these weights.

Creating a machine learning-optimized testbench requires a new form of testbench design to allow for tight integration and control with the machine learning method. As such, this thesis will explore how such a test bench can be created and how machine learning can be wrapped around such a system. Tunning the weights will be done by evaluating the coverage results after a fixed amount of stimuli, with the goal of observing coverage increase. The best weights found should be saved for later use when the system completes its training phase. Ideally, the system should function for all devices, regardless of function or specification. To evaluate the solution's usefulness, we will compare it against a wide range of common verification methods. This allows us to evaluate whether it achieves better coverage for a set amount of stimuli. Finally, the machine learning efficiency will compare to see whether it can save CPU hours.

This thesis has aimed to implement one way to optimize the weights based on Bayesian optimization. To the author's knowledge, it is the first implementation of such a system for machine learning used for verification. The goal is to conduct a rigorous analysis of this system and provide some recommendations and pitfalls that might help the field going forward. Since the field is in such an early stage, there are still many possibilities for methods to explore and evaluate, and this is such an initial exploration.

CHAPTER 3

THEORY

This chapter introduces a theory relevant to the implementation part. It is quite a large section, with various topics broken off with some examples. The goal of having some examples in this section is to show the reader how to apply the theory to a smaller problem; this should be the foundation when using it to improve verification. Most examples also contain plots or other visual aids to show the process step by step.

3.1 Bayesian statistics

This section introduces the field of Bayesian statistics, where probability measures how certain we are that an event will happen. The higher the probability, the more likely the event will happen. In Bayesian statistics, the foundation is what we know about the event or a belief about it. We can use both to predict the initial probability of the event occurring. Bayes theorem updates this initial prediction based on new information, typically an experiment. Bayes' theorem is given by Equation 3.1.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3.1}$$

If we used Bayes theorem on a dice, we assume it has a 1/6 probability to land on one side. This would be our initial belief about how the dice should behave in Bayesian statistics. However, we might roll this die eight times, and it shows six eyes five of these times. The suspicion is then that the probability for each side is not equal but skewed towards six eyes. So in Equation 3.1 event A denotes the event where the dice show six eyes, while in event B is how many sixes we observe on eight tosses. Consider if the dice are weighted on one side. $P(B|A)$ would be the probability for use observing 5 out eight 8 throws landing on 6 if the dice indeed had 1/6 probability for each side. We can find our updated belief about how certain we are for the dice to show 6 by applying Bayes Theorem using all known probabilities. The probability $P(B)$ is the probability for the event B to happen. It can be calculated using the law of total probability in Equation 3.2, which only applies when there is a finite set of outcomes. This is the probability of observing six out of eight throws to show six eyes given all possible weighting of the dice. Finally, $P(A)$ is called the prior and is our belief about how the dice are weighted, so 1/6. After using all these known probabilities in Equation 3.1, we get an updated probability for event A, which is 1/4 with these numbers.

$$P(B) = P(B|A_1)P(A_1) + P(B|A_2)P(A_2) + \cdots + P(B|A_n)P(A_n) = \sum_{i=1}^{n} P(B|A_i)P(A_i) \tag{3.2}$$

If there are infinitely many outcomes from our experiment, the calculating $P(B)$ involves the integral over all values of A. The integral needed to find the total probability for an event B when we know the probability density function $f_A(x)$ is given by Equation 3.3.

$$P(B) = \int_{-\inf}^{\inf} P(B|A = x) f_A(x) dx \tag{3.3}$$

One special property of Bayesian statistics is that it captures our belief, and there is no need to give a scientific reason for this initial belief. It can be whatever we want it to be. If our initial belief about an event is far from what is true, our probability will slowly converge toward the actual value. However, the more our initial guess is, the more experiments need to be performed before it converges. We would need to throw the dice eight times and count the number of six-eyed sides multiple times until the probability stopped changing when using Bayes' theorem. The assumptions made in Bayesian statistics are explicitly stated compared to other models where this would be hidden in the distribution the data is forced to fit into.

3.2 Bayesian optimization

Bayesian optimization (BO) is based on Bayesian statistics principles and leverages Bayes theory. It is a machine-learning optimization method to solve the black-box optimization problem. The black box optimization problem can be formulated as follows, given a function f with an input $\mathbf{x}$, and the goal is to find the parameters which maximize the value of the function f inside the bound of legal values A. This formulation is equitant with the one found in Equation 3.4.

$$\max_{\mathbf{x} \in A} f(\mathbf{x}) \tag{3.4}$$

We can call this a black box optimization problem because we have no insight into what the function f represents. The only thing we know about f is that it has a set amount of input parameters, and the function results in a scalar value y. Bayesian optimization is a derivative-free optimization method because it does not know the internals of the function f or the derivative. In normal machine learning, we would compute the loss between the output y and some target t a propagate this change using the known derivative of f. However, bayesian optimization cannot access this information, so it starts by evaluating f at k initial points according to some policy. This policy might be as simple as randomly selecting a point inside the legal set A. After this initial evaluation, the algorithm places a posterior probability distribution corresponding to our belief of the values at $f(\mathbf{x})$. The algorithm then has a budget of N evaluations of the function f, so we terminate the process once we have done $k + N$ evaluations. The algorithm uses an acquisition function that determines which point is the best to evaluate next. When the new point is evaluated, the posterior prior is updated according to the new information. This is done with the use of a Gaussian process. The pseudo-code for this algorithm is presented in Algorithm 1 [4].

For the function f to be well suited for use with Bayesian optimization, it should have the following properties [4]:

- The function f has a set for inputs $\mathbf{x}$ from $\mathbb{R}^d$ where d is reasonably small.
- The legal set of values, A, should be a hyper-rectangle such that it is easy to access if a point is a part of the legal set.
- The function is continuous for all legal values of $\mathbf{x}$.
- f has no special know structures, so modeling it using this would be more efficient.
- We have no access to the first or second-order derivative.

Algorithm 1 Pseudo-code for Bayesian optmization

```
Sample k points from f
Make a posterior probability distribution over f
for i ← 1, N do
    Compute the point x, which maximizes the acquisition function for the current posterior distribution
    Observe the function at the computed point f(x)
    Update the posterior probability distribution over f
    if f(x) > CurrentMax then
        CurrentMax ← f(x)
    end if
end for
return CurrentMax
```

These rules formalize that f is a black box function, and the goal is to find the optimal point inside the allowed set A. The function should also not have any known structures, but utilizing these results structures might yield more optimal results. Known structures mean if we know it is a polynomial, convex, or concave inside the set of legal input. Bayesian optimization is what is called a surrogate method. The reason why these kinds of methods are called surrogate methods is the use of a surrogate function to approximate the actual function. The surrogate is much less expensive to evaluate than the actual function, and this function is the estimation of our beliefs about the function f. The surrogate function is acquired using Bayesian statistics and Gaussian process regression.

A Gaussian process creates the surrogate model in Bayesian optimization; the details are explained in section 3.5. But it can be considered a probability distribution over functions, so for every point $\mathbf{x}$, it can return a mean value $\mu(\mathbf{x})$ and a standard deviation $\sigma(\mathbf{x})$. So each point is normally distributed and defined by this mean and standard deviation. This lets us say how certain we are about our prediction. If we are unsure which values the point can have, it is assigned a standard deviation. This posterior distribution is the surrogate function for f and is cheaper to evaluate at an arbitrary point. Using an acquisition function, we can efficiently explore the search space to find a global optimum.

3.3 Multivariate normal distribution

A multivariate normal distribution is a distribution where all random variables are normal distributions. Such a distribution can also be called multivariate Gaussian. One of the unique properties of the Gaussian distribution is that when multiple random variables are Gaussian distributed, their joint distribution is going to be Gaussian [5]. The Multivariate normal distribution can are defined by the mean vector μ and a covariance matrix Σ.

The mean vector is defined by the mean of each normal distribution, which gives a mean for one specific dimension. So a random variable $\mathbf{x}$ is multivariant and normally distributed according to Equation 3.5. The covariance matrix Σ tells us how variables in each dimension are correlated, such as how a change in one dimension would affect the value in another. Two unique properties of the covariance matrix are that it is symmetric and positive semi-definite [4]. All these means that the covariance matrix is positive for all entries. The covariance is calculated according to Equation 3.6 between two random variables i and j where COV defines a covariance function determining how much a change in one variable would change in the other.

$$\boldsymbol{x} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} \sim \mathcal{N}(\mu, \Sigma) \tag{3.5}$$

$$\Sigma = COV(X_i, X_j) \tag{3.6}$$

3.4 Properties of the Gaussian distribution

The Gaussian distribution has the nice property of being closed under conditioning and marginalization [6]. These properties mean the resulting distribution would also be Gaussian when applying these operations. We can explore this property given a multivariant Gaussian distribution with two random variables X and Y; this is given by Equation 3.7.

$$P_{X,y} = \begin{bmatrix} X \\ Y \end{bmatrix} \sim \mathcal{N}(\mu, \Sigma) = \mathcal{N}\begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}, \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{bmatrix} \tag{3.7}$$

Conditioning is the operation to show how much one variable is dependent on some other random variable. The conditioning on the variable X is given by Equation 3.8 [7].

$$X|Y \sim \mathcal{N}(\mu_X + \Sigma_{XY}\Sigma_{YY}^{-1}(Y - \mu_Y), \Sigma_{XX} - \Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{YX}) \tag{3.8}$$

Marginalization can be explained as given a random variable X has taken one specific value x, all values for y which can contribute to this result need to be considered. This requires summing or integrating over all values of y as done in Equation 3.9 [5].

$$p_X(x) = \int_y p_{X,Y}(x, y)dy = \int_y p_{X,Y}(x|y)p_Y(y)dy \tag{3.9}$$

3.5 Gaussian process

A Gaussian process (GP) is a way to model a function in Bayesian statistics. This section derives the Gaussian process from linear regression based on the work done in [6]. The function f we want to model is then sampled at N points, so the information needs to start a GP regression is a collection of points and corresponding values, a collection of $\mathbf{x}$ and y points we can denote this as our data D. The goal is the same as for any regression problem to find a model which can predict values given a point. Where GP differs from regular regression is that we are not committing to a model in order to be able to make a prediction. In GP, regression is rather concerned with directly modeling the distribution over the output y given an input $\mathbf{x}$. Like with linear regression where one would first choose to use linear regression on the problem and then find the set of parameters that gives the best prediction. Instead, we assume that each point is drawn from a Gaussian distribution; in GP regression, this resulting distribution is then multivariate normal [4].

We can start looking at the case where we want to do linear regression on a collection of points. This view is typically called the weight space view in the literature since we start by assuming we want to find the weight, which makes our observation most likely. Some noise ϵ influences our observations; this term models the uncertainty and error for an observation. In GP, we model this error as a normal distribution in the following way $\epsilon \sim \mathcal{N}(0, \psi^2)$. The regression model is defined by Equation 3.10a, which is the equation for a line trough origo in multiple dimensions. The noisy observation of our target function is given by Equation 3.10b.

$$f(x) = \mathbf{x}^T\mathbf{w} \tag{3.10a}$$

$$y = f(\mathbf{x}) + \epsilon \tag{3.10b}$$

Our goal can be expressed that given our data D, we can, given an input point $\mathbf{x}_*$ make a prediction y. We then want to find the values for the weights $\mathbf{w}$, which makes our observation most likely.

This is done by computing the marginal likelihood estimator, which tells us how likely a certain point is given a weight $\mathbf{w}$:

$$P(D|\mathbf{w}) = \prod_{i=0}^{n} P(y_i|x_i, \mathbf{w}) \tag{3.11}$$

Before we start regression using GP, we need to place a prior on the weights. The standard prior to use for the weight is with mean $\mu = 0$ and a covariance matrix Σ; a kernel can be used to define the covariance matrix. Different kernels are more discussed in subsection 3.5.2. So our initial prior on the weights are:

$$\mathbf{w} \sim \mathcal{N}(0, \Sigma) \tag{3.12}$$

We can, with the help of Bayes theorem, find the probability for the weight given the data D, and this corresponds to how likely we would be to observe the data D given that it was sampled from a model defined by the weight $\mathbf{w}$:

$$P(\mathbf{w}|D) = \frac{P(D|\mathbf{w})P(\mathbf{w})}{P(D)} \tag{3.13}$$

What we need to make a prediction y given a point $\mathbf{x}$ is a model. So given that we have a model, the most likely weights $\mathbf{w}$ given the data D. We can make a prediction given the weight in the following way:

$$P(y|\mathbf{x}, \mathbf{w}, D) = P(y|\mathbf{x}, \mathbf{w})P(\mathbf{w}|D) \tag{3.14}$$

We can then marginalize away the model by marginalizing out the weight $\mathbf{w}$. Since this is equivalent to integrating overall values for $\mathbf{w}$, it can be thought of as considering all possible weights. Considering all possible weights would be the same as considering all possible lines in the case of linear regression. This can then be thought of as having a distribution over infinite functions.

$$P(y|\mathbf{x}, D) = \int_{\mathbf{w}} P(y|\mathbf{x}, \mathbf{w})P(\mathbf{w}|D)d\mathbf{w} \tag{3.15}$$

The result of this marginalization is that $P(y|\mathbf{x}, D)$ would be normally distributed. The reason for this is that both terms are a normal distribution. $P(\mathbf{w}|D)$ is given by Equation 3.13. That equation is a sum of two Gaussian distributions, where $P(D|\mathbf{w})$ is Gaussian distributed since it is a product of many Gaussian given by Equation 3.11 and the weights which we assumed to be Gaussian distributed. The other term in Equation 3.15 $P(y|\mathbf{x}, \mathbf{w})$ is also Gaussian distributed since it is the probability for a value y given an input $\mathbf{x}$ and the weights $\mathbf{w}$. So this is again Gaussian distributed because of the assumption in Equation 3.12. When marginalizing out a variable from a Gaussian distribution, the result is also Gaussian. When marginalizing out the weight or, in other words considering all lines, we result in Equation 3.16. Since each set of weights corresponds to one linear model or one line, we have, by marginalizing it out, considering all possible lines.

$$P(y|\mathbf{x}, D) \sim \mathcal{N}(\mu, \Sigma) \tag{3.16}$$

In general, the mean of the Gaussian distribution is not very interesting, and it can be computed directly from the data D. The interesting parameter is then the covariance matrix or kernel. This specifies the covariance between the data and our prediction point $\mathbf{x}$. When doing linear regression, we assumed all the points would lie on a line. This can be translated to points that lie close to some arbitrary line and have a high covariance. So knowing the y_1 at a point $\mathbf{x_1}$ values of these points makes it easy to predict a value for a point $\mathbf{x_2}$. Similar to a small change in $\mathbf{x}$ value would result in some change in y. So by defining how we want to correlate points, we have incorporated

a model. So, instead of committing to one specific model, the covariance determines the general shape of considered functions.

3.5.1 Covariance plot

We can have a look at a covariance matrix for to set of variables y_1 and y_2. The covariance matrix for these two variables is defined in Equation 3.17. The numbers on the diagonal of the matrix would be the σ for each variable which would correspond to the assumed noise ϵ. The other number is the covariance, which is set to a value for this example. Covariance between two variables will always be a number between zero and one. With one being, the two variables are perfectly correlated; if one changes with plus 20, the other also changes with 20. If the covariance is zero, predicting what would happen with one variable is impossible based on the other one. But could be defined by a kernel or function.

$$\Sigma = \begin{pmatrix} 1 & 0.6 \\ 0.6 & 1 \end{pmatrix} \tag{3.17}$$

Then, we plotted the covariance between the two variables in Figure 3.1. If we think that y_1 was the data we would use to make a prediction, we can observe a positive correlation between y_1 and y_2. So if our data says that y_1 has a large value, we could expect that y_2 also would have a larger value. In this figure, the mean of both y_1 and y_2 is set to zero. The closer the ellipsis gets to a circle, the closer the covariance between the variables gets to zero.

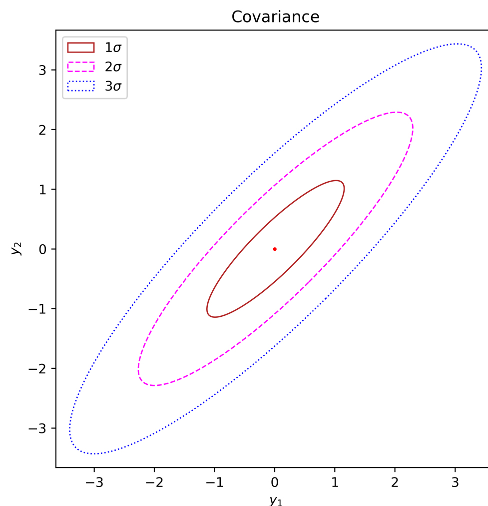


Figure 3.1: A covariance plot between two variables y_1 and y_2.

Say that we have measured y_1 to be 2 in our training data; we can see how this would affect our expectation for y_2. To do this, we need to condition on the variable y_1m, and the result is a normal distribution over the values for y_2 given by Figure 3.2.

So far, we have focused on computing a distribution for a finite set of points $\mathbf{x}$ given our data D. This does not translate to the definition of a Gaussian process as a prior over infinite dimensions of functions. So far, we can compute a prior on a finite set of points, or the covariance matrix would be infinite-dimensional. In practical applications, we compute this prior on a finite set of points, but it is a function of $\mathbf{x}$, so a corresponding y value can be computed for any input. All of this means that $f(\mathbf{x})$ can be approximated for any value of $\mathbf{x}$ by a GP with a mean function $\mu(\mathbf{x})$ and a covariance function $\Sigma(\mathbf{x})$ as given by Equation 3.18. Which kernels or covariance functions give rise to certain types of functions is further discussed in subsection 3.5.2.

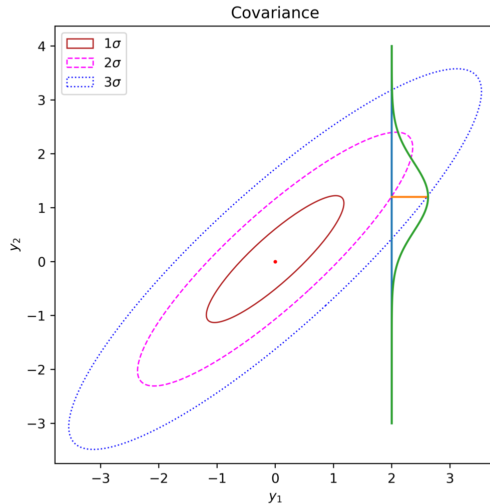


Figure 3.2: The resulting Gaussian distribution when conditioning on $y_1 = 2$.

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), \Sigma(\mathbf{x})) \tag{3.18}$$

3.5.2 Kernels

This section will introduce two kernels commonly used to define the covariance for a Gaussian process. Kernels are a generalization of the covariance matrix because they can tell the covariance between any two points. A kernel choice determines which shapes of functions Bayesian optimization will consider. One of the main foundations on which most kernels are built is that points that are close together have a high correlation, while points further apart have a smaller correlation.

Radial Basis Function

The Radial Basis Function (RBF) is a commonly used kernel, presented in Equation 3.19 [8]. The covariance between points decreases exponentially with increased distance. This covariance is plotted in Figure 3.3a, where the x-axis is the difference between $X_1 - X_2$.

$$K(X_1, X_2) = \sigma_f^2 exp\Big(-\frac{||X_1 - X_2||^2}{2\ell^2}\Big) \tag{3.19}$$

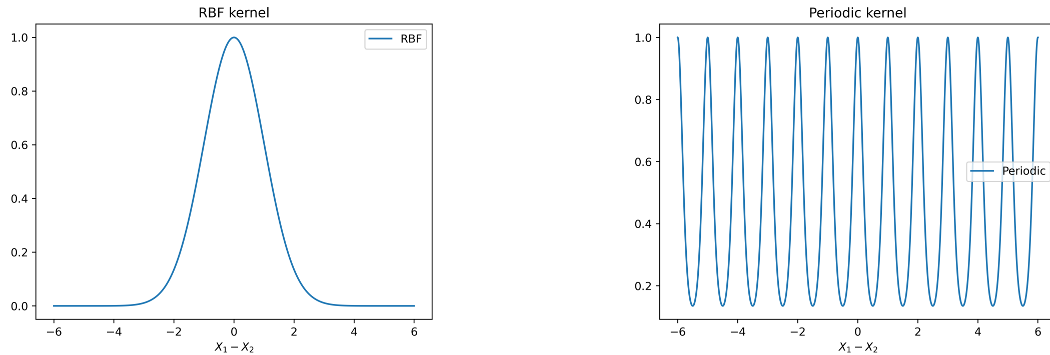
RBF computes the Euclidean distance between the points X_1 and X_2. The kernel has only one hyperparameter ℓ, which controls how far data can be extrapolated. All kernels have the parameter σ_f^2, a scaling factor that gives the average distance the function can be from the mean. If the points are very close or even equal, the value of the RBF kernel is 1. When the distance is large, RBF gives a value close to 0. The RBF function in Equation 3.19 is plotted for $\ell = 1$ with the x-axis as a function of $X_1 - X_2$. So for these values, points with a greater distance than 2 have very low covariance.

Periodic kernel

Another commonly used kernels are the periodic kernel. This kernel has two hyperparameters, ℓ, and p, how fare data is extrapolated and the period for the function. Applying a periodic kernel is useful for modeling periodic functions in Figure 3.3b, how the covariance varies with an increase

in distance. We can observe that the covariance is high if the points are separated by one period and decrease exponentially when elsewhere.

$$K(X_1, X_2) = \sigma_f^2 exp(-\frac{2}{\ell^2} sin^2(\pi \frac{X_1 - X_2}{p})) \tag{3.20}$$



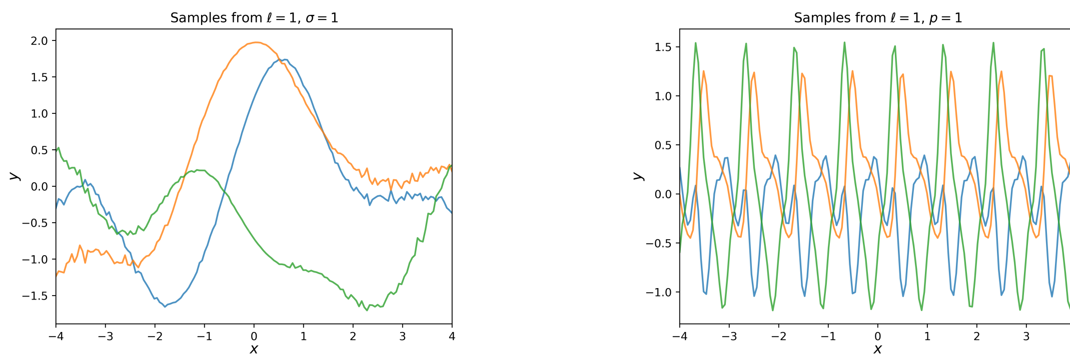
(a) RBF kernel plotted for $\sigma = 1$ on the interval $[-6, 6]$.

(b) Periodic kernel plotted for $l = 1$ and $p = 1$ on the interval $[-6, 6]$.

Figure 3.3: Plot of a selection of kernels.

Kernel function shapes

The kernel used to create the covariance matrix will determine the shape of the function, which the Bayesian optimization will consider. Such as, using an RBF kernel will never result in a linear function, so it would be a bad choice if the underlying data was linear. The kernel fully determines the general shape of the functions, which the Gaussian process will consider. In Figure 3.4, we have plotted the general shape of the function, which can be drawn from a distribution defined by the kernels presented in this section.



(a) A set of sample functions that might be considered using the RBF kernel.

(b) Different functions considered when using a periodic kernel.

Figure 3.4: Plot of potential functions considered by using different kernels.

3.5.3 A practical example of Gaussian process

We are now going to apply a Gaussian process to a practical example. Say that you want to find out how much you can sell your car for. You have a friend who has recently sold his car and wants to use this information to predict the price of your car. But your cars do differ in terms of mileage

and model year. These would be the $\mathbf{x}$ values, so the covariance when using a kernel would be computed based on the difference in mileage and year. Finally, the y_1 value would be how much your friend sold their car. So based on the $\mathbf{x}$ values, we know how much we should let your friend's price influence our expectation for the price of our car. The mean value μ has not to be considered much, but in this case, it would typically be the average price of cars sold. All these assumptions seem fair; if we had the same care as our friend with the same mileage and everything, we would expect to get the same price. However, one of us could be more or less lucky with how much someone pays for our car, given rise to some variance or noise. If our cars were vastly different, by being much older or newer, we would get less information from our friend's price. But how this correlation in price exactly is computed is defined by the kernel and can be tuned.

Assume that you will sell your car, and research Finn.no [9] and find out that the average price for your car model is 325 000 kr. This only indicates what people are listing the car for, not what people are paying for the cars. Fortunately, you have a friend with a car of the same model, which differs in year and mileage. But he has recently sold his car and tells you what he sold it for, which was 334 000 kr. You are now interested in what that means for your price.

Table 3.1: The state of the cars.

Feature	Friend	You
Year	2010	2012
Mileage (km)	120 000	89 000

We will use the Radial Basis Function (RBF), defined in Equation 3.19. For this problem, we have chosen the parameter ℓ to be 20000, and we will also assume that σ is 100. When then applying RBF to the information found in Table 3.1, we get the following covariance matrix:

$$\Sigma = \begin{pmatrix} 100^2 & 3000 \\ 3000 & 100^2 \end{pmatrix} \tag{3.21}$$

The covariance matrix is realized in Figure 3.5. In this figure, the axis is scaled to represent thousands of kroner.

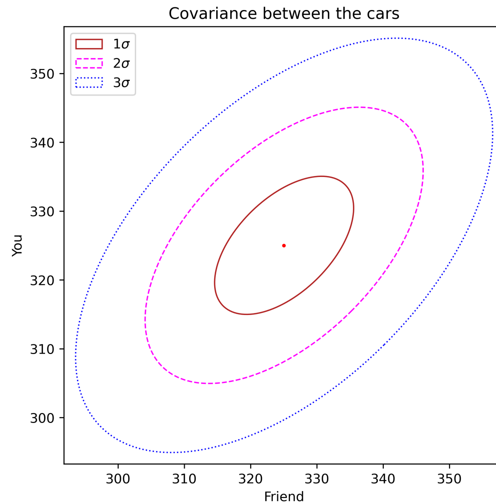


Figure 3.5: The covariance plot between the two cars based on the covariance matrix in Equation 3.21.

In Figure 3.5, it is possible to plot a line at 334000 kr, which was your friend's price. We can then condition this variable and finally get the probability distribution over the price of your car. To condition on your friend's car price, we can use Equation 3.8. In this equation, every variable is

known, such as the average car price and σ. This gives us that price distribution over your car is $\mathcal{N}(335000, 9100)$. We can then plot this normal distribution on the covariance plot, which gives us Figure 3.6.

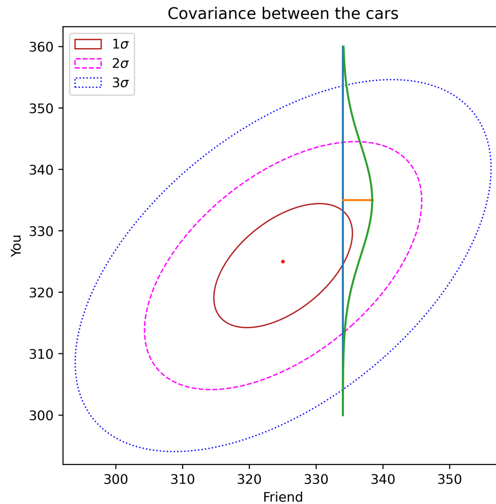


Figure 3.6: The normal distribution when conditioning on your friend's price.

Since the result here is just a Gaussian distribution with a known mean and standard deviation, it is easy to predict how certain you can be about selling your car in a specific price range. Given even more cars with a known selling price, we could improve the prediction to get more confident.

3.5.4 Acquisition function

The main task of the acquisition function is to determine which point to explore next. We can see that this is an essential part of Algorithm 1. For each x value inside the set of legal values, we have a Gaussian distribution over the y values. This distribution over the y values is the input to the acquisition function. A good acquisition function typically strikes a good balance between exploration and exploitation. Exploration is the term used for describing and evaluating the function in unexplored areas. So evaluating areas of the function where there is spare between evaluations, or it has not been evaluated. Exploitation is using the information we have gathered from previous evaluations such that the function is growing near a certain point, and this can be used to get even closer to the peak. The acquisition function takes the posterior distribution over functions and determines a point x, which should be evaluated next. Bayesian optimization utilizes proxy optimization to determine $argmax_x a(x)$, where $a(x)$ is the acquisition function. By optimizing this, the coordinate x should correspond to removing many uncertainties from the GP or finding the next best point for f.

Upper confidence bound

The upper confidence bound (UCB) balances exploration and exploitation by containing μ and σ directly [10]. By tunning β, the function can be encouraged to explore more or less. A high β would make it prefer regions with high uncertainty. A high uncertainty would indicate that there are spares with evaluations. While small β would make the function prefer regions with high mean prediction.

$$a(x, \beta) = \mu(x) - \beta\sigma(x) \tag{3.22}$$

UCB considers both the mean value μ at a point and the variance σ at that point which corresponds

to the uncertainty. So a point gets a high UCB score if there is a high mean prediction, and depending on the parameter β a point with high uncertainty will also get a high score.

Probability of improvment

The probability of improvement is the probability for the next point to be better than the currently best-observed point. This involves calculating the part of the Gaussian prior, which we have for every x, and seeing if this has a higher value than the current best observation $f(x*)$. The improvement is then assigned this value. The size of the value indicates how likely it is that the function value at point x is higher than the current best observation. The value does not include the size of the potential improvement; as such, it would typically prefer small but certain improvements over larger and more uncertain improvements.

$$a(x) = \max(f(x) - f(x*), 0) \tag{3.23}$$

Expected improvment

Where the probability of improvement only considers the chance of improving and not the size of a potential improvement. For many applications, a very small improvement might not be too interesting, especially if there is another point with a potentially larger improvement. This is done by taking the expectation value of the probability of improvement and is given by Equation 3.24. The expectation value is a weighted average between the improvement and how likely the improvement is. So the highest score would be given to a point with a very large improvement and a high chance of improving over the current best observation.

$$a(x) = E[f(x) - f(x*)] \tag{3.24}$$

3.6 A practical example of Bayesian optimization

This section will examine how the prior or surrogate function is updated during Bayesian optimization. It is based on code found in [11]. In the example, we want to predict the price of a car. We assume that the price only depends on the car's age. Furthermore, years with less production result in a higher price because cars are less available from that year. Our method for finding the most valuable car is to sell cars from our collection at an auction. Using what a certain car sold for an auction, we can find the most expensive car. This is a continuous example, so the car is between 0 and 10 years old. We can also sell any care within this range at the auction. But, we have a limitation on the number of cars we can sell, which is set to 8. After selling eight cars, the one which sold for the most is our guess for the age with the highest prices and lowest production.

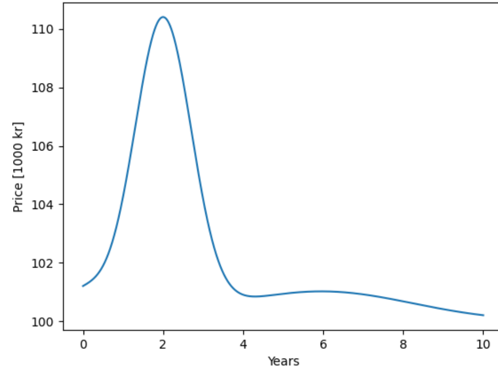


Figure 3.7: The function $f(x)$ gives the value of a car based on its age.

In this example, we know the function determining the car's price at the auction; it is plotted in Figure 3.7. We can observe that cars being 2 years old will sell for the highest price and are thus the rarest. The start of BO requires some random initialization points, and here we choose two such points. This then involves randomly picking two points between 0 and 10 and getting the function value at these. We are then constructing a prior, over-the-range of legal values. This prior is a distribution for every x value containing both a mean and variance. The mean value is our best guess for what value we will observe if we evaluate the point, while the variance is equal to our uncertainty. High variance indicates that the value might deviate a lot from the mean. It indicates that the prediction is not very good. After two random observations, the prior is shown in Figure 3.8. Here the mean prediction is a dotted line, and the 95% confidence interval is colored around.

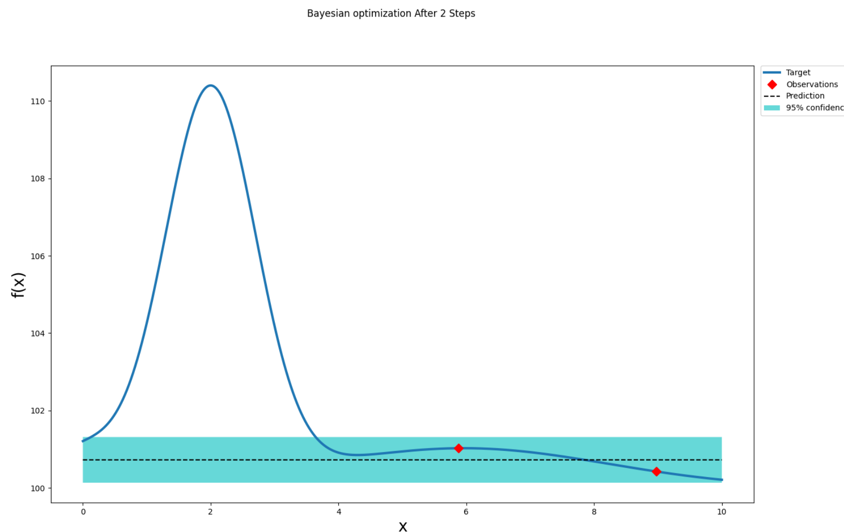


Figure 3.8: The prediction over the function after two random initialization points.

After the random initialization points, we move inside the for loop of Algorithm 1. The first step is to maximize the acquisition function; in this example, we use the UCB acquisition function introduced in section 3.5.4. It selects a point with the highest mix between mean prediction and variance. After selecting two iterations of the for loop, we get the update posterior in Figure 3.9.

The posterior is created by taking the values of the function at the known points and calculating how these will affect the value at a point x. From the example in subsection 3.5.3, this would equal varying how old your car was compared to your friend's car. So by substituting x for our

car's age, we have a function that gives the price for an arbitrary year. Selling more cars in the auction is the same as asking more than one friend what they sold their car for. However, we try to ask carefully friends who have sold cars in an interesting area of the function. The acquisition function determines this interesting area.

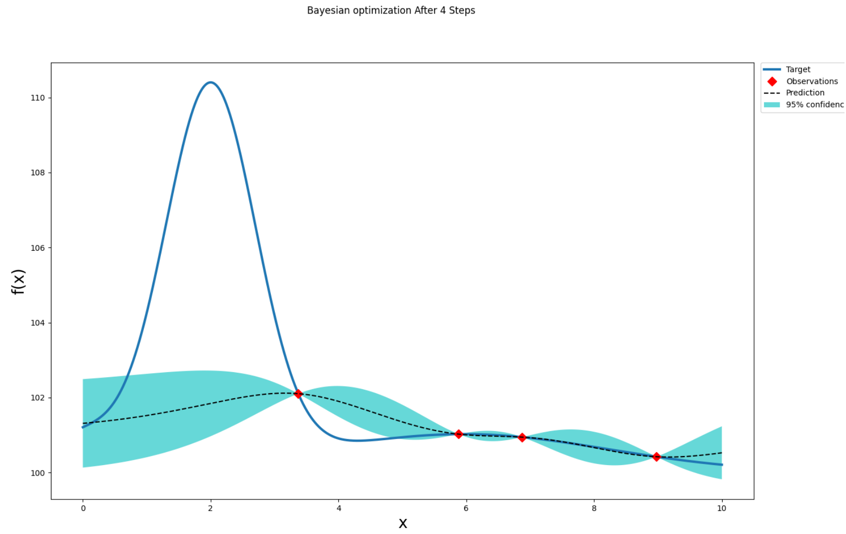


Figure 3.9: The prediction over the function after four points with two random initialization points and two points with UCB.

From Figure 3.9, we can see that our posterior from 4 to 0 has a quite high degree of uncertainty by the wide 95% confidence interval. This would be a region the acquisition function would prefer because of this combination. When adding the remaining four observations, we get the posterior in Figure 3.10. From this, it can be observed that all evaluation points were added for x values between 0 and 4 in x value. Our prediction for the most expensive car has a deviation in x value of $0,3$. So we are $0,3$ year away from the period with the lowest production and thus the highest car prices.

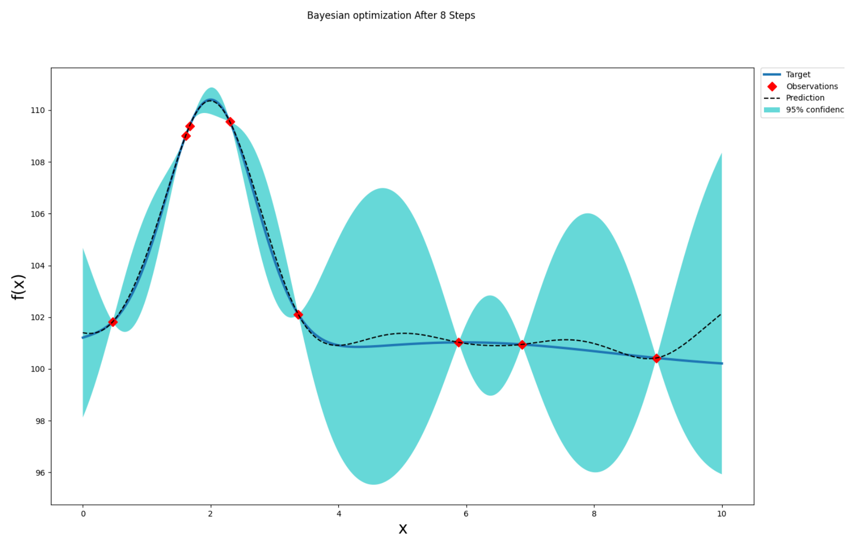


Figure 3.10: The prediction over the function after eight evaluation points.

3.7 Bayesian optimization in high dimensions

One big drawback with Bayesian optimization (BO) is that it becomes computationally infeasible in high dimensions. The current state-of-the-art struggles for problems with more than 20 dimensions [12]. One reason for the rise in computational cost is the computation of the inverse of the kernel matrix. This matrix grows with the number of observations, and higher dimensions require exponentially more observations to have a similar distance between each observation. The cost of computing the inverse is $\mathcal{O}(N^3)$ as a function of the size. So, the cost is not directly tied to the number of dimensions but rather to the number of observations. However, having more dimensions requires more observations to get a similar feel for how the functions vary in the space, which makes it essentially scale with the number of dimensions. Maximizing the acquisition function is directly dependent on the number of dimensions with a cost of $\mathcal{O}(\zeta^{-D})$ to reach an accuracy ζ [12] for D dimensions. Much of the current research on BO involves solving problems with computational cost, such that they can be applied to problems in higher dimensions.

There are currently two main methods for extending BO to high dimensions. The first assumes that the problem has a low intrinsic dimensionality. If a problem exists in three dimensions, but it exists on a plan, this would have an intrinsic dimensionality of two. A plain would be an example of a two-dimensional manifold, but manifolds extend into higher dimensions. On a two-dimensional manifold, two coordinates can fully describe the position. The other method assumes that the problem has an additive structure. For this to be true, the larger problem can be optimized by optimizing many smaller problems and combing their solutions.

Solving problems with an additive structure has been done by choosing a subset of dimensions and optimizing these on each iteration. The remaining dimensions are filled with random data or the previous best results. As long as the function is known to be additive, these methods have proven it can yield optimal results [13]. However, for problems that are non-additive, it might give a good result, but there is no research to prove this is the case in general.

For functions with low intrinsic dimensionality, it is possible to use a random embedding. This method involves solving the problem in a low dimension and the project the result to a higher dimension. The idea is that a point in the low-dimensional space maximizes the original function after being projected to the higher dimension. The random projection can be achieved by creating a matrix A with size $D \cdot d$ where D is the number of dimensions in the high dimensional space. At the same time, d is the dimension in the low dimensional space where BO tries to solve the problem.

Variational Autoencoders (VAE) have been used in the drug discovery field to scale BO to high dimensions. The method involves first training the VAE on a set of molecules. The decoder then has a lower dimensional input than output. By then, having a way to measure how good the output from the decoder is BO can be used to optimize the problem in the input dimension to the encoder. Since VAE is trained to produce output similar to the training data, every point in this low-dimensional space will share some traits with the data found in the training set. In the paper, this method could slightly optimize the quality of the outputted molecules [14].

3.8 Preceptron

A neural network is built by combining many smaller blocks called a perceptron. The perceptron was introduced by Rosenblatt in 1958 in this paper [15], so the idea is not new. However, due to the massive increase in computational power found in modern GPUs, it has been possible to utilize it to a greater extent. Different types of neural network architectures change how these perceptrons are connected. Input to the perceptron is defined as $[x_0, x_1, \ldots x_n]$ each input has a weight associated with it given as $[w_0, w_1, \ldots w_n]$, and the final component is a bias b. Bias is a constant that does not change with the input but is independent of this and affects the sum one way or another. The output from a perceptron is given by Equation 3.25, with n number of inputs and weights and a bias b. The parameters which can be controlled to achieve the desired output are the weights $[w_0, w_1, \ldots w_n]$ and the bias b. These are tuned such that the sum y matches a

target y_t.

$$y = b + \sum_{i=0}^{n} x_i \cdot w_i \tag{3.25}$$

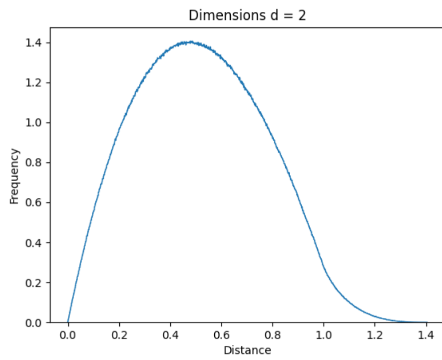
3.8.1 Curse of dimensionality

The Curse of dimensionality describes what happens with data when analyzing and clustering data in high dimensions. Some of the most significant issues are that data is becoming sparse and far away from each other. The point of data becoming sparse can be illustrated by looking at points uniformly distributed inside a unit hypercube in different dimensions. A unit hypercube has edges from 0 to 1 in all dimensions, with a length of 1. Then n points are spread uniformly inside this hypercube, so all points are equally likely to be chosen. The goal is to define a smaller hypercube containing the k nearest neighbors for every point. If the edges of this cube are small, it means that nearby points lie quite close together. If the edge length of the cube containing the k nearest neighbors starts to approach 1, the data is very sparse and lies on the edge of the hypercube. The edge length l of the hypercube containing the k nearest points is given by $l \approx (\frac{k}{n})^{1/d}$ [16]. If we set a fixed amount of points to distribute uniformly inside the hypercube and a fixed number of neighbors, we can observe how the edge length of the cube increases with the number of dimensions. This experiment is shown in Table 3.2; in tree dimension, each side only needs to be 0.03 long, while in 100, it was $0,9$. So in 100 dimensions, the edge length of a cube containing the ten nearest points is only $0,1$ shorter than the hypercube, which contains all 1000 points. So all data points can be thought of as being far apart.

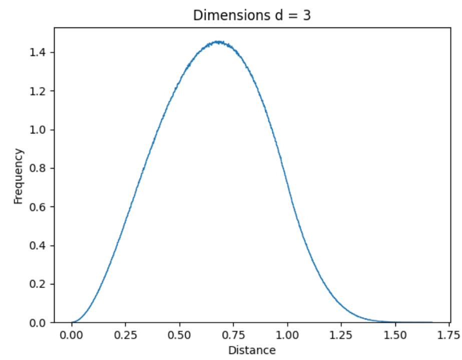
Table 3.2: Hypercube edge length containing the $k = 10$ nearest points for $n = 10000$ point uniformly distributed inside a unit hypercube.

d	l
2	0.03
3	0.1
10	0.5
100	0.9

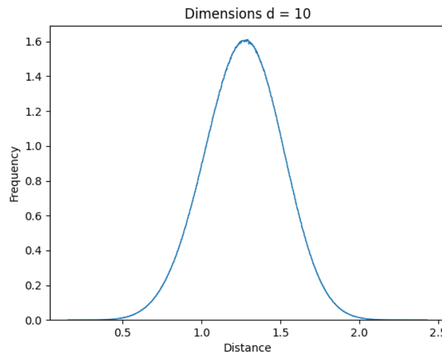
Another problem is that distance between points varies less in higher dimensional spaces. If we plot the distribution of Euclidean distances between points uniformly distributed inside a unit hypercube, we can observe this. These distributions are shown in Figure 3.11. We can observe that it is not just the mean value of the distribution which is shifting. The distribution changes as the dimensions increase, so all distances lie closer to the average. This makes it hard to apply the information we know about one point to make a prediction about another point. This is because almost all points are the same distance away, so all distances would be equal in enough dimensions if the points were uniformly distributed. However, if we know a point is either a member of class A or B, knowing which class one point is a member of would not allow us to conclude that all points also are a member of this class. We need some distance difference between points to use information about one point to make a prediction for another.



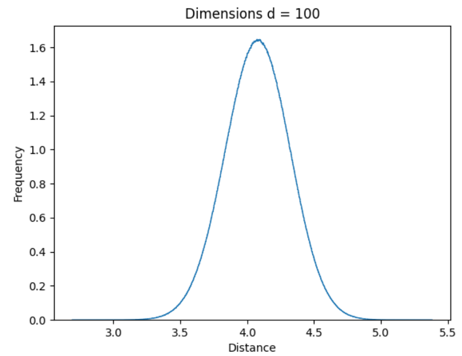
(a) The distributions of distances for 2 dimensions.



(b) The distributions of distances for 3 dimensions.



(c) The distributions of distances for 10 dimensions.



(d) The distributions of distances for 100 dimensions.

Figure 3.11: The Euclidean distance between each pair of points inside a unit hypercube.

3.9 Coverage

This section introduces different types of coverage, what they represent, and which conditions need to be satisfied for us to call them covered. The information in this section is a refined version of the introduction to coverage found in the project thesis from the fall of 2022 see [1]. Coverage gives us a measurement of how thoroughly the design is tested. A higher coverage number indicates that more functionality more of functionality was tested. So, when a design has 100% coverage and no known faults, we can be pretty sure the design functions correctly. There are two types of coverage code coverage and functional coverage. Both these types of coverage will be explained in the following sub-sections.

3.9.1 Code coverage

Code coverage is the name suggested related to the design code in resister transfer level language such as SystemVerilog. The simulator automatically extracts the different types of code coverage. What the different coverage types are called and how they are defined will vary slightly from one simulator to another, but this description should be valid for most simulators. We will focus on four coverage types: Statement, Branch, Expression, and Toggle.

Statement coverage

Statement coverage measures how often each line in the design code has been executed; the simulator will increment the count for each line every time it executes it during simulation.

Branch coverage

Branch coverage ensures that each execution path of an if/else conditional assignment statement is taken during simulation [17]. Full branch coverage requires that both the true and false path of the branch has been observed. Full branch coverage will require sig to have all values during simulation for the SystemVerilog case statement in Listing 3.1. If sig does not have all these values during simulation, one case will not be executed. If the case or branch has not to been executed during simulation, it is impossible to tell if its functionality is correct from the result.

```
1     logic [1:0] sig;
2     logic [1:0] res;
3     logic [7:0] read;
4     logic con;
5
6     always_comb begin
7         case (sig)
8             2'b11: read = 'h0X;
9             2'b10: read = 'hAX;
10            2'b01: read = 'h1B;
11            2'b00: read = 'hXX;
12        endcase
13    end
14
15    res = con ? 2'b01 : 2'b00;
```

Listing 3.1: Case statement in SystemVerilog.

Expression coverage

Expression coverage analyses expressions and ensures that all possible combinations resulting in true and false for an expression have been observed during simulation [17]. The other input to the expression must be in a state such that only one input control the output. Finally, the controlling signal must be observed for full coverage in the 0 and 1 states. Looking at a practical example such as the code in Listing 3.2, full expressional coverage for the and-expression on line 5 requires both signals A and B to control the result of the and operation. So A needs to be 0 and 1 while signal B needs to be 1. Since we know the truth table for a the and operation, we know that if signal B is 0, the output will always be 0 no matter what signal A is, so, effectively, signal A does not control the output. Also, the opposite needs to happen where signal A is 1 while B has both values 0 and 1.

```
1         logic out;
2         logic A;
3         logic B;
4         always_comb begin
5             if( A & B) begin
6                 out  = 1'b1;
7             else:
8                 out  = 1'b0;
9             end
10        end
```

Listing 3.2: If code block in SystemVerilog.

Toggle coverage

Toggle coverage record each time a signal transitions from one state to another. Toggle coverage can relate to the physical circuit and show parts not exercised during simulation [17]. Each bit must transition from 0 to 1 or 1 to 0 to get a toggle coverage of 100% for an array of multiple bits.

As the name suggests, functional coverage is tied to the design's functionality. The designer or verification engineer must write all functional cover points. Functional cover points ensure that all functionality is covered during the verification and that the design function as intended inside the specifications [17]. It can help ensure that the design complies with design constraints as intended. In Listing 3.3, we can see how cover points are defined for the two operands and the opcode. To ensure that all operands work together with all opcodes, the designer uses a cross, which creates one bin for all possible combinations of the three signals. If this, then get 100% covered; all opcodes have been observed for all possible combinations of registers. We can then be sure this functionality then works after verification.

```
1      logic clk;
2      logic [3:0] Operand_A, Operand_B;
3      logic [2:0] Opcode;
4
5      covergroup CG @(posedge clk);
6          c1: coverpoint Operand_A;
7          c2: coverpoint Operand_B;
8          c3: coverpoint Opcode;
9
10         Func: cross c1,c2,c3;
11     endgroup: CG
```

Listing 3.3: Case statement in SystemVerilog.

It is significantly more expensive to monitor and capture functional coverage. Functional coverage point results are captured by using a monitor, a construct that, during simulation, looks to see if the cover point is fulfilled. These monitors are typically more expensive than just capturing code coverage [17]. They actively monitor all signals, part of the cover-point, instead of counting how many times a line has been executed.

CHAPTER 4 IMPLEMENTATION

This chapter describes the detail of implementing a system that dynamically adjusts the weights for parameters in a SystemVerilog testbench. The goal is to find an optimal set of weights for the stimuli generator in the test bench measured in the number of achieved coverage bins. The chapter is structured in the following way, first is a short introduction to the two devices used for testing. Following is a breakdown of the SystemVerilog testbench and its subcomponents; before specific considerations for the two devices are described. The last part describes the machine learning framework for optimizing the SystemVerilog testbench's weight distribution.

4.1 Device under test (DUT)

The device under test, or DUT, will be used to evaluate how much improvement our system can achieve over other verification methods. Our test devices include one state machine and an arithmetic-heavy device. When selecting the devices, the goal was to find two contrasting devices to test how different functionality in the test circuits might affect the results. The state machine should provide a bigger challenge in linking commands together. For the arithmetic-heavy device, the input order will not matter; the goal is to find the right data items to operate on. In this unit, the number of possible inputs is quite large, with only a tiny percentage being important. So the challenge is quite different.

The two devices are a part of the Arm Mali GPU and are used to get a realistic test scenario. Since these are Arm Intellectual property (IP), this report will not provide details on their functionality or interface. Instead, we will focus on the number of cover points and lines of code to measure their complexity. Which types of signals the machine learning framework will control will also be presented.

4.1.1 Device A

This device is a state machine, whit a total of 19 states, and has 1055 unique lines of code. It has code cover points defined in Table 4.1. One of the main drivers for coverage in this unit will be to provide the sequence of commands to guide the state machine through all transitions. For this device, the machine learning framework will control 17 variables. It will control the weight of a distribution over the five control commands and two variables tied to a control signal. The remaining ten variables control the delay between input commands and the delay in responding to the DUT. A response or request can be delayed between 0 and 4 clock cycles.

Table 4.1: The number of code cover point for block A.

Cover type	Number
Statement	228
Branch	143
Expression	192
Toggle	2144

4.1.2 Device B

This device is a heavy arithmetic device constructed as a four-stage pipeline. Our system will try to optimize the distribution of six 32-bit numbers, which is a part of the input to the device. Based on the relative values of these six numbers, the device will perform different calculations and potentially hit some corner cases. The remaining inputs to the device are randomized inside of the constraint. This randomization will persist for all methods. Finding the optimal ones will be difficult because there is a total of 2^{32} 32-bit numbers. The device consists of three different subblocks and 2287 unique lines of code. The number of code coverage points for the device is given in Table 4.2.

Table 4.2: The number of code cover point for device B.

Cover type	Number
Statement	607
Branch	196
Expression	445
Toggle	11076

4.2 SystemVerilog Testbench

The primary purpose of the SystemVerilog testbench is to drive stimuli into the device under test and collect the coverage. The testbench aims to abstract as many DUT-specific considerations as possible from the machine learning framework. An overview of the verification components present in the testbench can be found in Figure 4.1. All components are named per the IEEE standard for Universal Verification Methodology (UVM) [18]. The project does not use UVM for this project as it is not sufficiently large or provides any advantages. However, many UVM principles were used when designing the testbench. Every verification component corresponds to one file; this chapter will further describe the functionality. Components are modular, so most changes can only affect a single component. It allows for great flexibility and reuse. Our test environment does not include components to check for the design's functional correctness; both devices are verified, and it was very unlikely to find any bugs.

4.2.1 Transaction item

The transaction class contains the information necessary to communicate between two or more components [18]. All signals that the test should control are defined in the transaction class. However, this class will not define signals needed for the protocol, such as ready or valid signals. All variables are defined with the *rand* keyword; this tells SystemVerilog it is a random variable that is uniformly distributed over its legal values. Creating a nonuniform distribution is done by creating a constraint that contains a SystemVerilog distribution. It is done by using *dist* as a keyword and assigning a weight to a value or range of values. A given value's probability of appearing in the output is proportional to its weight. Changing the weights allows controlling how frequently a value appears in the output. Specific values might yield better coverage since they exercise more functionality of the circuits. The optimal distribution of these values should yield more coverage for fewer transactions processed by the DUT.

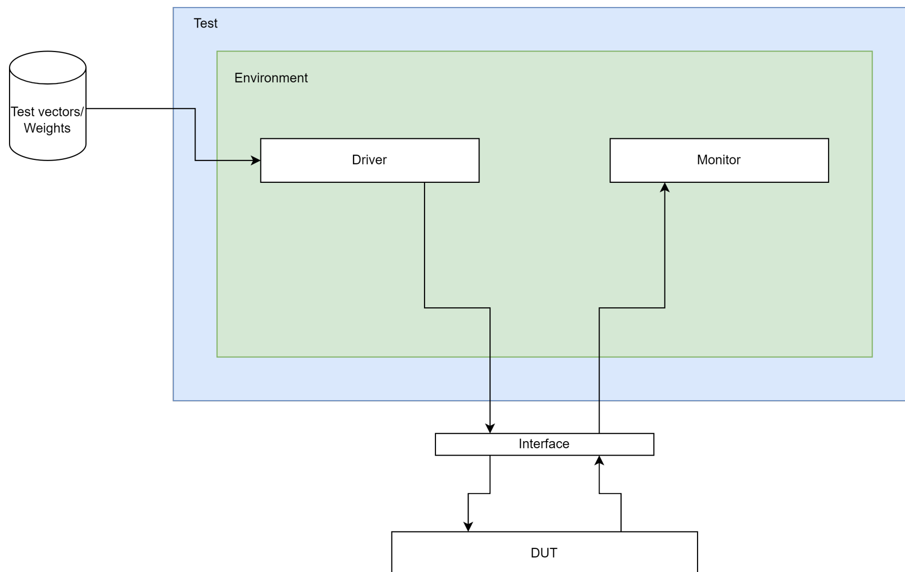


Figure 4.1: The overview of the SystemVerilog test environment.

Transaction items do not need to know the weights of the distributions at compile time. It is only necessary for the weights to be known right before the transaction object is created. This allows us to dynamically alter the weights without recompiling the testbench. This ML-enabled testbench reads the weight from a file during execution and uses them when creating the transaction item. Changing an existing testbench to read the weights from an external file should require minimal changes. This should be advantageous in cases where one would retrofit an existing testbench with ML integration.

All signals in the transaction item are ensured to be valid since constraints are specified, so they comply with the DUTs protocol. Constraint ensures that variables are randomized without resulting in an invalid or illegal value. Constraints can also be placed between variables to keep intervariable constraints. Variables defined as members of a class can be randomized by calling the function *randomize()* on the class instance. This function will either fail if the constraints can not be satisfied or succeed if it does.

4.2.2 Interface

The interface is a SystemVerilog construct that allows for grouping related signals. We can use an interface to group all signals to a DUT together, making passing the signals to different verification components easier. Interfaces can also have member functions and tasks, which makes them function almost like a class. Having the ability to construct task which functions on the interface allow for easy implementation of send and read functions over the interface. In our testbench, we will utilize this and create tasks for sending transaction items to the DUT. Together with reset, monitoring the DUT for a valid output and responding to requests from the DUT. Associating these tasks with the interface allows the underlying protocol for sending and receiving to change without being transparent for the other verification components.

The task has all control over how to communicate with the DUT. All operations that consume simulation time in SystemVerilog must be a task [19]. So, this requires tasks to perform sending and receiving operations over the interface. Some of the tasks on the interface will take transaction items as inputs. All data and control signals not part of the protocol are provided as input to the task from the transaction item. In summary, all information related to the communication protocol used by the DUT is contained in the interface.

4.2.3 Driver

The driver is the component that ties the transactions together with the DUT. In the driver class, incoming transactions are taken from the mailbox and sent over the interface to the DUT. Our driver is responsible for ensuring that new input to the DUT will not be sent before our DUT is ready for new stimuli. Waiting until the DUT is ready for new stimuli is done by calling a wait function implemented in the interface, which monitors the DUT output and returns when the DUT signals it is ready for new stimuli. Pseudocode for a driver and how it uses the interface to drive stimuli and wait can be found in Algorithm 2.

Algorithm 2 Pseudocode for the driver.

while $mailbox \neq empty$ **do**
  $item \leftarrow mailbox.get()$
  while $interface \neq ready$ **do**
    wait()
  end while
  $interface.send() \leftarrow item$
end while

4.2.4 Monitor

The monitor is a device that observes the signals from the DUT without driving them [18]. In this testbench, the monitor's functionality is limited to recording the number of transactions processed by the DUT. When the number of transactions matches the expected number, the monitor ends the simulation. Performing the check that the number of transactions in equals the transaction out ensures the testbench is not losing transactions. An example of pseudocode for the monitor's functionality is given in Algorithm 3.

Algorithm 3 Pseudocode of the monitor's function.

$ValidOutput \leftarrow 0$
while $ValidOutput \neq NumTransactions$ **do**
  if $interface.output = valid$ **then**
    $ValidOutput \leftarrow ValidOutput + 1$
  end if
end while
$finish$

4.2.5 Environment

The term environment is used for a class that defines the topology of the testbench [18]. The environment will contain both the driver and the monitor. Alternatively, even multiple monitors and drivers for devices which has multiple interfaces. Each component instantiated in the environment will run in a separate thread such that the components execute their functionality in parallel.

4.2.6 Test

The test is one configuration of the environment to test a particular functionality of the DUT [18]. On the test level, the number of transactions will be set, and possibly also which kinds of transactions or sequences of transactions. In our ML-enabled testbench, we will not provide this kind of detailed management and will only set the number of transaction items a test will run. Our test environment is responsible for reading in the weights, creating transaction items with these weights, and putting them in the mailbox for the driver. Pseudocode for creating transactions with weight from a file and sending these to the driver can be found in Algorithm 2. When all

these transactions are created, it will run the environment which applies stimuli to the DUT and end the simulation when all transactions are processed.

Algorithm 4 Pseudocode for generating transaction items in the test class.

$weights \leftarrow read("weights.txt")$
$NumItems \leftarrow 0$
while $NumItems \neq NumTransactions$ **do**
  $item \leftarrow Transaction(weights)$
  $item.randomize()$
  $malibox.put() \leftarrow item$
  $NumItems \leftarrow NumItems + 1$
end while

4.2.7 Top-level testbench

The top-level testbench bench module is called for running a simulation. This module contains one instance of the interface to the DUT, and the DUT is bound to this interface. It will also include a block for driving the required clocks. Then, the module will create an instance of the test class and connect the interface from the test class to the one found in the top-level DUT module. Finally, the function run is called on the test class to generate transactions and apply the stimuli to the DUT.

4.2.8 Testbench for Device A

This section will explain the data flow in the testbench for device A and the considerations made when making it. Device A needs two different drivers to provide valid stimuli. One sends requests and will be called *driver_tx*, while the other responds to requests and is named *driver_rx*. Both drivers have transaction items associated with them. For this testbench, we will control the number of transactions sent to *driver_tx*, so if we say the device has had 10 test vectors applied to it, *driver_tx* will have received ten transaction items. However, it can be a difference between the number of transactions processed by *driver_tx* and *driver_rx* because the DUT does not have a one-to-one ratio between input commands and required responses.

In the testbench, the test which will be executed is fixed and is a simple test that sends a fixed number of transactions to the DUT before it terminates. However, the exact content of each of these transactions is not fixed but rather randomized. Our control points will be the distribution for each of the random variables. By controlling this distribution, achieving a more optimal distribution of transaction items might be possible than what uniformly random can achieve. In the testbench, there is a total of 17 parameters that define the different distributions. The mapping from parameters to distributions is explained in subsection 4.1.1. The weights can be changed from run to run by editing a file which will be read in the test component as explained in subsection 4.2.6.

1. Call the simulator on the top-level testbench module and run the simulation.
2. The testbench instantiates the DUT, connects it to the interface, creates an instance of the test class, and runs it.
3. The file containing the weights for the transaction's random variable distributions is read.
4. Create new transactions with the imported weights, and calls randomize on them.
5. Put the created transactions in the mailbox to the correct driver.
6. In the test class, the environments run function is executed.

7. The environment creates a separate thread for running each component. In this case, it runs *driver_tx*, *driver_rx*, and the monitor.
8. *driver_tx* takes transactions from the mailbox and utilizes the interface to send them to the DUT.
9. *driver_rx* gets signals over the interface, and when it needs to respond, it takes a transaction item to determine how long it should wait before responding.
10. The monitor watches the device's output and will call finnish once driver_tx has processed all its transactions.

4.2.9 Testbench 1 for Device B

Two versions of the testbench have been created for device B, and this first requires a bigger involvement from a verification engineer to work. We have created two test benches for this device to see if machine learning is best used independently or in cooperation with a verification engineer. This first testbench uses the verification engineer (the author) to select which ranges of the six 32-bit numbers are important; this selection will be based on the device's specifications and knowledge about the design. Then the machine learning framework is allowed to tune the weight assigned to each of these ranges. It will work very similarly to the testbench for device A, with a file of weight read in the testbench to get the weighted distribution in the transaction class where the relevant ranges are already specified. It is identical to the testbench for device A, except it only has one driver *driver_tx* since the device does not require a response.

4.2.10 Testbench 2 for Device B

This section explains the changes made to the verification environment to make a complete standalone machine learning testbench for device B. So compear to testbench 1 for device B, a verification engineer no longer does any preselection of ranges. Finding these ranges is fully in the hands of the machine learning framework. Doing this requires more changes to the testbench's functionality, which will be outlined in this section.

The test for this device will focus on optimizing coverage by changing the data the DUT is operating on. So, given that the control signals are randomized, what is the best data? As the introduction about device B said, we will optimize six 32-bit numbers. In the transaction item, all control and data signals are defined; however, the 32-bit numbers are not defined as *rand*. Instead of generating these in the testbench, they will be generated directly by the ML framework and written into a file. So in the test class, we will not import weights but 32-bit numbers. In the test class, transactions are created and randomized; this results in a random set of control signals. Before the item is put in the mailbox, the six 32-bit numbers are read from the file and added to it. This results in a complete item that the driver can send to the DUT.

1. Step 1 and 2 is equal to the testbench for device A.
2. Generate transaction items and call randomize on these.
3. Read 32-bit numbers from the pre-generated set saved to a file.
4. Add six 32-bit numbers to each transaction item.
5. Put the transaction in the mailbox to the driver.
6. Call run on the environment class.
7. The driver takes a transaction from the mailbox and applies the stimuli over the interface to the DUT.
8. In parallel, the monitor reads the interface for valid outputs and terminates the test when the number matches the test preset number of transactions.

4.3 Machine learning framework

This section presents the details of the Machine learning framework based on Bayesian optimization. Two parts are important for the machine learning framework, the first being training and the other being inference. We will in this section start by introducing the general overview of how the Machine learning framework is trained. Afterward, we will transition to information about the libraries used for the implementation. Some more device-specific training loop details are then presented. The final parts present how the system is used during inference.

4.3.1 Training loop

This section explains the general training loop when using Bayesian optimization to tune the weights in a SystemVerilog testbench to improve coverage. In this phase, the goal is to let the ML framework try different weights and see how this affects the coverage output. When the system is done with training, we can take the most optimal weights and use this for inference, resulting in a more optimal distribution of transactions. The components used during training are presented in Figure 4.2. We can break it down into three different components: One is processing saved coverage from the SystemVerilog testbench named coverage converter. The second is the Bayesian optimization software, used to optimize the weight; this is contained in the box labeled Scikit-Optimize. The final part is converting the output from Bayesian optimization to a format SystemVerilog can read, and this component is called the weight preprocessor. All these components are run in a loop for a predetermined number of iterations which is how many times it is allowed to run the SytemVerilog testbench with a new set of weights.

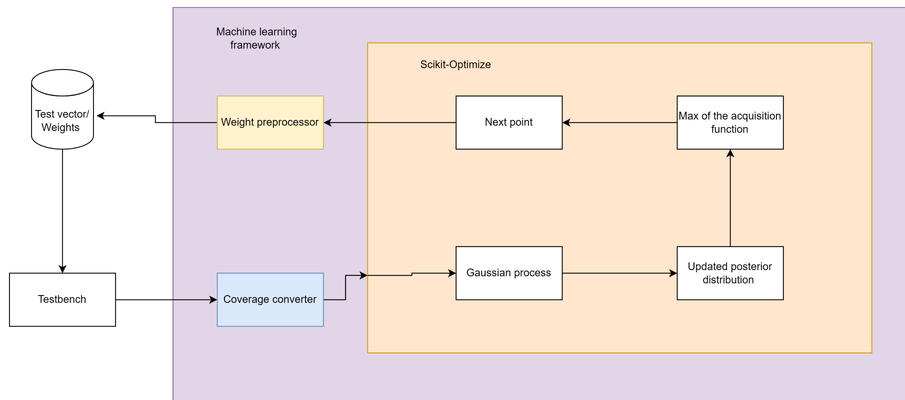


Figure 4.2: The figure shows an overview of the components used by the Machine learning framework. Colored boxes are bigger-picture functionality, while white boxes provide extra detail.

An overview of the components of the Machine learning framework is shown in Figure 4.2, Algorithm 5 contains the pseudocode giving a higher-level summary of the training loop. A very high-level summary is that it is very close to stock Bayesian optimization which has been explained in section 3.2 with some processing done to convert the output to a form where the SystemVerilog testbench can digest it. Also, the coverage database saved by the simulator needs to be preprocessed such that Scikit-Optimize can use the result. In detail, the algorithm works as follows, first, the currently best weights and results are initialized to 0. Scikit-Optimize will then provide some random weights as a starting point, which will be converted and saved so the testbench can use them. The testbench will then use the weight and save the coverage before the coverage is read and processed in Python into one number, the reaming bins to cover. This number corresponds to the function f value at a point from section 3.2. Our testbench is the function Bayesian optimization will optimize given the weight determining how likely a value is to be in the output. On the next iteration, Bayesian optimization will update the prior and use the acquisition function to determine the next point to evaluate. This loop will be repeated until it reaches the allowed number of evaluations n. When the process terminates, it returns the weights which produced the highest coverage.

Algorithm 5 Pseudocode for the machine learning frameworks training loop.

```
i ← n
BestRes ← 0
BestWeights ← 0
weights ← BayesianOptimization()
ConvertAndSave(weights)
RunTestbench()
while i ≠ 0 do
    ConvertCovergae()
    res ← read(Coverage)
    if res > BestRes then
        BestRes ← res
        BestWeights ← weights
    end if
    weights ← BayesianOptimization(res)
    ConvertAndSave(weights)
    RunTestbench()
end while
return BestWeights
```

4.3.2 Scikit-Optimize

Scikit-Optimize [20] was first introduced in August of 2016 and has since seen serval improvements. It is built on top of popular libraries such as NumPy, SciPy, and Scikit-Learn. Their goal is to make an easy and fast way to optimize expensive black-box functions. So, only gradient-free optimization methods are a part of the library. The library provides four optimization methods, *dummy_minimize()*, which is a random search within the bounds. Second is *forest_minimize()*, which uses decision trees to find the minimum of the function. Also, *gbrt_minimize()* is provided, using gradient-boosted decision trees to optimize the function. The final function is *gp_minimize()*, which is their implementation of Bayesian optimization using a Gaussian process; this will be the function we will use to perform Bayesian optimization on our weights.

Libraries such Scikit-Optimize must be provided with a well-defined Python function that can be called with a selected number of parameters and returns a single value indicating how good or bad the parameters are by some measurement. In our framework, the number of parameters corresponds to the weights in the testbench we want to control. We also need to specify what Scikit-Optimize call bounds for each parameter; this is the upper and lower value each parameter or weight can be set to. Scikti-Optimize will only then consider the weights inside these values, which is a requirement as stated section 3.2 the set of legal values needs to be inside a hyper-rectangle. Since Scikit-Optimize does not implement the ability to maximize a function, we pass the number of remaining bins to cover back as the function value. This is equal to maximizing the number of bins covered by the test.

4.3.3 Weight preprocessor

The term weight preprocessor describes the Python function, which takes the parameters provided by Scikit-Optimize and converts them into something usable for the SystemVerilog testbench. It will have different functionality depending on what the ML framework should control and which degrees of freedom it gets. This component has the most significant difference between devices A and B; implementation details for each device are found under their respective sections.

4.3.4 TensorFlow

TensorFlow [21] is an interface that can quickly describe and implement machine learning algorithms. It also supports acceleration with the use of GPUs or dedicated ML processors. TensorFlow also has a native Python API together with C++ and Java. This allows for easy prototyping and testing using Python, while production code can be ported to C++ for better performance. In this project, TensorFlow is used for all implementations of neural networks.

In addition, TensorFlow supports the Keras API [22], which aims to allow for fast prototyping of models. Keras can be thought of as the highest abstraction level of TensorFlow. It is achieved by providing higher-level constructs such as layers or activation functions. The functional or sequential model allows users to connect layers to make complex neural networks.

4.3.5 Coverage converter

The coverage converter is responsible for distilling down the results from the simulation into one single number representing how good the weight provided by Scikit-Optimize was. Coverage from a simulation is saved in a coverage database, which Python can not read, and the conversion needs to happen to get it in the correct format. Inside Arm, a tool is developed to convert the tool-specific database representation into a CSV format. The CSV files can then be read and processed in Python, and each coverage type is separated into one file, so during training, the user can choose which coverage type the program should focus on optimizing. The CSV has a column for each bin and a number indicating the number of times the test hit the bin. Since we do not want to encourage the framework to hit the same bin repeatedly, all bins are either covered 1 or not covered 0. If we did not do this, all tests would be concentrated on hitting the same easy-to-hit bins since that would give the highest score. As said in subsection 4.3.2, the number of not covered bins is then counted and passed to Scikit-Optimize, which minimizes this number. This method is used for both of the two devices.

4.3.6 Training loop for device A

The training loop constructed for device A is close to that in subsection 4.3.1. In that, the ML framework's way of controlling the testbench is by changing the weights inside of the transaction class. There is a total of 17 weights in the transaction class, which Bayesian optimization will control. These weights will then control how likely the value is to appear in the output when drawing from the random weighted distribution. The coverage converter used for device A is the same as that presented in subsection 4.3.5. As such, the training loop consists of Scikti-Optimize outputting a point, and this point is converted to weights for the testbench. Then the testbench is run, and coverage is saved before the coverage converter processes it. Finally, the coverage converter returns the number of uncovered bins to Scikit-Optimize. In the result section, we will investigate how potential changes to Bayesian Optimization through Scikit-Optimize changes the results, such as acquisition function and change of bounds.

Weight preprocessor

The weight preprocessor for device A takes the output from Scikit-Optimize and writes them to a file that the testbench can read. Scikti-Optimize gives us what we can call a point, which for device A is a point in a 17-dimensional space, one dimension for each weight we want to control in the testbench. The conversion from point to weights happens by taking each point and ensuring it is an integer since SystemVerilog does not allow floating-point weights in a dist [18]. Then each integer number is written to a separate line in a txt file; before this file is read in the testbench to update the weighted distribution. It is important to ensure that each weight is written to the file in the same order on each iteration, not to change what the weight controls. So, this device has a one-to-one mapping between the number of coordinates in the point and the weights.

4.3.7 Training loop for device B testbench 1

Testbench 1 for device B has a similar training loop as that used by device A. The coverage converter unit has no changes over the one in subsection 4.3.5. The control over the transactions is done by controlling a set of weights. The system aims to change these weights to become more optimal. It is done with Bayesian optimization through the Scikti-Optimize library. Weights are, as such, changed to observe higher coverage possibly. The weighting processor is equal to that used for device A, as there is no difference in which format the weights need to be one. However, which parameters they map to are handled inside the testbench. So all the details on how this weight preprocessor works are found in section 4.3.6.

4.3.8 Training loop for device B testbench 2

The training loop used for testbench 2 one device B is more complex than the others due to our goal of controlling the data the device processes. Since the data the device operates on is six 32-bit numbers, it is impossible to associate each value with a weight. The weight preprocessor handles how to use less than 20 weights to produce a weighted distribution over the six 32-bit numbers. Because of the limitation of Bayesian optimization discussed in section 3.2, we can not allow it to control more than 20 parameters. A general overview of the steps in the training loop is as follows:

1. Create a neural network with d number of inputs and D number of outputs. Where $d << D$
2. Save the neural network as a model to a file.
3. Create d variable with some bounds like [-1,1] for BO to control.
4. BO gives us a value for each of the d variables, giving a probability distribution over D ranges after being passed through the network.
5. Draw N ranges from the probability distribution over D; each range might be drawn more than once.
6. We select a random number within each of the N ranges.
7. Each of the N numbers will be written to a file.
8. This file will then be read in SystemVerilog and provided as data to the DUT.
9. The number of cover points hit is our measurement of how good the point in d was. This is the value returned to BO.
10. BO then tries to find the most optimal values for each variable in d dimensions.

Weight preprocessor

The weight preprocessor for device B needs to do more heavy lifting in the training loop. This is because we want to perform some dimensionality reduction techniques. Considering each value of a 32-bit to be a dimension controlling the weighted distribution would be a problem in 2^{32} dimensions. However, as we remember, we can only control 20 such dimensions, so we need to do something to use 20 weights to control 2^{32} weights; this is called dimensionality reduction.

Our first attempt at reducing the number of dimensions is to group nearby numbers together. When grouping numbers, we assign them the same probability, so if there are not individual numbers that are important but rather range, this can be effective. If we say we have 20 ranges, we will divide 2^{32} into 20 equally larger ranges. Each number inside a range has a uniform probability, while each range has a weighted probability. What we will call the range size is the number of unique values in each range which, if divided into 20 ranges, would be 214748365. Allowing for more ranges should have the ability to get finer control of which data device B receives, but it might also make it harder to find the important ranges.

So if we want to increase the number of ranges past 20, we need to use a projection. A short introduction to how a projection works can be found in section 3.7, but the short recap is that it makes fewer coordinates into more coordinates. Think about it as a multiplication between two matrices, one that is 1x2 and the other is 2x3. The result will then be a matrix that is 1x3, which indicates we have gone from two to three dimensions by using the matrix as a projection. This holds as long as the elements in the matrix are non-zero. For our projection, we will use a dense neural network instead of a matrix; this allows some more flexibility. However, no fundamental difference exists between a dense or fully connected neural network and a matrix. But by using TensorFlow and dense neural networks, we can apply a softmax function to the output, making it a probability distribution; all elements will summarize into 1. When a neural network is created, it gets initialized with some weights according to the initializer used by TensorFlow

Both ranges and projections are used as techniques to reduce the dimensionality of our problem down to something manageable. In Algorithm 6, the pseudocode for the weight preprocessor for device B is presented. Here Scikit-Optimize gives us some weights, which are used as input to the neural network, resulting in a probability distribution of a chosen number of ranges larger than 20. Then we want to draw n numbers at random for this distribution. First, we draw a range from the probability distribution, and then a number is drawn uniformly from the inside of this range. Finally, these numbers are added to a list, which repeats until N numbers are drawn. These are then returned and written to a file for the testbench to read. So, controlling the input weight to the neural network should result in different probability distributions, resulting in a higher or lower probability of selecting specific numbers.

Algorithm 6 Pseudocode for random number generation using projections.

$weights \leftarrow BayesianOptimization()$
$ProbabilityDist \leftarrow NeuralNetwork(weights)$
for $i = 0$ to $n - 1$ **do**
    $range \leftarrow Draw(ProbabilityDist)$
    $number \leftarrow DrawUniform(range)$
    $TestNumbers.append(number)$
end for
return $TestNumbers$

For the pseudocode in Algorithm 6, we can observe that the neural network is never trained during the training loop of device B. It is intentional not to train the network, as this would result in a nonstatic mapping between the two dimensions. The neural network is not the machine learning part of this system; that role is filled by Bayesian optimization through Scikti-Optimize. If the weights were to change, the projection would be different on each iteration, and Bayesian optimization could try different inputs but still potentially get the same output. Even if we wanted to train the neural network, we have no target of what is a more optimal mapping than the random mapping produced by the initialization of the weights. So the neural network is randomly initialized with some weight by TensorFlow using the *glorot_uniform* initializer. It randomly sets each w_i in Equation 3.25 to something. Her x_i would be our input from Bayesian optimization, and y would be one of the coordinates in the high dimensional space. Doing this allows us to make as many y coordinates as needed with the same inputs x_i but with different weights. By changing the x_i, we change the value of the y's, which is the weight assigned to the ranges allowing us to control the weighted distribution. This is one possible way to make a random projection explained, which was introduced in section 3.7.

To reiterate what was said in section 3.7, scaling using a random projection which is what using the neural network will be like, has its limitations. Some of them are that we assume the problem has certain properties to work well, such as existing on a manifold in the high dimensional space. Our problem will not necessarily lie on such a manifold; we will see it in the results if this is not the case.

4.3.9 Inference

Inference is the other part of the machine-learning framework, and this is how we can use the information obtained during training. In general, the inference is more lite weight since no learning takes place; as such, we can remove some components compared to what is shown in Figure 4.2. When removing the components no longer needed, we end up with the system in Figure 4.3. The only component shared across these two systems is the weight preprocessor. For device A, the weight preprocessor only needs to run once to convert the best weight found during training into the format the testbench accepts.

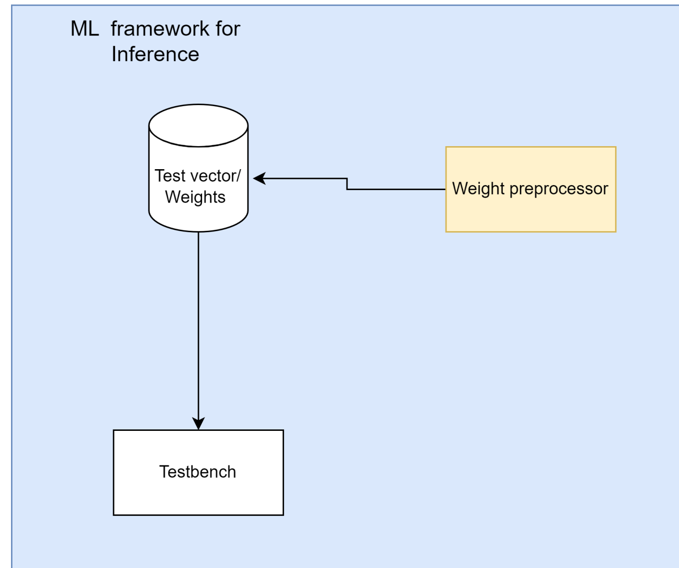


Figure 4.3: The figure shows the components needed to use the ML framework for inference.

For device B, inference functions differently as the weight preprocessor generates the data (32-bit numbers). To generate numbers, it will load in the best weights found by Scikit-Optimize during training and the neural network used for the projection during training. Since the network needs to be run at least once to get the probability distribution from which the numbers are drawn, there is more overhead related to inference for this method. All individual 32-bit values are then drawn from the distribution produced by the network, as explained in section 4.3.8. Finally, each number is written to a file the testbench can read and then add to the transaction item. As this is the case, we need to specify to the weight processor how many 32-bit numbers it should generate the same number needs to be set in the testbench.

CHAPTER 5 RESULTS

This chapter evaluates how well the Bayesian optimization Machine learning framework can improve coverage on the two devices from section 4.1. It starts with an overview of the testing methodology used. Some discussions and considerations related to each test result will be a part of this section, while more general discussions are carried out in chapter 6. After introducing the methodology, we will test the scaling of random simulation to find the number of transactions to test the system with. Next is a search for the most optimal parameters for Bayesian optimization for each device before comparing the best parameters with other verification methods. The result will so look at a comparison of computational cost across methods. Our final topic is scaling, looking into how Bayesian optimization scales with more transactions.

5.1 Testing methodology

The result in this section was collected using a Register transfer level (RTL) simulation with the Simens Questa Advance Simulator [23] with coverage collection turned on. Each simulation will result in toggle, expression, statement, and branch coverage. These coverage types will be used to determine how successful the ML framework is in improving the coverage. The machine learning testbench has all been allowed to train for 100 iterations, which should allow it to gather enough information [24]. The system can achieve two types of improvement an increase in coverage for a fixed number of tests or a lower compute time for the same coverage. Both can be useful, but it will depend on the intent behind the test.

Each method and setting we test will be evaluated based on the average coverage and the standard deviation. Since we are changing a weighted random generator's weights, the results will be stochastic and vary with different seeds. We will run multiple seeds to compensate for this and then calculate the average coverage and standard deviation. To determine how many seeds are needed for the results to become stable, we have investigated this in section 5.1.2. We usually prefer the one with the highest average coverage when comparing methods. However, all other things being equal, a method with a low variance is preferable. Finally, we want to determine how significant our observed results are. The significance of the results will be measured by its p-value, where a lower value means it is more unlikely to observe due to random variation. A p-value of 0.5 means there is a 50% chance one would observe this change in mean even if no changes to the weights were made. How the p-value is calculated is explained in subsection 5.1.1, while why the coverage types can be approximated as a normal distribution is explained in subsection 5.1.2.

The goal of this thesis has been to develop a system with the potential to reduce the cost of verification. There are two components related to this cost, the first being the verification engineers and the second being the number of CPU hours spent on running simulations. We will in this

thesis only focus on the second one. To determine this, we have focused on two measurements: the method's relative time for a fixed set of tests and the other a fixed compute where total coverage is measured. The first measurement will be called Total Compute Time (TCT) and measure the time it tasks for a fixed number of transactions. A lower time in this measurement means the methods are more efficient in creating transactions as the simulation part is fixed. While our other measurement, called Equivalent Compute Time (ECT), captures which method achieves the most coverage in the same amount of time. None of these measurements consider training time; we will discuss why in section 6.2.

5.1.1 Welch's t-test

We will use statistical power to reject or accept changes to the machine learning algorithm. Statistical power shows how unlikely our observation should be before we throw away the null hypothesis. We have a baseline for each setting we will test; statistical power gives us an estimate our improvement might have been too random variation. We will choose a p-value of 0.05, so observation has a five percent chance of being observed if the null hypothesis is true. The baseline configuration will be considered the null hypothesis, then a change from this setting will be the alternative hypothesis. The baseline configuration will be specified for each setting we wish to test. If we observe that the alternative hypothesis has an improvement over the null hypothesis, we will use this setting instead. Similarly, when comparing methods, we will only conclude that we see an improvement if we observe a p-value lower than 0.05.

We will use Welch's t-test to measure the significance of our observed result. This test determines if two different populations have an equal mean. Our population is the number of cover points hit for different seeds; we can calculate a population average and estimated standard deviation from these seeds. The Welch's t-test also can do this computation even for populations with different means. One condition for the Welch t-test to work is that both distributions are normally distributed; we will see if this is the case in subsection 5.1.2. When comparing the two distributions, Welch's t-test will return a number which is the statical t; this value can be used with the Student t distribution to get the p-value.

The Welch's t-test can be calculated according to Equation 5.1 [25]. Where $\overline{x}_1$ and $\overline{x}_2$ are the population's mean, and s_1^2 and s_2^2 are the variance of the population. Finally, n_1 and n_2 are the sizes of their respective populations.

$$t = \frac{(\overline{x}_1 - \overline{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5.1}$$

5.1.2 Normally distributed

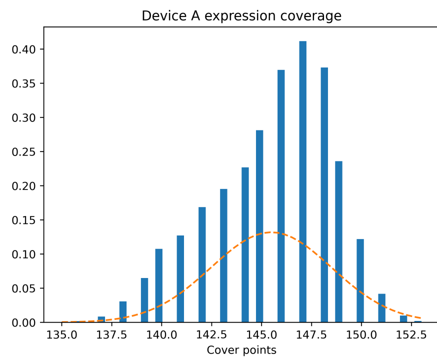
This section will determine if our different coverage types can be approximated as normally distributed. It is crucial to learn about this since Welch's t-test can only be used if they are. Our methodology for determining if a coverage type is normally distributed is to run 1000 random seeds and look at the coverage distribution. We can calculate a mean and standard deviation based on the coverage number from these 1000 seeds. Suppose the cover looks to be distributed similarly to the approximated normal distribution given by the calculated mean and standard deviation. In that case, it should be safe to consider the coverage type normally distributed. If the cover looks not normally distributed, using Welch's t-test will result in higher inaccuracy.

The distributions for the different coverage types will only partially follow the normal distribution given by the sample mean and standard deviation. Coverage produced by the simulator is a discrete distribution, so a bin is either hit or not for every seed. However, the normal distribution we calculate from the seeds is continuous, meaning a bin can exist in a partially hit state. For coverage types with a lower variance, discretization will have a bigger effect resulting in a distribution that looks less normally distributed. Also, running more seeds for the same weight will better approximate the underlying distribution.

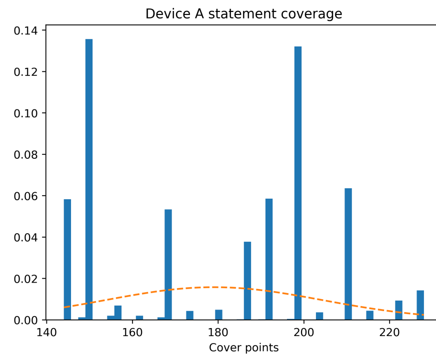
Device A

Here is a look at how the different coverage types for device A are distributed. In Figure 5.1, we have plotted the distribution of hit coverage points for 1000 different seeds. Different coverage types might be more or less normally distributed based on the functionality of the underlying circuits. We can see this in the figure, as both expressional in Figure 5.1a and toggle coverage in Figure 5.1c closely follows the dotted orange line, which is the calculated normal distribution. Based on this, we should get accurate results using Welch's t-test on device A coverage types.

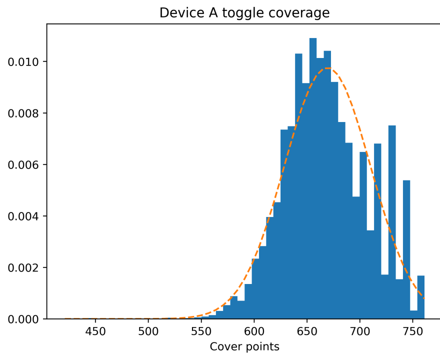
From Figure 5.1, we can see that both statement and branch coverage has a larger deviation from the normal distribution from the calculated normal distribution based on the observations. Statement coverage in Figure 5.1b has some high spikes, but these spikes are not close to the mean. If we compear it to the calculated normal distribution, we see that approximating statement coverage as normally distributed is not quite accurate. Using Welch's t-test on statement coverage is expected to yield inaccurate results because of how the coverage is distributed. Branch coverage in Figure 5.1d exhibits some of the same traits as statement coverage, but it almost resembles a distribution of multiple normal distributions. We can see this for the values around 110 and 120, which appear to follow a normally distributed locally. We still expect Welch's t-test to be slightly more inaccurate for this coverage type.



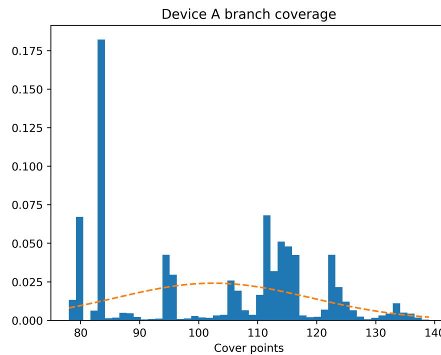
(a) Expressional coverage distributions for 1000 seeds, plotted with the normal distribution defined by the calculated mean and standard deviation.



(b) Statement coverage distribution for 1000 seeds, plotted with the calculated normal distribution.



(c) The distribution of hit toggle coverage points for 1000 seeds, plotted together with the normal distribution.

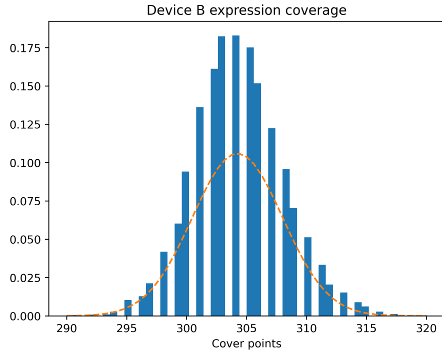


(d) Branch coverage for device A, for 1000 different seeds.

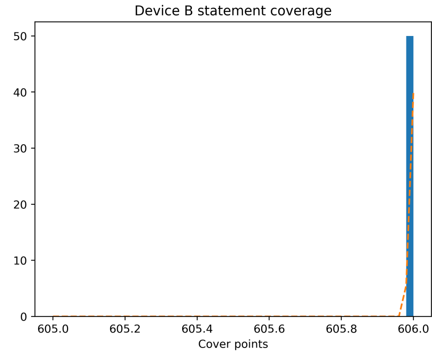
Figure 5.1: The distribution of hit coverage points for different coverage types over 1000 seeds for device A.

Device B

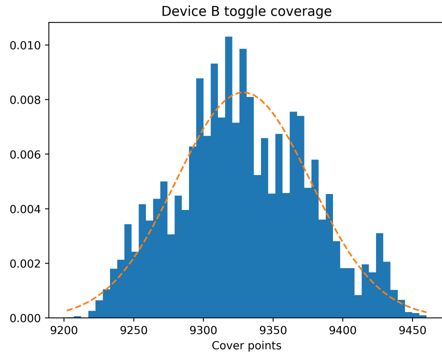
Here we present the coverage distributions for the different types of coverage on device B. From the Figure 5.2, we can observe a difference in how coverage gets distributed. The reason for this would be a combination of the device's functionality and how the test is constructed. Also, for this device, both expressional in Figure 5.2a and toggle coverage in Figure 5.2c look to be very close to normally distributed. For device B, branch coverage in Figure 5.2d also looks reasonably close to normally distributed and different from that of device A. It is a good thing as it allows us to use Welch's t-test with a very high degree of accuracy. Finally, we can observe that the distribution of statement coverage in Figure 5.2b has a constant probability of hitting 606 coverage bins. There is a chance it will yield inaccurate results when using Welch's t-test, but comparing the mean values, which should be reasonably accurate with this kind of distribution, is possible. Since the coverage is discreet, it can not be approximated well when it does not have a bigger variance.



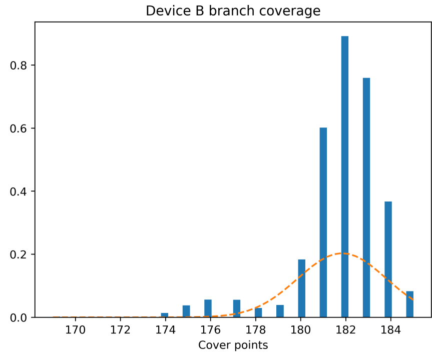
(a) Expressional coverage distributions for 1000 seeds, plotted with the normal distribution given by the calculated mean and standard deviation.



(b) Statement coverage distribution for 1000 seeds, plotted with the calculated normal distribution.



(c) The distribution of hit toggle coverage points for 1000 seeds, plotted together with the normal distribution.



(d) Branch coverage for device B, for 1000 different seeds.

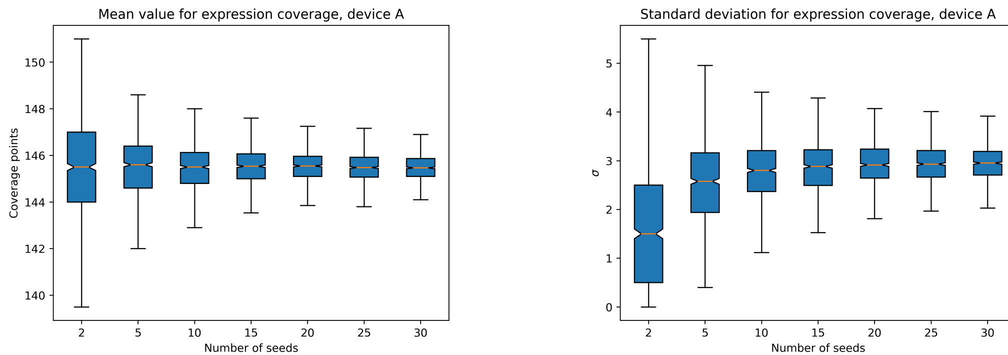
Figure 5.2: The distribution of hit coverage points for different coverage types over 1000 seeds for device B.

Number of seeds

In the previous section, most coverage types for devices A and B are well approximated as normally distributed. We simulated the same test 100 times with different seeds and examined the coverage results to prove this. However, running more seeds is more computationally expensive when evaluating the results. In this section, we wish to find the fewest seeds that still approximate the mean and standard deviation for coverage well.

The following methodology will be used to find the fewest numbers of seeds, which accurately approximates the mean and standard deviation. We will utilize the same 1000 seeds as in subsection 5.1.2, then randomly n of the seeds and calculate the mean and standard deviation based on these. We will do each number 1000 times to compensate for this being a stochastical process. Figure 5.3a, show a box plot for expressional coverage for different numbers of seeds; it can be interpreted in the following way. The orange line represents the average prediction on the x-axis, the number of seeds used for the calculation given. The blue box encapsulates one quantile on each side of the mean value. So one-fourth of the observations on each side of the mean are inside the blue box. The final part of the figures is called the whiskers, extending 1.5 inter-quartile range (IQR) above and below the quantiles. So, they extend 1.5 times the difference between the quantile above the mean and below and must end on an observation. For more details on understanding a box plot, see [26]. A smaller box and whiskers are generally positive since they indicate that the approximations vary less.

In this section, we will only plot how the approximation for expressional coverage changes with an increasing amount of seeds. The mean and standard deviation approximation for device A are shown in Figure 5.3. At the same time, for device B they are shown in Figure 5.4. From both Figure 5.3a and Figure 5.4a, we can observe that the calculated mean for both devices starts to stabilize quickly, even for as low as five different seeds. However, the quantiles or boxes are still quite large, which indicates that even though the mean approximation is good, many outliers exist. Based on the approximated standard deviation plots in Figure 5.3b and Figure 5.4b, it seems harder to approximate with fewer seeds. The prediction starts to settle for ten and more seeds in the approximation. Based on the observations, 15 seeds strike a good balance between being accurate and not being too computationally expensive, so this amount will be used during testing. Including more seeds than this has minimal impact on the approximation.

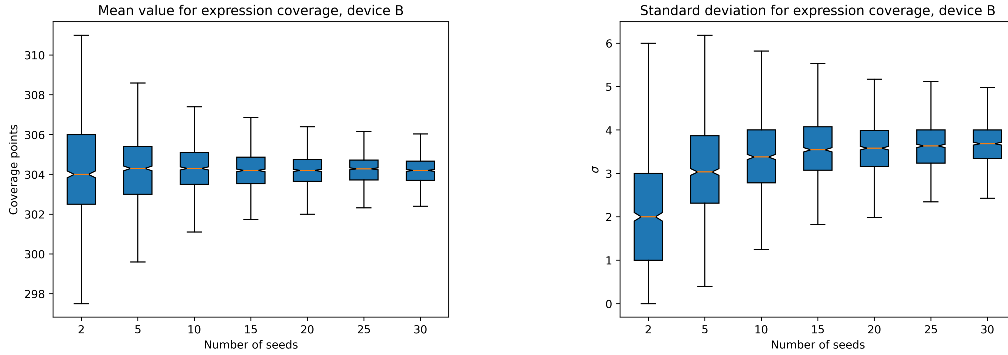


(a) Different amounts of seeds approximate the mean value for the expressional coverage distribution for device A.

(b) Different amounts of seeds approximate the standard deviation for the expressional coverage distribution for device A.

Figure 5.3: The approximation of standard deviation and mean value for expressional coverage for device A with different amounts of seeds.

Since we have concluded that using 15 seeds gives a good approximation for expressional coverage, based on the plots in Figure 5.3 and Figure 5.4, we would like to see how accurate they are for other coverage types. We calculate how accurate they are compared to the calculated mean and standard deviation from 1000 seeds to do this. We have calculated the mean and standard deviation based on 15 seeds 1000 times and taken the average of this. Then calculate how accurate this prediction gets compared to the mean and standard deviation calculated based on 1000 seeds. The results from this are found in Table 5.1 for both devices. We can see that the mean prediction percentage accuracy for all coverage types is above 95%. Also, the standard deviation is above 90% for all except statement coverage for device B. If we look back at the distribution of statement coverage in Figure 5.2b, it appears to be only at one value, but it has probably also hit 605 bins once if this value is selected. So getting one value that is not 606 when calculating the standard deviation for 15 seeds results in a big difference if the true value is close to 0.



(a) Different amounts of seeds approximate the mean value for the expressional coverage distribution for device B.

(b) Different amounts of seeds approximate the standard deviation for the expressional coverage distribution for device B.

Figure 5.4: The approximation of standard deviation and mean value for expressional coverage for device B with different amounts of seeds.

Table 5.1: Calculated mean and standard variance accuracy for devices A and B for 15 samples over the true values for 1000 samples.

Accuracy[%]								
Cover type	Expressional		Statement		Toggle		Branch	
Device	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
A	99.95	94.97	99.98	96.19	99.960	93.42	99.96	95.70
B	99.99	95.8	100	7.49	100	95.47	99.99	91.02

5.2 Determining the number of tests

This section aims to determine the optimal number of transactions for each device to do the initial investigation. Throughout this first part, we wish to test many different configurations for each method; because of this, we want to run as few transactions as possible to minimize the time spent. We will examine how random testing scales with increasing transactions to select the optimal number. We wish to find the lowest point where random testing stops scaling, as this should be a point where improvements are possible. Selecting the point above where random testing scales well ensures we observe less variation in the results due to random variation.

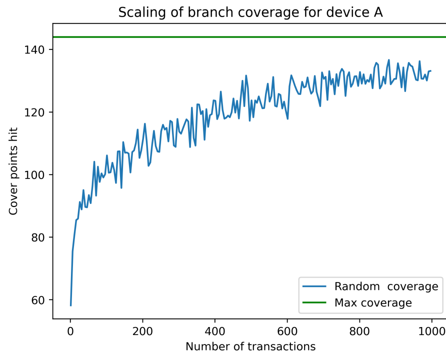
The scaling of random testing will be determined by applying a gradually increasing number of transactions and recording the coverage. Based on this, we will get a plot that on the x-axis will contain the number of transactions applied while the y-axis will be the number of bins hit for a cover type. We will evaluate transactions between 1 and 1000. The results for each device are presented separately in subsection 5.2.1 and subsection 5.2.2.

5.2.1 Device A

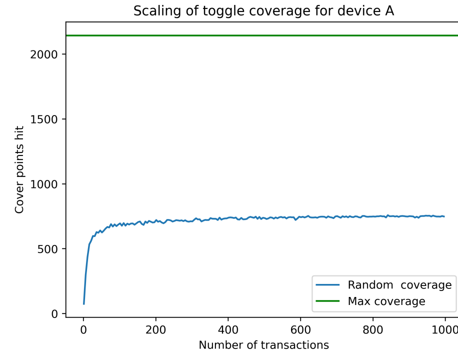
The scaling of random testing for device A for all coverage types is shown in Figure 5.5. From this, we can observe that some coverage types scale better or worse. Branch and statement coverage is shown in Figure 5.5a and Figure 5.5d, has a consistent scaling from 1 transaction to 1000 transactions. It indicates that these coverage types are responsive to random testing and scale well for an increasing number of transactions.

Expressional and toggle coverage are on the opposite end of the spectrum, with poor scaling for an increasing number of transactions. They have an instant increase before slowly it drops off, and there is almost no increase. The plot of the increase for these two coverage types can be found in Figure 5.5b and Figure 5.5c. These exhibit properties that should make them good candidates

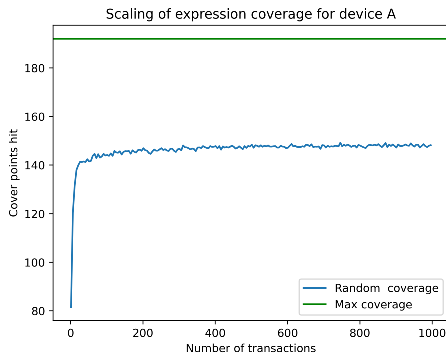
to benefit from machine learning. We should be able to use as few as 100 transactions for device A and still see potential improvement for different settings. So all settings tested will be run with this number of transactions. This will be explicitly stated if we test with another number of transactions.



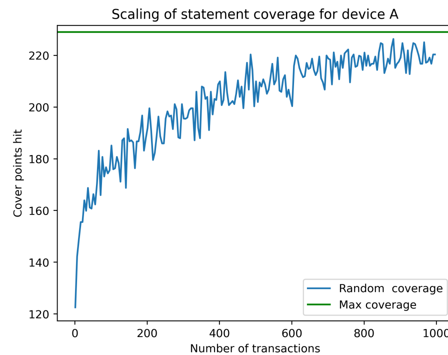
(a) The scaling of branch coverage for an increasing amount of transactions plotted for device A.



(b) The scaling of toggle coverage for an increasing amount of transactions plotted for device A.



(c) The scaling of expression coverage for an increasing amount of transactions plotted for device A.

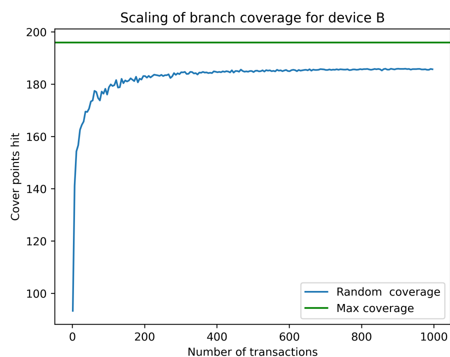


(d) The scaling of statement coverage for an increasing amount of transactions plotted for device A.

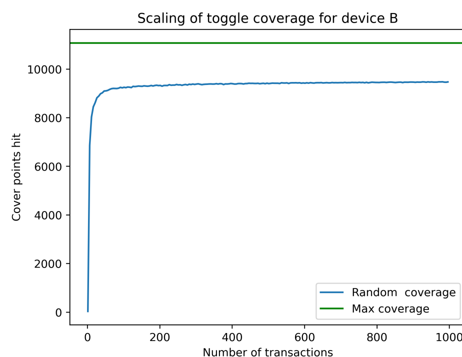
Figure 5.5: How coverage types scale with random testing for between 1 to 1000 transactions for device A.

5.2.2 Device B

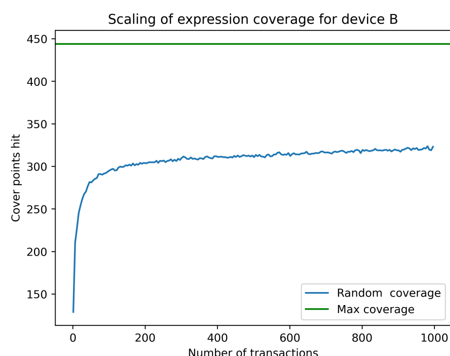
Here is a look at how random testing scales for device B, the methodology used, are the same as for device A. We can observe from the plots in Figure 5.6 that the scaling for the different coverage types is worse for this device than for device A. However, we get closer to the max achievable coverage for both toggle, statement, and branch. From Figure 5.6a and Figure 5.6b, we can see very little increase passed even the first 50 tests. So our random transactions are quickly stagnating for these cover types. The statement cover for device B almost instantly hit all bins, which is almost to be expected based on the circuit's design; we can see this in Figure 5.6d. Expressional coverage in Figure 5.6c is an outlier, with the biggest gap up to max coverage. However, it has a slow but steady increase, indicating there should be potential for improvement. Based on how random testing scales, 200 transactions look to be a good number where none of the coverage types still scales well. For testing different settings for the ML framework on device B, we chose to use 200 transactions.



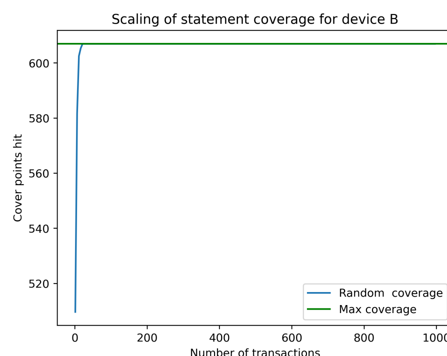
(a) The scaling of branch coverage for an increasing amount of transactions plotted for device B.



(b) The scaling of toggle coverage for an increasing amount of transactions plotted for device B.



(c) The scaling of expression coverage for an increasing amount of transactions plotted for device B.



(d) The scaling of statement coverage for an increasing amount of transactions plotted for device B.

Figure 5.6: How coverage types scale with random testing for between 1 to 1000 transactions for device B.

5.3 Parameter tuning

This section will examine how different settings for Bayesian optimization and the Machine learning framework might affect the results. Finding these optimal parameters for both devices gives the machine learning framework its best chance compared to other methods. For each setting, we wish to find the best one to construct the most optimal configuration of Bayesian optimization. A setting is more optimal if it has a higher mean value measured in coverage bins hit and a p-value below 0.005; the change is statistically significant. Using each optimal setting will then construct a more optimal overall configuration.

5.3.1 How to read the results

This section is a short introduction to reading and understanding the results. The results in this section will be presented in tables requiring more careful reading. As stated in the introduction to this section, our primary goal is to improve the average coverage generated across some number of seeds for a fixed amount of transactions. However, our measurements in the table are the average coverage across the 15 seeds and the standard deviation. These values are needed to use Welch's t-test, which will give us the p-value indirectly. A baseline or default setting will be picked for each setting we test. This setting is an arbitrary choice, but it allows us to compare settings and calculate the p-value compared to this. It is essential to state that the Baseline setting is not necessarily the worst.

So when reading these tables, two values are important to keep an eye on the mean and p-value. First, look at the mean if this is higher than the baseline, then at the p-value. If the p-value is lower than 0.05, the improvement would be observed less than 5% of the time due to random variations. Such a measurement should be significant and not just an improvement due to random luck. If the p-value is higher than the threshold, we can safely assume that the setting results in the same amount of coverage and differences are due to random effects. A setting can also result in a mean below the baseline with a p-value under the threshold, which means it negatively affects coverage. The standard deviation can be examined to determine which test generates the most consistent coverage but will not be weighted heavily in our results.

5.3.2 Device A

Device A has a very limited amount of settings to change. We have only two settings to change for this device: the bounds and acquisition function.

Bounds

Here we will look at how changing the bounds for the Bayesian optimization affects the result. Bayesian optimization will always get a lower bound of 0, but the higher one will vary. This is because SystemVerilog can not handle negative numbers in a dist; they can, at minimum, be zero, meaning they are a constraint. So, when randomized, the signal will never be set to a value set to zero in the dist. Increasing the upper bound should allow the system to provide a finer resolution to the weights in the SystemVerilog testbench. If the upper bound is ten, each weight can be any integer number from 0 to 10, which limits how finely the ratio between specific values can be set. When increasing this to 100 or 1000, the ratio can be finer and potentially get closer to the optimal, so we expect a higher upper bound to get better results on average. However, it will also result in more potential ratios, so it might be more challenging for Bayesian optimization to find the right one.

We have tested four different upper bounds 1, 10, 100, and 1000. The results from this testing are presented in Table 5.2. Generally, the effect is as expected, where a higher upper bound looks beneficial. However, it does not result in a p-value below the threshold before the upper bound of 100. We only observe a p-value below 0.05 for expressional coverage, which has an improvement of 1.6 extra bins hit on average. Nevertheless, both statement and branch coverage has an improvement, but because of the bigger standard deviation of these coverage types, we can not conclude that they have improved. Either way, the results are the same, and the upper bound of 1000 gives Bayesian optimization the greatest chance of finding the optimal ratio between the weights.

Table 5.2: The results of testing different bounds for Bayesian optimization on device A.

Coverage				
Bounds configuration	Coverage type	Mean	Standard deviation	P-value(over $[0, 1]$)
$[0, 1]$	Expression	144.13	3.23	1
	Statement	171.53	22.13	1
	Toggle	678.09	50.39	1
	Branch	96.72	14.05	1
$[0, 10]$	Expression	145.04	3.47	0.0996
	Statement	172.33	22.50	0.8265
	Toggle	676.69	39.38	0.8499
	Branch	97.73	14.37	0.6630
$[0, 100]$	Expression	145.21	3.52	0.0519
	Statement	176.23	23.88	0.2126
	Toggle	648.2	44.35	0.0002
	Branch	99.91	15.43	0.1882
$[0, 1000]$	Expression	145.73	2.78	0.0014
	Statement	175.32	18.87	0.2613
	Toggle	675.27	39.46	0.7027
	Branch	99.56	11.85	0.1831

Acquisition function

In this section, we have tested all the supported acquisition functions for the Scikit-Optimize library. The supported functions are Expected improvement (EI), Probability of improvement (PI), and Lowe confidence bound(LCB) [20]. When testing according to our methodology, we get the results in Table 5.3. We can see that LCB, on average, performs worse for all coverage types even though it does not come under the p-value threshold. EI and PI are equally matched for all but toggle coverage, where PI hits an extra 30 bins with a p-value under 0.05. Because of this, we will prefer to use PI because of the better coverage of toggle cover points. These results show that the Gaussian process gives a bad approximation of the coverage numbers as a function of the weights. EI would be preferred if this approximation was better because it considers where the potential improvement is biggest and how likely it is. While PI finds the area where the Gaussian process is most certain, there is an improvement. So, it can better predict if a set of weights is an improvement over another without saying something about the size of the improvement.

Table 5.3: The results of testing different acquisition functions for Bayesian optimization on device A.

Coverage				
Acquisition function	Coverage type	Mean	Standard deviation	P-value(over EI)
Expected improvement (EI)	Expression	145.21	3.52	1
	Statement	176.24	23.88	1
	Toggle	648.2	44.35	1
	Branch	99.91	15.43	1
Lower confidence bound (LCB)	Expression	144.64	3.56	0.3228
	Statement	170.45	19.89	0.1090
	Toggle	635.39	45.79	0.0837
	Branch	95.88	12.84	0.0845
Probability of improvement (PI)	Expression	145.8	3.19	0.2862
	Statement	176.91	19.82	0.8526
	Toggle	680.49	35.74	0.0000
	Branch	100.79	12.72	0.7038

5.3.3 Device B

For device B we have constructed two testbenches. However, we will assume that optimal bounds and acquisition function settings for testbench 1 are the same as for device A's testbench. So this section will only test settings related to testbench 2, where the implementation and overview are shown in subsection 4.3.8. We have more settings and configurations to change for this testbench as it uses a neural network to perform a projection on the output from Bayesian optimization. We will focus on testing two aspects of the neural network which might affect the results the size of the ranges and the network complexity. The size of ranges focuses on determining how many numbers should be uniformly distributed, while network complexity is about how many layers the neural network consists of and their width.formly distributed, while network complexity is about how many layers the neural network consists of and their width.

Bounds

In this section, we explore how different bounds affect the achieved coverage for device B. Testbench 2 has a different connection between the bounds and the weight used. The bounds here affect which values the input to the neural network can have; the neural network's output is how the different values are weighted. Since the Softmax activation function will force the output to be a probability distribution, the input should not matter too much. We expect little difference between the bounds; for the neural network, each input can be a floating point number. So, Bayesian optimization has much more control even for a lower bound since the values can be much finer controlled than when we required integers for device A.

We have tested four different bounds, and they are all symmetrical around zero since there are no constraints that they can not be negative. There results from this testing are shown in Table 5.4. We can see that even though the mean of coverage bins hit some increases when moving to higher bounds, the p-value are not below 0.05 for other than expressional coverage for $[-100, 100]$. Here there is a slight increase with a p-value lower than 0.05, which indicate that this set of bound produces the best-weighted distribution when used as input for the neural network. Based on this, we will use the $[-100, 100]$ as bounds going forward.

Table 5.4: The results of testing different bounds for Bayesian optimization on device B.

Coverage				
Bounds configuration	Coverage type	Mean	Standard deviation	P-value(over $[0, 1]$)
$[-1, 1]$	Expression	303.76	3.54	1
	Statement	605.93	0.25	1
	Toggle	9311.56	52.23	1
	Branch	182.6	1.02	1
$[-10, 10]$	Expression	304.49	3.01	0.1739
	Statement	605.93	0.25	1
	Toggle	9313.27	50.79	0.8395
	Branch	182.57	1.02	0.8731
$[-100, 100]$	Expression	305.17	3.55	0.0158
	Statement	605.93	0.25	1
	Toggle	9321.05	54.19	0.7479
	Branch	182.55	1.01	0.1882
$[-1000, 1000]$	Expression	304.64	3.45	0.1254
	Statement	605.93	0.25	1
	Toggle	9316.85	49.57	0.5253
	Branch	182.48	1.22	0.5136

Acquisition function

We have tested how testbench 2 for device B responds to the acquisition function found in Scikit-Optimize. These results are found in Table 5.5. From these results, we can observe very little difference between acquisition functions. It could indicate that the Gaussian process has high uncertainty for all inputs, with no idea which input to the neural network results in a probability distribution that might produce higher coverage. We will better understand if this is the case when comparing the method to other methods in subsection 5.4.2. We will keep the acquisition function as Expected improvement (EI) for now.

Table 5.5: The results of testing different acquisition functions for Bayesian optimization on device B.

	Coverage			
Acquisition function	Coverage type	Mean	Standard deviation	P-value(over EI)
Expected improvement (EI)	Expression	303.76	3.54	1
	Statement	605.93	0.25	1
	Toggle	9311.56	52.23	1
	Branch	182.60	1.02	1
Lower confidence bound (LCB)	Expression	303.72	3.52	0.9448
	Statement	605.93	0.25	1
	Toggle	9311.56	52.23	0.9975
	Branch	182.60	1.02	1
Probability of improvement (PI)	Expression	303.83	3.36	0.9061
	Statement	605.93	0.25	1
	Toggle	9311.53	52.81	0.9975
	Branch	182.60	1.02	1

Range size

We will, in this section, test different sizes of output ranges. The ranges sizes determine how many values are assigned the same probability. A larger range requires a smaller projection and, thus, a smaller neural network since it requires fewer perceptrons in the output layer. A smaller neural network will require less computation since fewer multiplications and additions are needed. Allowing a smaller range size might be beneficial since it can potentially finer control the numbers used as input to the DUT. However, it may be harder to find the correct ranges since there are so many possibilities to select from. A larger rage size might be beneficial if the DUT is less sensitive to individual values. We expect it to prefer more values in one range based on the specification.

The results of testing six different range sizes are presented in Table 5.6. All configurations are compared to the base configuration with 128 numbers in the range. We observe a slight increase in the mean value for different coverage types when increasing the range size. However, there is only 4096, which achieves a p-value of less than 0.05; this aligns with our expectations. These results are interesting as they show that device B is not this sensitive to which number it gets as an input. We could potentially go for even more values in one range, as we have not yet crossed the threshold where the ranges are getting too large.

Table 5.6: The results of changing the number of values assigned uniform probability in the weighted probability distribution.

Coverage				
Range size	Coverage type	Mean	Standard deviation	P-value(over 128)
128	Expression	304.04	4.20	1
	Statement	605.93	0.25	1
	Toggle	9319.41	42.97	1
	Branch	182.53	0.88	1
256	Expression	303.76	3.54	0.4924
	Statement	605.93	0.25	1
	Toggle	9317.51	45.35	0.7875
	Branch	182.53	0.88	0.9271
512	Expression	304.73	4.68	0.3411
	Statement	605.93	0.25	1
	Toggle	9321.75	42.55	0.7329
	Branch	182.53	0.88	0.9271
1024	Expression	305.25	3.83	0.0665
	Statement	605.93	0.25	1
	Toggle	9321.95	42.97	0.7124
	Branch	182.53	0.88	0.9271
2048	Expression	303.76	3.54	0.6597
	Statement	605.93	0.25	1
	Toggle	9316.85	49.57	0.3076
	Branch	182.48	1.22	0.7344
4096	Expression	305.6	3.26	0.0121
	Statement	605.93	0.25	1
	Toggle	9311.56	52.23	0.9393
	Branch	182.6	1.02	0.7137

Projection optimization

This section will examine how different projections can affect the number of cover points hit. We do this by changing the number of perceptrons in the layers of the neural network. We could have done this by hand but will instead use Bayesian optimization to try and optimize the number of perceptrons in each layer. We will do what can be called meta optimization, where Bayesian optimization proposes the number of perceptrons in each layer. Based on this, we create a neural network to use as a projection explained in section 4.3.8. We then run the training loop presented in subsection 4.3.8 to optimize coverage results with the given projection. If the projection makes the inner loop find better coverage results, this projection is preferred. Using this allows us to explore the space of possible projections efficiently. So projections might work better because they better preserve the structure of the problem. Bayesian optimization through Scikit-Optimize was run for 40 iterations, so a total of 40 projections was evaluated. In Table 5.7, the best one found is presented.

The results of using Bayesian optimization to find a more optimal projection are compared with the simplest possible projection. This simple projection goes from 15 inputs to 1048576 outputs, which gives a range size of 4096. When using Bayesian optimization to find a more optimal projection, we will keep the input and output sizes the same but allow for five layers between them which can have varying sizes. Changing the number of perceptrons in each such layer allows for different mappings between the input and output, where one could be more beneficial. An example would be if the input weights were more linked to control the probability for numbers associated with higher coverage.

The best projection found using the described method is presented in Table 5.7. The configuration tells us how many perceptrons are used in each layer. When looking at the mean values for all coverage types revealed that it does not improve over the simple projection. Also, the p-value

tells the same story as it is higher than 0.05, which was set as the threshold for the result to be significant. The p-value is calculated compared to the base configuration, which is the projection from 15 → 1048576.

Table 5.7: Results of preforming a meta-optimization to find an optimal projection.

Coverage				
Projection configuration	Coverage type	Mean	Standard deviation	P-value
15 → 1048576	Expression	305.6	3.26	1
	Statement	605.93	0.25	1
	Toggle	9311.56	52.23	1
	Branch	182.6	1.02	1
15 → 703 → 1024 → 16 → 334 → 615 → 1048576	Expression	305.53	4.51	0.9134
	Statement	605.93	0.25	1
	Toggle	9314.38	49.91	0.7358
	Branch	182.73	0.85	0.3979

5.4 Comparison with other methods

Until now, we have only evaluated how different settings affect the coverage generated by the machine learning testbench. This section will compare the best configuration of Bayesian optimization for each device with random and random constraint testing. We will compare these methods for different numbers of transactions to see if the results are consistent.

Our base configuration is random testing since this requires no extra work from the verification engineer. For random testing, we have done nothing other than provide constraints for stimuli to the DUT to ensure it is valid. While for random constrained testing, weights are assigned to values deemed important in generating more coverage so they appear more frequently in the input. Random constraint involves extra work for the verification engineer because important values and stimuli must be identified by looking at the code or specifications; this makes it more time-consuming and challenging. Ideally, we want our Machine learning framework to perform as well as random constraint since it could free up the time the verification engineer uses to optimize testbench weights. For this project, the author has replaced the verification engineer and chosen the weights for the random constrained testbench. The focus has been on finding the right ratio between the weight rather than taking the time to fin-tuning their values.

5.4.1 Device A

All three verification methods presented in the previous section are used to test how they compare against each other in hitting all coverage bins for device A. These will be tested on three different numbers of transaction items, starting with 100, 1000, and 100 000. We expect random constraint to be the best method, then the Machine learning testbench, before random in terms of hitting coverage bins.

The coverage achieved after applying 100 transactions to device A is found in Table 5.8. From these, we can see that random and ML are nearly identical for all coverage types. The only coverage type with a p-value below 0.05 is toggle coverage, with a drop in mean value from 682.91 to 665.31. There is a less than 4% chance we will observe this drop in coverage due to random variation, so we can be pretty sure the ML testbench has worsened the toggle coverage. The random constrained testing with weights selected by the author of this thesis has a huge leap for most coverage types. It can, on average, increase the number of branch cover points hit by more than eight bins. Toggle coverage gets increased with 24 extra cover points hit. Finally, state men have an increase of approximately 11 cover points. Also, expressional coverage almost hit an extra two bins on average. All coverage increases are also statistically significant, with a p-value below 0.05. The results align with what we expected, but the machine learning testbench did worse than expected, as it should have improved a bit over random.

Table 5.8: Coverage results for different verification methods for 100 transactions on device A.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	144.46	3.43	1
	Statement	175.44	25.82	1
	Toggle	682.91	51.07	1
	Branch	99.71	17.01	1
Random constrained	Expression	146.33	2.60	0.0002
	Statement	186.64	23.82	0.0065
	Toggle	708.22	35.49	0.0006
	Branch	108.16	16.01	0.0021
ML testbench	Expression	145.25	3.15	0.1457
	Statement	175.17	20.25	0.9431
	Toggle	665.31	52.86	0.0398
	Branch	99.44	13.13	0.9131

We can see less difference between the methods when looking at the results for 1000 transaction methods in Table 5.9. Some of this is probably due to us reaching a saturation point for coverage; both branch and statement coverage has only a handful of coverage points missing to be fully covered. The only coverage type that random constraints can improve meaningfully is statement coverage, with an increase in almost four cover points over random; thus, this is the only one with a p-value below 0.05. When looking at the ML testbench, we see that it is slightly worse in measured mean coverage values for all cover types, but none has a p-value below 0.05. Because of the p-value, we can not necessarily say it is worse since it is possible to observe all these mean values in more than five percent of cases. The correct classification is that the ML testbench and Random are equal with a slight advantage to Random constrained for 1000 transactions, so no benefit is returned from the time spent on training.

Table 5.9: Coverage results for different verification methods for 1000 transactions on device A.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	148.66	1.61	1
	Statement	221.31	11.25	1
	Toggle	751.16	12.14	1
	Branch	133.36	7.97	1
Random constrained	Expression	148.82	1.70	0.5658
	Statement	225.31	6.77	0.0094
	Toggle	749.77	11.64	0.4792
	Branch	136.64	4.83	0.0027
ML testbench	Expression	148.58	2.07	0.7915
	Statement	217.69	14.80	0.0941
	Toggle	748.59	13.48	0.2221
	Branch	130.56	110.83	0.0740

The coverage results after 100 000 transactions are shown in Table 5.10. We can see that the testbench and all methods have reached their saturation point. There is virtually no difference between the three methods; they all produce the same amount of coverage with all p-values above 0.05 except branch coverage for random constraint. However, this is because both distributions have a tiny standard deviation, and since the coverage is discrete, we can not trust the p-value for such a small standard deviation. For device A, none of the methods is significantly better for 100 000 transactions. This indicates it is too many to see any difference between the methods, and we have probably reached the limit of the coverage points the test can reach.

Table 5.10: Coverage results for different verification methods for 100 000 transactions on device A.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	148.89	1.46	1
	Statement	227.97	0.15	1
	Toggle	760.98	0.15	1
	Branch	142.49	0.50	1
Random constrained	Expression	149.0	1.45	0.6407
	Statement	227.98	0.15	1
	Toggle	760.98	0.15	1
	Branch	142.18	0.44	0.0008
ML testbnech	Expression	148.96	1.88	0.7963
	Statement	227.93	0.25	0.1866
	Toggle	760.93	0.25	0.1866
	Branch	142.41	0.45	0.3323

5.4.2 Device B

We will utilize all the methods presented in the introduction for device B. In addition, we will also compear it with testbench 1 for device B which involves more labor but is expected to have higher potential since the verification engineer already selects the important values. Tunning the weight of these chosen values is guaranteed to have an almost immediate effect on coverage. There should be more time to fine-tune the weights since it does not need to determine which values are important first. In the second place, we expect it to be randomly constrained as the verification engineer has a great opportunity to select the right ranges for the size 32-bit float numbers. It would be great for Testbench 2 to get as close as possible to random constraints; this would be a good result. When ranking methods, random should end up last as it is not actively doing something to give the device a more optimal set of stimuli to the DUT. We will also test how the different methods perform for 200, 2000, and 100 000 transaction items for device B.

We start by comparing the methods for 200 transactions with the results found in Table 5.11. When looking at the mean value for certain coverage types, we can see a substantial improvement for both random constraint and ML testbench 1, especially in expressional coverage. There is an 11 and 14 cover point increase corresponding to 3.6% and 4.6% improvement over random. However, there is a cost related to this improvement which comes in the form of decreased toggle coverage. This is expected as the two coverage types have opposite requirements, with toggle wanting more spread in the 32-bit input numbers and expressional wanting less. Since most of the computation inside the device only happens if the numbers are inside a range and have a certain relationship, triggering this requires us to toggle the signals less. It is worth noting that the ML testbench 1 also has improvements compared to random constraints for branch coverage. For ML testbench 2, we see almost zero improvements compared to random testing, expressional coverage has an 0.3 cover point increase on average, but the p-value tells us this observation is not significant. Other things to highlight are that statement and branch coverage are slightly worse, with a p-value of less than 0.05, meaning there is a less than five percent chance this is due to random variations. Toggle coverage remains at the same level as for random while being better than random constraint and ML testbench 1, but this is expected due to the much lower expressional coverage. Our prediction was quite accurate, and testbench one shows promising results compared to random constrained.

Table 5.11: Coverage results for different verification methods for 200 transactions on device B.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	304.6	3.71	1
	Statement	606.98	0.15	1
	Toggle	9323.04	47.87	1
	Branch	183.15	1.63	1
Random constrained	Expression	315.55	3.59	0
	Statement	606.98	0.15	1
	Toggle	9245.58	16.59	0
	Branch	180.4	1.22	0
ML testbench 1	Expression	318.46	4.67	0.0000
	Statement	605.98	0.15	0.0000
	Toggle	9238.08	18.25	0.0000
	Branch	183.69	1.56	0.0426
ML testbench 2	Expression	304.09	3.27	0.3760
	Statement	605.93	0.25	0
	Toggle	9314.73	43.69	0.2686
	Branch	182.72	0.90	0.0453

The trend observed for 200 transactions continues for 2000 for device B, with the results found in Table 5.12. Again, looking at expressional coverage for both random constraint and ML testbench 1, we observe a substantial improvement. With an increase of 14 and 18, extra cover points hit, resulting in a 4,3 and 5,3 percent improvement. So, the improvement over random increased when compared to 200 transactions. It does come at the cost of toggle coverage, which loses many cover points compared to random. ML testbench 1 improves slightly over random constraint measured in toggle coverage while improving expressional coverage. Again ML testbench 2 looks to be on par with random verification when comparing mean values and is way worse than testbench 1 and random constraint. So, the fully ML-enabled testbench 2 cannot find the correct ranges for the 32-bit numbers. It was expected it would struggle, but not to the degree it shows in the results. We expected to observe at least some improvement over random with a p-value smaller than 0.05.

Table 5.12: Coverage results for different verification methods for 2000 transactions on device B.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	331.64	4.65	1
	Statement	606.98	0.15	1
	Toggle	9510.60	11.59	1
	Branch	185.98	0.15	1
Random constrained	Expression	345.96	4.42	0.0000
	Statement	605.98	0.15	0.0000
	Toggle	9353.82	22.74	0.0000
	Branch	184.09	1.13	0.0000
ML testbench 1	Expression	349.28	4.89	0.0000
	Statement	605.98	0.15	0.0000
	Toggle	9384.22	35.65	0.0000
	Branch	184.62	0.48	0.0000
ML testbench 2	Expression	330.2	5.50	0.0845
	Statement	605.93	0.25	0.0000
	Toggle	9508.07	13.99	0.2292
	Branch	184.93	0.25	0.0000

The general trend continues when increasing the number of transactions to 100 000. Expressional coverage increased with 15 cover points for random and ML testbench 1; the ML testbench 1

lead has disappeared. We can read this from the results in Table 5.13. These numbers led to an improvement of 4% over random, which is lower than for 2000 transactions. This drop in lead might indicate that we have started reaching the test limits; we can also observe this when looking at the standard deviation for all methods. When the standard deviation becomes lower, it indicates less difference in coverage across different seeds. When this approaches zero, it tells us all seeds hit the same amount of cover points, and we start to see the limits of the test. It is not a direct problem for the ML methods but tells us that the testbench could be more optimal. Finally, ML testbench 2 is also on par with random for 100 000 transactions. So it lags quite a bite between Testbench 1 and random constraint testing measured in expressional coverage. For device B, Testbench 1 provides a potential benefit for fewer transactions, with the advantage disappearing for a larger number of transactions. It also appears favorably compared to random constrained testing with a one percentage point lead. Testbench 2 has no improvement over random and is a more expensive way to generate random tests.

Table 5.13: Coverage results for different verification methods for 100 000 transactions on device B.

Coverage				
Testbench / Methode	Coverage type	Mean	Standard deviation	P-value(over Random)
Random	Expression	360.06	1.65	1
	Statement	606.98	0.15	1
	Toggle	9629.89	0.43	1
	Branch	185.98	0.15	1
Random constrained	Expression	375.15	2.12	0.0158
	Statement	606.98	0.15	1
	Toggle	9635.06	9.45	0.8395
	Branch	185.98	0.15	0.8731
ML testbench 1	Expression	374.73	2.17	0.0000
	Statement	605.98	0.15	0.0000
	Toggle	9640.93	0.33	0.0000
	Branch	184.98	0.15	0.0000
ML testbench 2	Expression	360.16	2.26	0.1254
	Statement	605.93	0.25	0.0000
	Toggle	9629.82	0.46	0.5253
	Branch	184.93	0.25	0.5136

5.5 Computational cost

This section will focus on how the different methods utilize compute time and measure the time used for a given number of transactions. We will also look at alternative ways compute time can be used by either generating smarter tests or spamming less smartly created tests.

5.5.1 Total Compute Time (TCT)

Total compute time (TCT) will measure how long each method uses to generate and execute a set number of transactions. We can see comparatively how long each method uses to process a fixed number of transactions. Since this thesis aims to optimize the computing time consumed by verification for a hardware design project, we prefer a lower time per processed transaction. However, it is not necessarily such that lower compute time per transaction is beneficial since it all depends on the coverage achieved. TCT solely measures each method's time to drive a fixed amount of stimuli through the DUT. And can be thought of as answering what is the best method to use if they all achieved similar coverage numbers.

We can see the results of running 200 000 transactions through each device in Figure 5.7. We can observe that each of the two methods for device A uses a similar amount of time. Since each

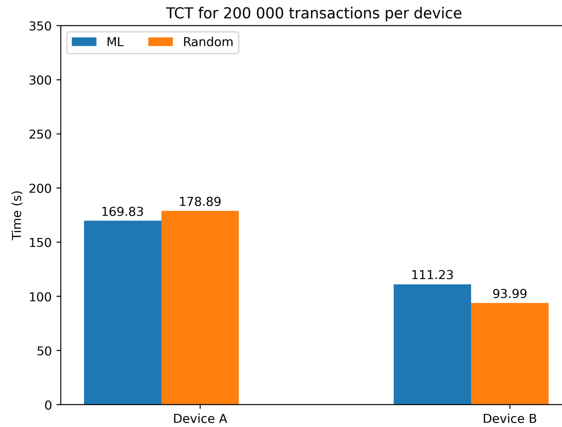


Figure 5.7: Total compute time measured for the two devices with 200 000 transactions.

method controls the delay between each package, the reason for the extra time used by random is probably that one average has a longer delay between responses or new commands than the ML testbench. Our ML testbench with Bayesian optimization has understood that the delay is unimportant for generating coverage and thus likely has a shorter delay.

The results for device B are presented in Figure 5.7, and we can observe a much larger difference in TCT. We have only included testbench 2 for this device, but testbench 1 should be approximately similar to random. The machine learning method uses around 1.2 times much time for the same amount of transactions. It is quite a considerable increase in the cost of producing each transaction. Most of this increase is because we passed the variables from Bayesian optimization through a neural network before drawing numbers from this randomly weighted distribution. All 32-bit numbers are written to a file and must be read by SystemVerilog; both create extra overhead. So, each transaction item takes an extra 20% to process, which is especially disappointing when considering the coverage numbers.

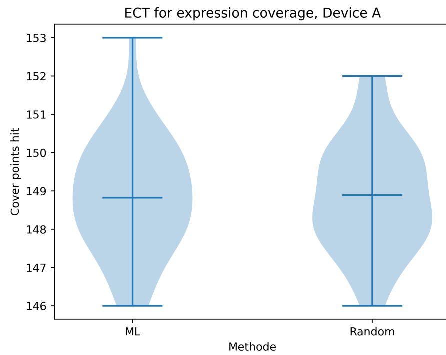
5.5.2 Equivalent Compute Time (ECT)

This section will measure the Equivalent Compute Time (ECT) for our two test devices, A and B. ECT measures how much coverage each method can achieve in a fixed amount of time, giving each method a budget of x number of seconds to run on the CPU and then observe the resulting coverage. Since some methods, as measured in subsection 5.5.1, use more time per transaction, they must be similarly more directed to produce the same amount of coverage. So the method with the highest coverage score for ECT will be the most efficient way to use the CPU hours. In general, this will measure how much more efficient one method is when not considering the time spent training the method or optimizing a testbench. So, it will be a good indication of the fixed cost of using one testbench over another.

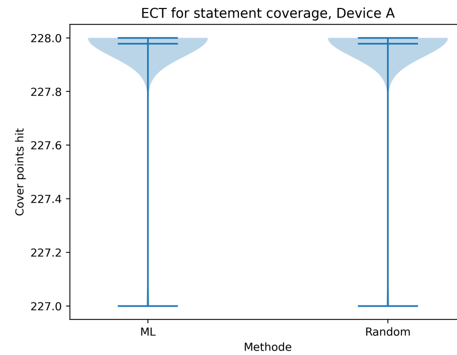
When testing ECT, we want to be in a region where it is still possible to achieve more coverage. Because of this, we would not like to allow each method to have more than 10 seconds of total computing time. Based on calculation from measurements in Figure 5.7, device A has time for 12 000 ML transactions and 11 000 random. Device B has time for approximately 18 000 ML transactions generated by testbench 1 and 21 000 random transactions. These numbers are found after considering some extra cost in starting a simulation and giving a result in actual runtime of around 10 seconds.

The result of allowing both methods ECT on device A are shown in Figure 5.8. The upper and lower horizontal lines mark the highest and lowest observations measured in coverage points hit. At the same time, the line between these is the average coverage across all runs. The size of the blue field marks the relative amount of observation that hit the corresponding number of cover points on the y-axis. For this device, we can see that both statement and toggle coverage is identical from Figure 5.8b and Figure 5.8c. Both methods can hit all cover points for statement coverage. For

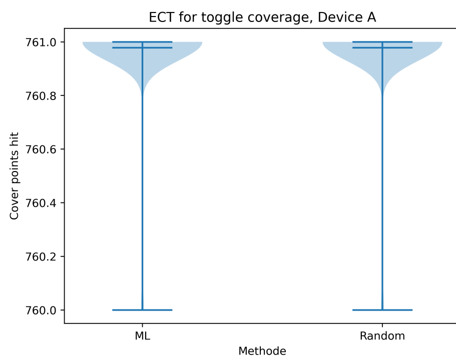
expressional coverage in Figure 5.8a, the result is that ML has a higher ceiling, but on average, the results are similar. Finally, for branch coverage, the coverage floor for random is higher together with a higher ceiling; this is shown in Figure 5.8d. However, on average, the difference is less than one cover point.



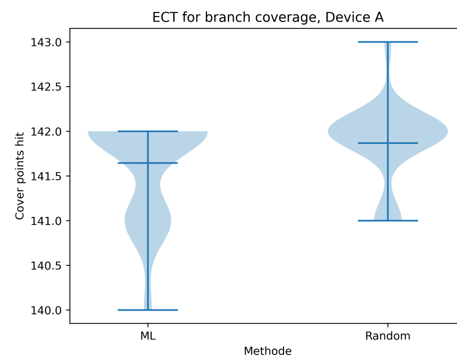
(a) The number of expressional coverage points hit with 10 seconds of ECT for device A.



(b) The number of statement coverage points hit with 10 seconds of ECT for device A.



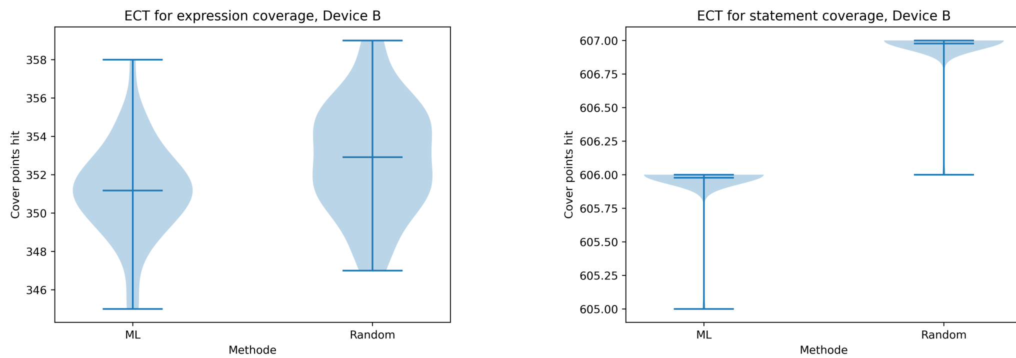
(c) The number of toggle coverage points hit with 10 seconds of ECT for device A.



(d) The number of branch coverage points hit with 10 seconds of ECT for device A.

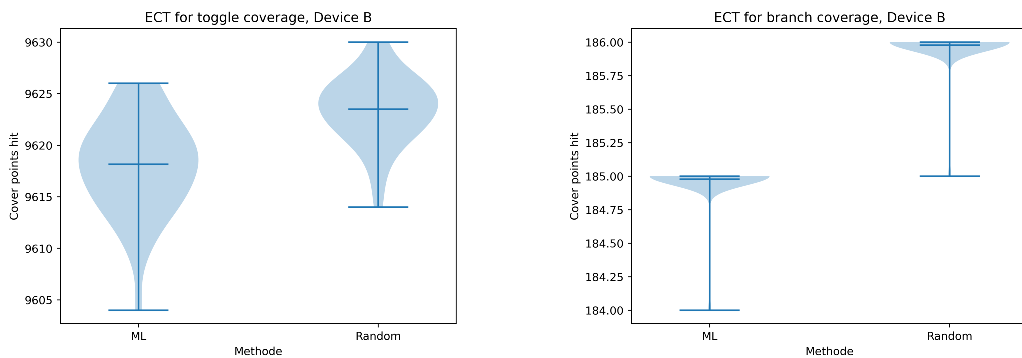
Figure 5.8: Measured ECT for all coverage types on device A.

We have only measured ECT for testbench 2 on device B, which is fully ML-enabled. The results of measuring ECT for device B are presented in Figure 5.9. In general, the results seem to favor random slightly, even though the difference for most coverage types is around one cover point. For statement and branch coverage shown in Figure 5.9b and Figure 5.9d, the difference between the methods is hitting all cover points or missing one in favor of random verification. The methods look closely matched for expression coverage shown in Figure 5.9a, but random still has the slight edge with about an extra cover point hit on average. Again, random has a higher floor, so even if we are unlucky, it expects to hit more cover points. The results for toggle coverage are shown in Figure 5.9c and have the biggest difference in the number of cover points hit between the two methods. For toggle coverage, the difference measured in cover points is around 10 for the lower higher, and the average number of cover points hit. So, the results align with what we can expect after comparing the methods in subsection 5.4.2. Since both methods were closely matched then, we expected random to pull away when allowed with some extra transactions.



(a) The number of expressional coverage points hit with 10 seconds of ECT for device B.

(b) The number of statement coverage points hit with 10 seconds of ECT for device B.



(c) The number of toggle coverage points hit with 10 seconds of ECT for device B.

(d) The number of branch coverage points hit with 10 seconds of ECT for device B.

Figure 5.9: Measured ECT for all coverage types on device B.

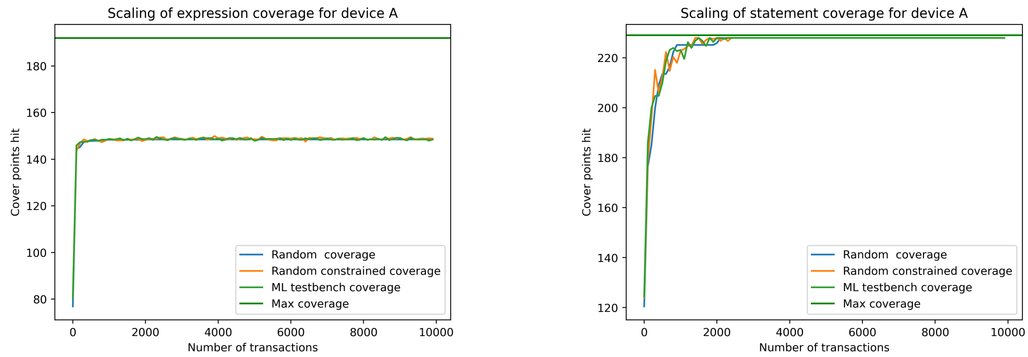
5.6 Scaling

This section looks at how the machine learning framework scales when allowed to produce an increasing number of transactions. We have done this by training the framework on a reasonably small number of transactions, 200. Taking what was found to be the optimal weights here, we will look at how these scales beyond these 200 transactions. To look at scaling, we have tested from 1 to 10 000 transactions with increments of 100. For each number of transactions, we have run 15 seeds and plotted the average coverage across these. In the plot, we have also included the max number of cover points as a horizontal line. The point of looking at this scaling is to determine if there is any real difference with a higher initial increase or better scaling as more transactions are added.

5.6.1 Device A

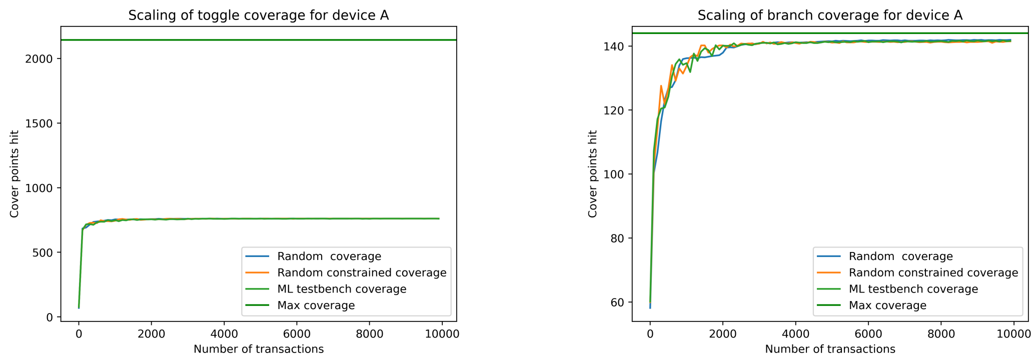
When looking at the coverage scaling of the different methods in Figure 5.10, we can see that the coverage number follows each other closely. Looking at the scaling of expressional coverage in Figure 5.10a, we see virtually no difference other than the ML testbench and random constrained being more wiggly, so it varies more. Toggle coverage found in Figure 5.10c is virtually identical between methods. Looking at statement coverage in Figure 5.10b, we see that full coverage is reached quickly for just over 2000 transactions. We can observe that random constraints have some big variations for less than 2000 transactions. Random constraint does not have the right weights when this happens since it indicates that the chance for important events is lower than it should be. Branch coverage is much of the same story as statement coverage. We see the branch

coverage for the device in Figure 5.10d. The number of coverage points hit for random constraint testing is slightly higher, with the ML testbench surpassing random for around 2000 transactions.



(a) The scaling of expressional coverage for device A.

(b) The scaling of statement coverage for device A.



(c) The scaling of toggle coverage for device A.

(d) The scaling of branch coverage for device A.

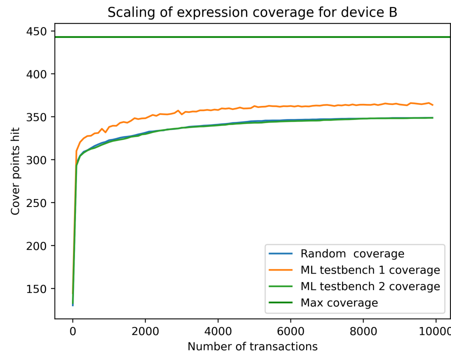
Figure 5.10: The scaling for the different coverage types between 1 and 10 000 transactions for device A.

From the results presented in Figure 5.10, we can see that the coverage bins hit as a function of the number of transactions scales similarly to random. There are no clear indications that the ML testbench has a more consistent scaling. Instead, we can observe that it is very similar to random constraint, where both have larger variations than random. In terms of more spikes and valleys, there can even be a drop in average coverage when more transactions are applied as stimuli. This is especially present when only a few transactions are applied to the DUT. The discussion section will explain why this might be the case. In general, both the ML testbench and random constraint provide little benefits over random, especially if the training and extra work going into these methods are considered.

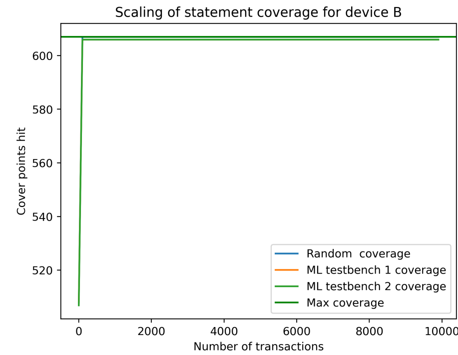
5.6.2 Device B

When looking at the scaling for different coverage types for device B, the result is presented in Figure 5.11. Random constraint testing is not included for this device as it was proven to be worse than testbench 1 in subsection 5.4.2. Thus we instead want to focus on comparing the two ML testbenches. We can observe that statement coverage is reached instantly for all methods, shown in Figure 5.11b. We expect this because the coding-style guidelines make statement coverage easy to hit. One interesting observation is the pore scaling of both branch and toggle coverage. We expected coverage types such as toggle coverage to scale better with more inputs when looking at Figure 5.11c, but it does not even for random. Scaling of branch coverage follows a similar trajectory in Figure 5.11d, with no difference between the methods and poor scaling for more

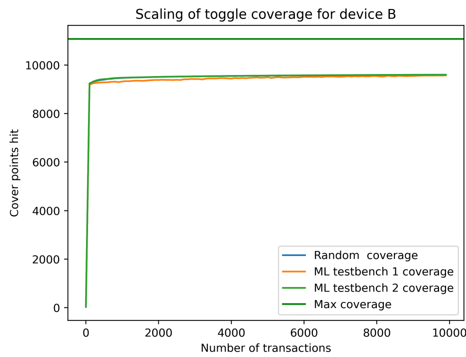
transactions. On the other hand, expressional coverage found in Figure 5.11a is interesting. Testbench 2 and random have a similar coverage number, which we found in subsection 5.4.2. But looking at testbench 1 we see a substantial improvement, but it does not necessarily scale better. It seems to be close to the same curve, just adjusted vertically. So this ML testbench looks most beneficial for a smaller number of transactions since it gives a good initial boost. We could observe the same in subsection 5.4.2 where the improvement was biggest for 2000 transactions over random and then decreased again measured in percentages. It aligns with our expectations, as all methods should have the same coverage ceiling.



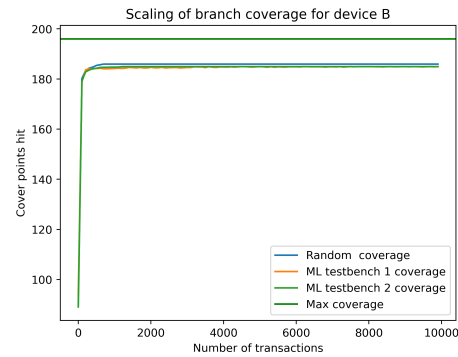
(a) The scaling of expressional coverage for device B.



(b) The scaling of statement coverage for device B.



(c) The scaling of toggle coverage for device B.



(d) The scaling of branch coverage for device B.

Figure 5.11: The scaling for the different coverage types between 1 and 10 000 transactions for device B.

Our testing found no huge scaling improvement using the ML testbench. However, testbench 1 gives a considerable initial boost over random. One thing to read from the graph is that testbench 1 can achieve the expressional coverage of 10 0000 randomly generated transactions with just 1700 transactions or less than five times as many. This shows the benefit of that initial boost, but the difference compared to random constrained would be smaller. Based on the results in Table 5.12, random constrained could produce the same coverage with around 2000 transactions. However, suppose one is willing to take the extra cost of both using a verification engineer's time and the cost of training. In that case, it can considerably improve over random with a smaller boost over random constrained. Using a fully automated such as testbench 2 is inefficient, and it is better to stick to random verification.

CHAPTER 6

DISCUSION

6.1 Coverage results

In section 5.4, we have seen that the coverage produced by the machine learning framework utilizing Bayesian optimization is lacking. For device B, there is a stark contrast between the two testbenches. No improvement was observed for testbench 2, while testbench 1 improved compared to all methods. So, based on our results, it looks like using a projection to scale the number of parameters that Bayesian optimization can control fails. We see that testbench 2 struggles with this and looks to have a close to uniform probability for all ranges. It might do that because weighing the heavier towards the wrong numbers results in even worse coverage. So, it has observed the best results for the close to a uniform distribution. Methods struggle with this because there is no gradient or indication of getting closer to a good value. One might solve this by running multiple seeds for each set of weights during training. A cover point can be hit four out of five times when running multiple seeds, so it could give a better indication back to Bayesian optimization that it moves the weight in the right direction. However, Bayesian optimization can handle noisy observations, which we can call running just one seed for each set of weights during training. Running multiple seeds might not be desirable because it will ultimately increase training costs, even though training time could stay the same since it could be run in parallel.

The results are better when looking at the results for testbench 1, which uses some human input to select the right ranges. We see a substantial improvement over random which is to be expected, but also over constrained random, which is good. It means that under the right circumstances, Bayesian optimization can better adjust the weight than a human. While we did not measure the TCT for testbench 1 on device B, it is expected to be the same as for random if we look at the results for device A which uses the same machine learning and testbench architecture. Using a system as proposed in testbench 1 has no apparent disadvantages other than some training costs. Using it might have benefits, but it will depend and is probably not much easier or less time-consuming than a verification engineer choosing the weights. A cost-benefit calculation should be performed to see if the one percentage point improvement is worth the extra cost of training the system.

As repeatedly stated during this thesis, testbench 1 for device B and the machine learning testbench for device A are built on the same principles. However, we can see that the results are quite different, where a considerable improvement can be seen over random for device B. Nevertheless, the improvement in coverage for device A is nowhere to be found; much of this can be attributed to the underlying testbench for device A. We have constructed a simple testbench for device A; the number of coverage points for this device is also fewer. It all contributes to there simply being little to optimize, and the difference from 1000 to 100 000 transactions only hits a couple more cover points. It cannot hit all cover points because the testbench does not allow it to enter an

alternative operating mode. It would require some special input and responses, so the transaction item would need to be slightly different, so it was not included.

6.2 Computational cost

Our goal for this thesis was to reduce the overall cost of verification. To do this, we either need to reduce the amount of manual labor or the compute time spent on verification. Measuring the amount of manual labor is hard, especially at a prototyping stage, so our focus has been on computing cost, which is easier to measure. We have focused on the time spent on inference since this should be a bigger part of the cost than training. The system is unlikely to save costs if training is a huge part of the total cost. One should ideally use such a system on smaller testbenches that are used often because of the initial boost, a system based on Bayesian optimization could potentially produce a boost in coverage. Also, targeting such testbenches should ideally make the training cost negligible because it is spread across enough testbench runs.

We see from our measurement of total compute time that saving weights to a file and importing and creating transactions based on this has a negligible compute cost. It is ideal as it allows the system to justify quite a modest boost in coverage since it does not introduce an extra cost of inference or running the testbench. They also do not start with a deficit of, say 10%, which they need to make up by being more than 10% more efficient per transaction. The only cost that must be justified is the training time which for Bayesian optimization should be quite small.

When using a projection to scale Bayesian optimization, we see a huge penalty for each transaction of approximately 20%. It makes this method start on the wrong foot since it needs to be more than 20% efficient in generating coverage per transaction to achieve an improvement. However, the speed of the implementation has not been a huge consideration, so it could likely be optimized. Even if one reduces the cost per transaction to only a bit more than random, the method does not make sense since it does not achieve any extra coverage. It would need to be on par with random constrained for it to provide a potential cost benefit since it could then reduce human involvement.

6.3 Problems with projection

We have already confirmed that a projection could work better for device B. The question then arises as to why it is not working. Firstly, the problem testbench 2 tries to solve is significantly more complicated than that of testbench 1. Where testbench 1 only needs to find the suitable weights, testbench 2 first needs to find the correct ranges and then assign them an optimal combination of weights. Assigning the wrong weight to the right range will make it seem unimportant, so it is hard. We also need to remember that all of this is done through a projection, making the weights not directly correlated to one range, so all values used as input to the projection affect the output.

It might also be that the problem maps badly to a projection or lies on something other than a manifold. If two coordinates are used to describe a point in a three-dimensional space, it is impossible to describe all points. Using a projection only allows us to describe a subset of the points in three dimensions since it maps all points in the two-dimensional space into a point in the three-dimensional space. However, it might be that our problem has no optimal solution for these points. This has likely happened for device B where the points we can describe are in the high dimensional space does not have the right weights assigned to the important ranges. Another device might produce better results when using a projection, but the method does not seem promising. It should be avoided for now or can be studied to find out for what kind of devices it could work.

6.4 Golden coverage

The term golden coverage encapsulates how many cover points a test can hit given what it can control. A test might not be able to cover all cover points, depending on which signal it controls. But it is also hard to answer which cover points it can hit. Determining if a cover point can be hit can be reduced to the boolean satisfiability problem (SAT), which is NP-complete [27]. Determining expression in an if statement can be true or false can be reduced to a boolean equation as a function of the device's input. This is exactly what the SAT problem is. Because of this, it is incredibly costly to compute if all cover points can be hit given the inputs the test controls. Since it is both costly and complex to determine the maximum number of cover points a test can hit, we have instead focused on comparative results.

When looking at the scaling on the methods for both devices, they are plateauing for most coverage types even if they have not reached 100% coverage. At this point, we could question whether the test can hit the remaining cover points or needs more control over the input. But the goal of this thesis is to see if machine learning could improve over other methods, and as such, the upper potential for each method has not been a concern. For device B, we can see that there is possible to hit more expressional cover points from testbench 1 and random constraint testing results. The results for device A are a bigger indication that we have reached the limits of our test since no method can improve significantly. Here it would be beneficial to have an already existing testbench for device A as a baseline where coverage holes have an explanation. In an industry application, we might be concerned with coverage holes; when comparing methods, it is not imperative to close coverage. All methods have the same amount of signals they can control for the DUT; as such, no method should be able to cover more coverage points than another. So, the results are valid and correct, but the testbenches are not necessarily optimal or have accounted for all edge cases. It could have been beneficial to take a deeper look into the coverage holes for device A where we see almost zero improvements for all methods in terms of scaling. If a thorough coverage analysis was done here, it might better explain why there was almost no improvement. We can observe that both random constrained and testbench 1 for device B can cover more expressional cover points, so we know the testbench's potential has not been reached.

6.5 Limitation of the testing

Our testing methodology has its limits around what we have tested and how thoroughly it has been tested. The first thing to consider is that no two devices are the same, so the result found in this thesis might not apply to another problem. It is all about the relationship between the weights controlled by Bayeasin optimization and the coverage. This relationship might also change during development, so a set of weights that worked well initially might be suboptimal in the end.

Most testbenches are typically used for actively developed designs, which keeps their external interface. However, the inner functionality can be different when weighing tests; it is inventible to direct tests toward one part of the design; if this part suddenly contains less functionality and thus fewer cover points, this might be bad. Trying to estimate how underlying design changes could require the system to be retrained is important to get a complete picture of the cost. Also, this thesis does not consider getting an estimate of potential coverage reduction during design. These experiments could be done by using the git history to evaluate the efficiency of the testbench at different stages in a project.

We have in this thesis only tested the methodology on two devices, so there is an amount of data to conclude from. This again relates to the waste difference between devices, and there are good reasons to believe the results will vary between devices. The two test devices are also quite small and too small to be very interesting to use the system on. It was a choice made to ensure that the testbench needed was not too complex and intricate to make. However, it makes it hard to say if variations in results are due to the complexity of device size. A larger selection of devices would better indicate the system's limitations.

Our testing does not contain any efforts to explore how the results scale with training time. If

allowing Bayesian optimization to run for more iterations has a good return in the form of better weights. We have not varied this parameter and kept it constant at 100 iterations which is quite many for Bayesian optimization [24]. This is again an evaluation related to training costs which have yet to be a priority to answer in this thesis.

Most of the limitations in the testing are related to potential training costs and use with other devices. As stated, the cost of training and what is reasonable will depend very much on the use case and the other options. These considerations need to be made for it to make sense in industry application, but the foundation remains the same; it needs to improve coverage. Furthermore, we have provided a detailed analysis of the potential improvements in coverage.

CHAPTER 7

FUTURE WORK

In this thesis, the work has concentrated on using Bayesian optimizations to create a better distribution of transactions. Based on the results found in chapter 5, there is still quite a long way to go to get to a state where this might be usable. The main hurdle which needs to be solved before it can be widely adopted at a large scale is the number of weights it can control. If we can not scale this, circuits of this size might be better verified using other methods, or an engineer would need to select which signals were important to optimize carefully.

Besides scaling Bayesian optimization to a higher dimension, it would be a huge benefit if the machine learning network could digest more information. Much information is discarded as there is no way to ingest information about the design or specifications with Bayesian optimization. Not utilizing this information that companies already have makes the problem harder on purpose. An excellent first step would be to enable a system to get information about which bins were hit and not by the simulation. By getting information about this as well as the structure of the code, a pattern between input weights and part of the code exercised might be detected. A good approach for an ML system should be to get more information from existing sources rather than to simulate a bunch and look at the result. These might become increasingly expensive depending on how often they need to be trained. At the same time, a system that gets more information from other sources than simulation might be cheaper to train and use during development.

Functional coverage might provide some of this extra feedback for these new methods. Code coverage is not necessarily the be-all and end-all for verification and would typically be used together with functional coverage. However, introducing functional coverage would again require more human involvement in the verification process since functional coverage needs to be written by someone. Nevertheless, it would be interesting to see if it improved coverage generation as it would provide feedback about tested functionality better.

A potential path forward is to use more complex methods based on neural networks, which can potentially use much more information. It does, however, come at the cost of increased training requirements. One could potentially do transfer learning, where this network could be trained on a broad set of devices before some device-specific training. It would need to be explored if something like this is feasible or if different devices are too dissimilar.

CHAPTER 8

CONCLUSION

Machine learning has seen a massive surge in relevance in the last year and is said to improve both productivity and the quality of work delivered by humans. Proper verification pre-silicon is imperative to have little risk related to tape-out and provide high-quality products. Today verification is both time and labor-intensive and has the potential to see improvements with the use of AI. It could allow for less human intervention, freeing up engineering hours. If AI could deliver the same coverage at a lower computational cost, it would greatly benefit chip design companies and the environment because of the potential reduced data center usage.

This thesis has developed two products: a testbench that a machine learning framework can control and a training and inference loop for the machine learning method based on Bayesian optimization. Training and inference are built around Bayesian optimization, which has previously yielded good results in optimizing expensive to-evaluate functions.

The system has been thoroughly tested on two devices that are a part of the Arm Mali GPU. When testing on these devices, we saw a peek improvement in coverage generated of 5% over random when used with human knowledge, with no additional cost in compute time. It was also a one percentage point improvement over constrained random. Scaling Bayesian optimization to allow to control of extra weights yielded an added cost of a 20% increase in compute time with no improvement in coverage over random. The ML system had a 1% improvement over random constrained testing done by a human. Results for the other device showed no improvements at all and indicate that improvements will depend on the device and can not be assumed to be general without more testing.

The benefits of the wide-scale adoption of machine learning verification frameworks based on Bayesian optimization are yet to be presented. Even though they are the potential to improve coverage, the largest benefits are found when used in conjunction with a human, which is just a slight improvement over human results with random constrained. Currently, there is no way to apply this to larger devices because of the limitations of Bayesian optimization, which also hinder wide adoption. Results are also inconclusive as to whether the system can provide a cost-benefit and would need to be determined on a per-device basis. However, the foundation is there with no extra cost for inference when a projection is not used and, for the suitable devices, a slight increase in coverage over constrained random.

BIBLIOGRAPHY

[1] J. Tørnes, 'Monte carlo tree search and deep neural networks applied to gpu test generation', Project thesis at NTNU as a part of TFE4580.

[2] K. Hu. 'Chatgpt sets record for fastest-growing user base - analyst note'. (), [Online]. Available: https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/. (accessed: 2023.06.07).

[3] Intel. 'Cramming more components onto integrated circuits'. (), [Online]. Available: https://www.intel.com/content/www/us/en/history/virtual-vault/articles/moores-law.html. (accessed: 2023.06.11).

[4] P. I. Frazier, *A tutorial on bayesian optimization*, 2018. DOI: 10.48550/ARXIV.1807.02811. [Online]. Available: https://arxiv.org/abs/1807.02811.

[5] J. Görtler, R. Kehlbeck and O. Deussen, 'A visual exploration of gaussian processes', *Distill*, vol. 4, no. 4, e17, 2019.

[6] K. Weinberger, *Lecture 15: Gaussian processes*, 2018. [Online]. Available: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote15.html.

[7] J. Wu, 'Some properties of the gaussian distribution', *Atlanta, GA: Georgia Institute of Technology*, 2004.

[8] D. Duvenaud, 'The kernel cookbook: Advice on covariance functions', *URL https://www. cs. toronto. edu/duvenaud/cookbook*, 2014.

[9] *Finn.no*. [Online]. Available: https://www.finn.no/.

[10] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams and N. de Freitas, 'Taking the human out of the loop: A review of bayesian optimization', *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016. DOI: 10.1109/JPROC.2015.2494218.

[11] F. Nogueira, *Bayesian Optimization: Open source constrained global optimization tool for Python*, 2014–. [Online]. Available: https://github.com/fmfn/BayesianOptimization.

[12] M. Malu, G. Dasarathy and A. Spanias, 'Bayesian optimization in high-dimensional spaces: A brief survey', in *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, 2021, pp. 1–8. DOI: 10.1109/IISA52424.2021.9555522.

[13] K. Kandasamy, J. Schneider and B. Poczos, *High dimensional bayesian optimisation and bandits via additive models*, 2016. arXiv: 1503.01673 [stat.ML].

[14] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams and A. Aspuru-Guzik, 'Automatic chemical design using a data-driven continuous representation of molecules', *ACS central science*, vol. 4, no. 2, pp. 268–276, 2018.

[15] F. Rosenblatt, 'The perceptron: A probabilistic model for information storage and organization in the brain.', *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[16] K. Weinberger, *Lecture 2: K-nearest neighbors*, 2018. [Online]. Available: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html.

[17] S. Tasiran and K. Keutzer, 'Coverage metrics for functional validation of hardware designs', *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001. DOI: 10.1109/54.936247.

[18] 'Ieee standard for universal verification methodology language reference manual', *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, 2020. DOI: 10.1109/IEEESTD.2020.9195920.

[19] 'Ieee standard for systemverilog–unified hardware design, specification, and verification language', *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[20] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe and I. Shcherbatyi, *Scikit-optimize/scikit-optimize*, version v0.9.0, Oct. 2021. DOI: 10.5281/zenodo.5565057. [Online]. Available: https://doi.org/10.5281/zenodo.5565057.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[22] F. Chollet *et al.*, *Keras*, 2015. [Online]. Available: https://keras.io.

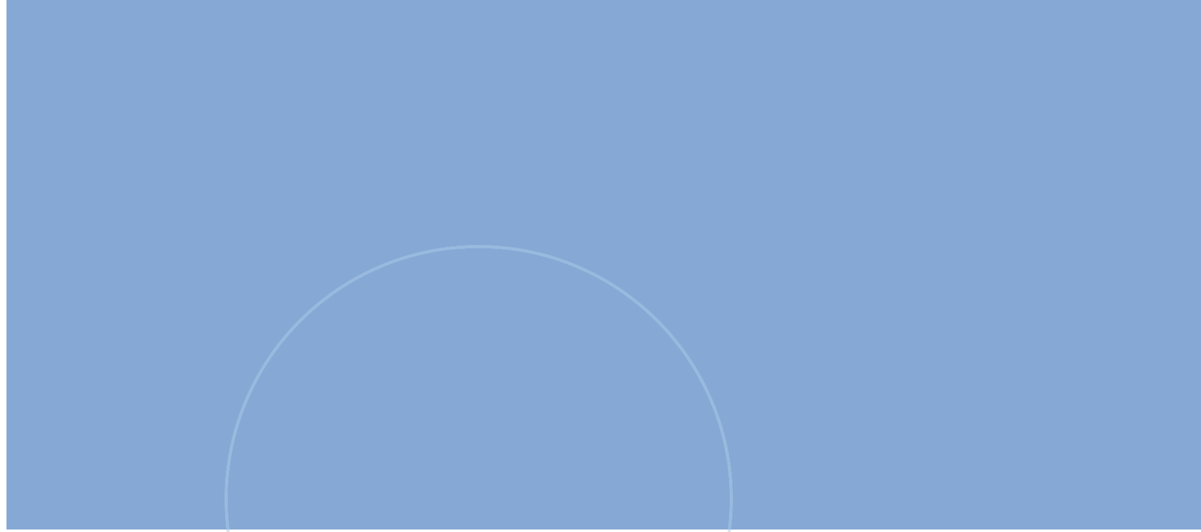
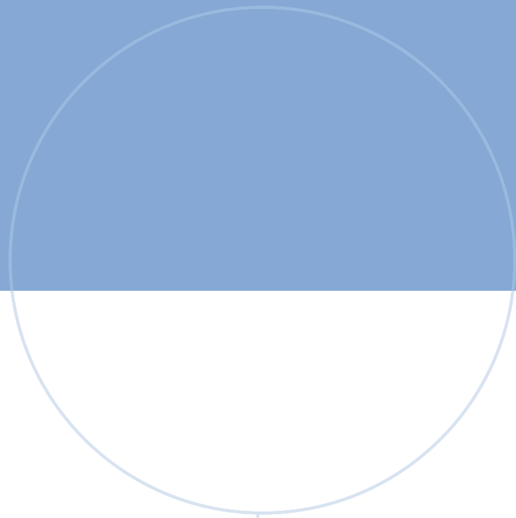
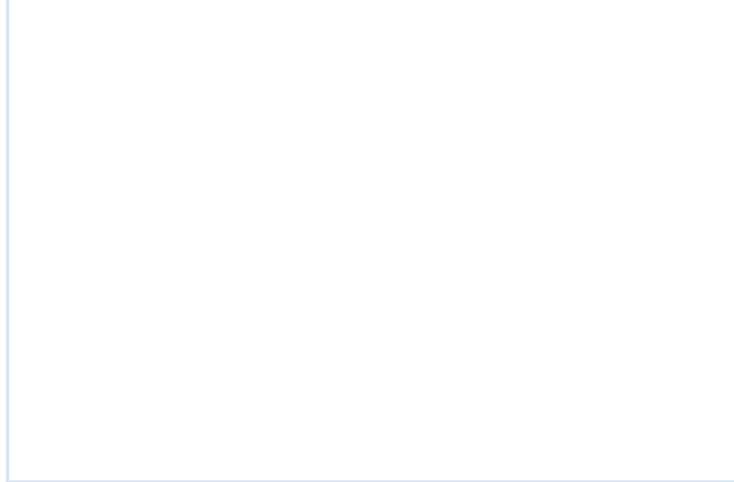
[23] Simens. 'Questa advanced simulator'. (), [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/ (visited on 30th May 2023).

[24] V. Nguyen, S. Schulze and M. A. Osborne, *Bayesian optimization for iterative learning*, 2021. arXiv: 1909.09593 [cs.LG].

[25] N. A. Ahad and S. S. S. Yahaya, 'Sensitivity analysis of welch'st-test', in *AIP Conference proceedings*, American Institute of Physics, vol. 1605, 2014, pp. 888–893.

[26] *Box plot*, Apr. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Box_plot.

[27] *Boolean satisfiability problem*, Jun. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

Norwegian University of
Science and Technology