Investigating Machine Learning for verification of AMBA APB protocol

ABHIRAM SRISAI & MOHAMMED WASIM

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Investigating Machine Learning for verification of AMBA APB protocol

Abhiram Srisai
ab7455ki-s@student.lu.se
Mohammed Wasim
mo7787wa-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors: Erik Larsson and Liang Liu

Examiner: Pietro Andreani

March 11, 2022

© 2022
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

It is a well-known fact that in any Application Specific Integrated Circuit (ASIC) design, verification consumes most time and resources. And when it comes to huge designs, finding bugs can be tedious given the area and the complexity. As per Moore's law, the design complexity is increasing exponentially due to the growing demand for performance. Therefore, On-Chip communication becomes crucial. The interconnects play a vital role in communication between two Intellectual Properties (IP) in a System-on-Chip (SOC), which makes it an utmost priority to verify the protocol. In order to achieve this, many test-scenarios are developed which in turn increases the debug effort and verification space. As Advanced Microcontroller Bus Architecture (AMBA) protocol is most commonly used as a communication protocol, the Design Under Test (DUT) for this thesis is Advanced Peripheral Bus (APB), a member of the AMBA family.

This thesis aims to investigate the applications of Machine Learning (ML) to reduce the overall verification time and effort. Basic classifiers such as K-Nearest Neighbors (KNN), Decision Trees (DT) are explored and studied, along with two types of Neural Networks, such as the FeedForward Neural Network (FFNN) and Recurrent Neural Network (RNN). These algorithms were trained overtime with various datasets along with fine-tuning their respective parameters. The Long Short Term Memory (LSTM) model, a variant of the RNN is the preferred among other models as it is capable of learning the complete behavior of the APB. From the results obtained, the LSTM was able to classify the write, read and the failed transactions with an accuracy of 90%. The results also discusses the accuracy obtained by other models and compares the time and effort taken to implement all of them. The study is concluded with a belief that ML can be a method in verification with suggested improvements. The ideas for future studies have been briefly presented as well.

Keywords : Machine learning, SOC Verification, AMBA, Neural Networks, Deep Learning, Assertions.

Acknowledgements

We would like to take this opportunity to thank Stefan Nilsson BA (Ericsson AB) for giving us an opportunity and resources to pursue our thesis at Ericsson, Lund. We would like to thank our supervisors Shkelqim Lahi and Magnus Österholm from Ericsson AB for their constant support and motivation. We want to thank our supervisors Erik Larsson and Liang Liu from LTH for their guidance. Furthermore, we are grateful to Christel Bergh for giving us an opportunity to shape our careers and ensuring our future growth at Ericsson.

Lastly, we would like to thank our parents, friends, and all the persons whom we forgot to mention here.

Popular Science Summary

Having an implemented design and its specification on the same page is the crux of any given ASIC. This desired overlapping is achieved by scrutinizing the design at every stage of its design cycle. Verification of an ASIC has grown extensively over the years from basic functional stimulus verification to formal verification standards and etc., and this alone has resulted in various verification methodologies, one such being *Assertion Based Verification (ABV)*. Unlike earlier days, verification engineers now have a huge demand in the industry. With growing design complexities, it is important to verify the hardware with various tests before tape-outs in order to avoid any possibility of a bug post tape-out.

Verifying large circuits comes with many challenges. Huge number of signals spanning over millions of clock cycles and monitoring their behaviour is laborious and time consuming. Although, assertions come in handy in highlighting the bugs, writing assertions are not as simple as it looks. The complexity of the design is directly proportional to the complexity of writing assertions.

On the other hand, with the advent of Machine Learning (ML), its extensive usage in the development of newer technologies and simplifying existing practices over the years has proven itself to be flexible and reliable. The ability to learn features and either classify the nature or perform mathematical calculations and predict the behavior in quantitative terms has paved a path for many industries, engineers and students. Few of the examples where ML is dominating are Image Processing, Robotics, Medical Engineering and Statistics. With various ML algorithms ranging in their complexities and the nature of predictions, understanding their behavior with regards to hardware design and verification can be helpful in determining their competence and need for it. Presently, very few articles and journals have been published as many leading industries such as Ericsson, ARM, Cadence, Accellera, Xilinx are currently researching in this field.

Taking note of these challenges and approaches, the aim of this thesis is to study and apply the concepts of machine learning in the field of hardware verification to understand how it improvises the current state-of-the art methodologies. A comparative study has been performed listing out its advantages and disadvantages along with the results of various algorithms explored.

Glossary

1. **Checker**

   A checker is a block in the simulation based verification environment used against a DUT to compare and verify the obtained data with the expected data..

2. **Code Coverage**

   Code coverage is a measure which describes the degree of which the source code of the program has been tested.

3. **Coverage**

   Coverage is a metric to assess the progress of any verification activity. Few types of coverage are code coverage and functional coverage.

4. **Driver**

   A driver is a block used in a simulation based verification environment that collects and sends data to the design.

5. **Functional Coverage**

   Functional Coverage is the determination of how much functionality of a design has been exercised by a verification environment.

6. **Machine Learning**

   Machine learning is the study of computer algorithms that can improve automatically through experience and by the use of data[20].

7. **Mass Erase**

   A flash memory consists of several blocks which store data. A mass erase operation will erase the data stored in all blocks hence clearing out the entire flash memory..

8. **Monitor**

   A monitor is a block used in a simulation based verification environment that monitors the data from the design to the scoreboard..

9. Page Erase

A page is the smallest programmable memory unit in a flash memory. A number of pages clubbed form a block. Once each page unit is written with data, performing a page erase operation then will clear out all the pages in a block..

10. Scoreboard

A scoreboard usually consists of a reference block that calculates the expected output from the design. It also consists of a comparison logic compares the obtained output from the design with the expected output..

11. Stimulus

Stimulus are packets of data usually in the form of transactions that are generated to be fed as either input data sequence and/or control sequence to a register component or control unit from the testbench.

Acronyms

12.	**ABV**	Assertion Based Verification.
13.	**AHB**	Advanced High Performance Bus.
14.	**AI**	Artificial Intelligence.
15.	**AMBA**	Advanced Microcontroller Bus Architecture.
16.	**APB**	Advanced Peripheral Bus.
17.	**ASIC**	Application Specific Integrated Circuit.
18.	**AXI**	Advanced Extensible Interface.
19.	**CNN**	Convolutional Neural Network.
20.	**DL**	Deep Learning.
21.	**DT**	Decision Trees.
22.	**DUT**	Design Under Test.
23.	**EL**	Ensemble Learning.
24.	**FFNN**	FeedForward Neural Network.
25.	**IP**	Intellectual Property.
26.	**KNN**	K-Nearest Neighbors.
27.	**LSTM**	Long Short Term Memory.
28.	**MDP**	Markov Decision Process.
29.	**ML**	Machine Learning.
30.	**MSE**	Mean-Squared Error.
31.	**NB**	Naive-Bayes.
32.	**NLP**	Natural Language Processing.
33.	**NN**	Neural Network.
34.	**RNN**	Recurrent Neural Network.
35.	**RTL**	Register-Transfer Level.

36. **SAPD**	Supervised Anomalous Path Detection.
37. **SOC**	System-on-Chip.
38. **STA**	Static Timing Analysis.
39. **SVA**	SystemVerilog Assertions.
40. **SVM**	Support Vector Machine.
41. **VIP**	Verification Intellectual Property.
42. **VLSI**	Very Large Scale Integration.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Verification plays a critical role in the design flow, where the design is verified against its specification. With the increasing demand to innovate and the technology node shrinking, verification takes about 70% of the overall time [12].

As the modern designs are more generalized and are no longer used for just one product but will be used across multiple generations of the same product. The main focus is to achieve functional correctness of the design before the tape out because any undetected bugs can lead to additional costs and also affect the time to market. To achieve this, industries have a dedicated team of verification engineers who spend their time planning, developing, and debugging System-On-Chips(SOC). Figure 1.1 shows a pie chart of the estimated time spent by verification engineers [10]. 22% of the total time is spent on developing test benches and creating test simulations, while 14% is dedicated to planning the test process. Whereas, the highest percentage, 39% is allotted to debugging the errors or bugs in the design.

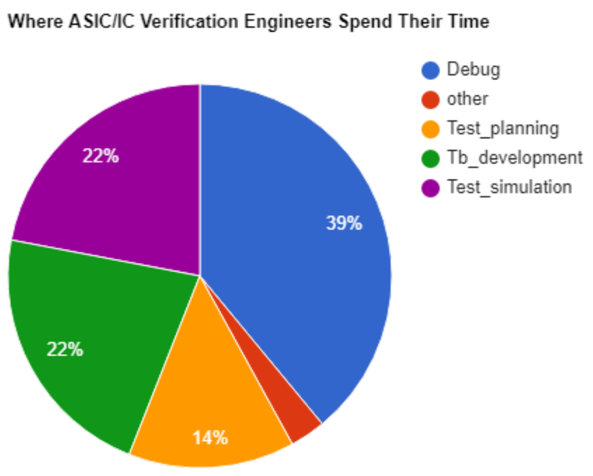


Figure 1.1: Pie Chart [10]

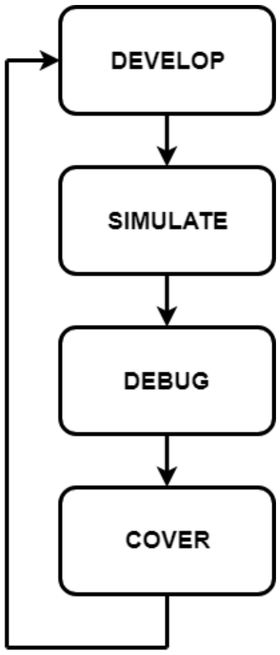


Figure 1.2: Verification Cycle [19]

Figure 1.2 illustrates the basic verification cycle. The cycle includes four phases:

- **Develop**: This phase includes a verification plan which lists procedures and methods that are used for verification, and a verification environment that can be formal or simulation based. The formal based environment consists of assertions that are statements used to validate the behavior of the system and simulation based environment consists of testbench and test cases created using UVM, C or systemverilog[5].

- **Simulate**: In this phase, the test cases developed are simulated using the EDA tools such as Incisive from Cadence, Questasim from Mentor Graphics, and emulators such as Palladium from Cadence.

- **Debug**: It is a crucial step in the cycle that includes debugging failures of the test cases or assertions on a transaction level, signal level, and etc.

- **Cover**: This step is to check the obtained coverages such as Functional Coverage[5] and Code Coverage[2] as defined in the verification plan , if not then it is fed back to the development phase to re-iterate the test cases with a different stimuli to achieve 100% coverage.

As the complexity of the design increases each of the above steps becomes challenging, along with higher possibility of bugs in the design.

1.1 Communication Protocol

In current era, modern System-on-Chip (SOC) designs consist of several Intellectual Properties (IP) [26] and one of the challenges is the on-chip communication between different IPs. To facilitate this communication, IP cores are designed with different interfaces and internal protocols, which can be problematic in integrating them into a SOC. To avoid this problem, standard on-chip bus protocols were developed. Some of the publicly available bus architectures from leading manufacturers are CoreConnect from IBM, AMBA from ARM, and SiliconBackplane from Sonics. The designs typically have one or more microcontrollers or microprocessors along with several other components such as internal memory or external memory bridge, DSP, and other peripherals like USB, UART, PCIE, I2C, etc, all of which are integrated on a single chip.

For the desired functionality of the SOC, the communication between the IPs plays a vital role[3]. Any failure of the protocols will lead to the failure of the complete functionality of a SOC. A few key points to consider when verifying a protocol are stated below [19] :

- **Functional Correctness and Verification Completeness**: The verification environment should generate the correct Stimulus[11], Checkers[1], and Coverage[3] of systematic transactions between the multi-layered interconnect. As the design complexity increases, it becomes very crucial to have a mechanism that can check every transaction from one point to the other, with different protocols, and also parallel executions.

  Functional correctness and verification completeness go hand in hand. To make sure that interconnect is functionally correct, all the possible corner cases, error scenarios should be covered. To achieve this goal, a robust stimulus generator, a response checker, and a coverage model to counter the challenges of functional correctness of the interconnect should be created.

- **Stress Verification**: To obtain the highest verification degree, randomizing and firing all the triggering points at the same time is important as this stresses the interconnects for functional verification and performance verification. Once the functionality of the interconnect has been verified with subsystem, it's time to fire all the subsystems at the same time, to mimic the real-life scenario. This will not only stress the interconnect for functional verification but also for its latencies and performance verification. This type of verification can be performed using the following approaches :

  - **Write and Read Tests**: This is a register access test, where a value is written into the register and read from the same. Any discrepancy is either a bug in read/write logic or the interconnect.
  - **Protocol Checkers**: This scenario is where all the checkers for a protocol are applied using assertions, these assertions will catch the protocol bugs for example control bit handling and delay logic.

- **Functional Errors due to Latencies and Security Management**: In this process, the unmatched bandwidth of read-write cycles and also the latencies from request to request, response to response should be checked. All the latest interconnects include security management features, the main functionality of this is to forbid any unsecured transactions targeted to memory space. This security feature is to prevent software attacks causing illegal instruction execution. The verification environment should check each transaction requested by initiators and abort with an error response if that transaction is targeting a protected area.

The Design Under Test (DUT) considered for thesis work is Advanced Peripheral Bus (APB) which is part of the Advanced Microcontroller Bus Architecture (AMBA) family. AMBA bus protocol is a set of interconnect specifications from ARM that standardizes on-chip communication between various IP blocks for designing a high-performance SOC. The primary goal of AMBA protocol is to have a standardized and efficient way to interconnect these IPs[1]. APB is a simple non-pipelined protocol which is used for connecting low bandwidth peripherals [1]. A detailed functionality of APB is explained in chapter 4.

1.2 Machine Learning

Machine Learning[6] is the study of computer algorithms that improve automatically through experience and by the use of data[20]. It aims to find natural patterns in the data and learn them in order to produce outputs. Some of its applications are in image processing, finance and stock market trading, medicinal engineering, text prediction and etc. With the constant support and improvements in ML over the years, it has proven itself to be flexible and reliable. Briefly, implementation of ML can be broken down into 3 parts. The first part being the type of prediction such as Classification or Regression. The second part is the type of algorithm to be employed under each prediction type. The algorithms studied and implemented in this thesis are Decision Trees (DT), K-Nearest Neighbors (KNN), FeedForward Neural Network (FFNN) and Long Short Term Memory (LSTM) to predict the behavior of APB. The third part is the selection of features. Features generally are data points required to feed the algorithm. These data points are mainly of two types, input data points and output data points. This data put together defines the behavior of an application. The data to any system can take various forms mainly such as numerical data, categorical data, text or time series data. The general idea is to teach the algorithm with some percentage of data(training data) and then test it against unseen data(test data) to determine how accurate the system is, this is usually done in iterations to strengthen the learning. Chapter 3 discusses the ML and its concepts in depth.

1.3 Motivation

With the advancements in the field of Very Large Scale Integration (VLSI), verification becomes bottleneck as millions of transistors are integrated on a single chip. One of the main challenges is to speed up the verification process while ensuring the verification quality. As the complex hardware involves more than a million lines of code. Test cases created to verify the Register-Transfer Level (RTL) generate a large amount of data after each simulation which can be a tedious job for an engineer to analyze [27]. Machine learning, a powerful technique has the potential to analyze enormous data and provide the expected results/predictions. The type of algorithm to be used can be determined by looking closely at the problem.

The common methods used to verify a design are by using formal verification[24] or by developing a systemverilog verification environment[23]. An assertion is a formal property check which is defined with the expected behavior of a design and is tested against the same design[4]. When an assertion fails, it could indicate an error in the design. A verification engineer writes the assertions based on the design specification. Likewise, a simulation environment is a type of test environment which is instantiated with the top module of a DUT. This environment usually consists of various components such as a Stimulus[11] , Driver[4], Monitor[8], Scoreboard[10] among many others[5].

Few of challenges faced by verification engineer during the development of either of the environments are :

- **Formal Verification Environment** : Formal Verification is a computationally expensive step in the verification of today's complex hardware designs[8]. In such an environment, a lot of time is spent on developing or writing assertions itself and this is due to three contributing factors[4].
    1. Verifying a larger design could require multiple assertions to cover all probable functional scenarios and corner cases of interest.
    2. Verifying a complex functionality of a design could require writing complex assertions as well.
    3. Lastly, it is important that all the assumptions or design specifications are considered carefully as missing out on these might lead to an incomplete assertion. This will result in assertion failures.
- **Simulation Verification Environment** : The simulation based verification has existed for many years and is considered as the backbone for verifying DUTs, but it has its challenges. For complex SOC designs, creating and integrating a testbench environment is a lengthy process in order to support all the types of testcases that cover multiple scenarios of a DUT [20], where the quality of the test is measured by coverage metrics [6].

With an intent to address the aforementioned challenges with a feasible solution, the application of ML to verify a APB protocol has been investigated in this thesis work. The investigation proposes four types of classification based ML algorithms which could suit to learn the functionality of the APB and verify the same efficiently. The four algorithms studied and implemented are K-Nearest Neighbors (KNN), Decision Trees (DT), FeedForward Neural Network (FFNN) and Long Short Term Memory (LSTM). A reference model using SystemVerilog Assertions (SVA) against random constrained stimuli of the APB control signals was created to train the models. The proposed study mainly investigates the trade-off between various algorithms such as the time taken to implement along with the complexity of the model, the nature of the datasets selected as features to train the model and most importantly, the accuracy of the models.

1.4 Thesis Structure

The following chapters in the report are organized as follows. Chapter 2 briefly explains work done in the related field. Chapter 3 provides the theoretical knowledge of ML, the APB protocol, the concepts of SVA and constrained random stimuli generation. Chapter 4 presents the implementation of the various ML algorithms and chapter 5 discusses the overall results obtained. Lastly, chapter 6 summarizes the thesis work along with future scope.

Chapter 2

Background Study

This chapter briefly talks about earlier work done in verifying a bus protocol. Studies done on how ML algorithms have been implemented to verify various aspects in a SOC have been discussed as well.

The work done by Gurha and Khandelwal in *"SystemVerilog Assertion Based Verification of AMBA-AHB"*[24] provided a basic introduction to Advanced High Performance Bus (AHB) protocol and how they are verified using SVA. The authors define a few key concepts such as *Properties* and *Sequences* which constitute an *Assertion*. A sequence can be defined as a set of Boolean expressions which are assessed on the same clock cycle or over a period of clock cycles. Multiple sequences can be merged/combined together to form a property which are then verified or *asserted* during a simulation. Chapter 5 provides a detailed explanation about SVA. Gurha and Khandelwal's implemented design consisted of the 3 masters and 4 slaves and was verified against various properties. These properties were described with the behavior of the handshaking of signals between various masters and slaves. The assertions are declared in a separate file and are then bound to respective instances using the 'Bind' function. This has two advantages, first advantage is to ensure both the processes of design and verification progress in parallel and second advantage is that they can be verified without having a control over the RTL files. For this work, various write and read transactions were initiated, the simulations were performed using ModelSim. Gurha and Khandelwal conclude the work by tabulating their observations for each assertion detecting the number of failed and passed transactions. Their results also claim that the overall coverage increased with the increase in the number assertions. For example, with 4 assertions they obtained a coverage of 28% and as they increased gradually to 20 assertions they were able to improve the coverage to 80%.

Similarly using the AHB protocol as the DUT, in another academic study [23], Perumalla and Choudhary verify the write and read behavior of the protocol by developing a verification environment in systemverilog. The DUT consisting of 1 master and 4 slaves is interfaced with the verification environment. This environment includes a Generator which generates randomized values for the data, address and control signals. These values are then driven to the DUT and the Scoreboard by a Driver simultaneously. The scoreboard obtains two sets of data, one from the DUT through the monitor and the other set of data from the driver which is used to calculate the expected output. A Scoreboard is used to compare the data obtained from the DUT with the expected data and decide if the transaction has passed or failed. Perumalla and Choudhary used QuestaSim to simulate and verify the protocol and obtain both functional and code coverages. From their coverage reports, they claim that they obtained 100% functional coverage but, they believe by improvising the testplans and the constraints for generation of random constraints, a 100% code coverage could be obtained.

With the same concepts of the systemverilog verification environment described in the above work[23], in another study, Han ke et al. design an AMBA Verification Intellectual Property (VIP) which includes an AHB master and an AHB monitor [12]. Likewise, they aim to reduce the time spent on visual inspection of waveforms by using a reference model which is used to compare the behavior of DUT with the expected behavior. Han ke et al. believe that their VIP can be reused to verify any AMBA protocol. The DUT in this study consists of two masters and four slaves. One master is the DMA and the other master is the VIP. The slaves are Flash memory, an SRAM, a ROM and a bridge that is connected to the APB protocol. In this paper [12], han ke et al. intend to verify the transactions between the VIP master and two of the four slaves which are the SRAM, and the Flash memory respectively. To verify the flash memory, initial write and read transactions were performed where the data values were stored in respective addresses. Then, the memories were cleared out by performing the Page Erase[9] and Mass Erase[7] erase operations which write 16'hFFFF to all the memory locations. Likewise, the write and read transactions in sequence were initiated to the SRAM. The reference model designed by the authors mimic the behavior of the SRAM and Flash memory to compare the responses obtained from the slaves to determine if the transaction has passed or failed. Han ke et al. conclude their study with their results and a drawback. From their results, a report was generated which highlighted the failed testcases. For example, in flash memory, three out of nine mass erase testcases had failed. Authors in Han ke et al. believe that their VIP can be used to verify the AMBA protocol and can reduce the time spent on waveform inspection but the main drawback being the design of reference model which is modelled specifically to mimic certain components. Improving the flexibility of the reference model was considered for future work.

Lida Bai and Lan Chen propose the idea of employing ML models for timing prediction in SOC physical design process [2]. A physical design flow mainly begins with importing netlists, creating floor-plan, placement and optimization of standard cells, clock tree synthesis, routing and design signoff. As the design advances in each stage, the static timing analysis is important to fix any timing violations. STA is used to validate the timing performance of the design by checking all the possible timing violations in the path however no functionality check is performed. The process of physical design in general is tool intensive, complex and time consuming. Their work involves the study and implementation of the ML learning models such as the Backpropogation neural network, Support Vector Machine, Ensemble Learning and a hybrid model which is a combination of the aforementioned models. The authors use these learning models to predict the slack right after the floor-plan has been established. For each of the learning models, various parameters such as the clock period, transitive fanout, standard cell utilization and many more have been extracted as features from the netlist, constraint and the floor-plan files. These features are the data sets used to train and test against the above-mentioned models. The authors Bai and Chen claim to have generated around 500 samples of data sets where 60% of the data was used for training while the rest was used to testing. They conclude their work with the result showing that the hybrid model has better prediction accuracy even with a few mispredictions in comparison with the rest of the learning models. With this result, the authors believe that their model can provide better guidance for timing closure effectively by reducing the time consumed in the process of physical design.

To ensure the ASIC design integrity, the authors in [21] have formulated an ML model to quantify variations in the RTL or the Graphic Design System II formats. In order to measure these variations, the timing delays are studied. The issue the authors address in their work is Static Timing Analysis (STA), by itself does not have the capability to detect design path variations among STA instances within the same SOC. Static timing analysis is done to detect timing violations for all possible paths in a design. STA is a crucial step in the process of physical design as it validates timing performance. Their work proposes a semi-supervised learning approach by employing Supervised Anomalous Path Detection (SAPD) to identify these variations by comparing the paths with various STA instances. Two sets of databases were created by considering various timing parameters and capacitances as features. The main difference between the databases is the random path variations in them in order to simulate unintended design modifications. From the experimental analysis, the authors claim that the algorithm was able to effectively detect path variations between STA instances and they believe that for future research, their algorithm can be utilized to develop self-correcting anomalous paths.

According to the work done in “Optimizing Random Test Constraints Using Machine Learning Algorithms” by Stan Sokorac[27] modern designs are extremely complex. It is impossible to manually come up with all the stimuli necessary to completely validate the design. To solve this problem, verification engineers rely heavily on constrained random simulations. By writing a set of constraints and generating random streams of transactions, one could hit both common and uncommon design corners. Stan states that random testing is also very inefficient and expensive, as it involves running millions of random simulations to find the corner case bugs. In this paper, he presents two ideas to make the random simulation more effective at finding these bugs. The first idea is a new type of coverage which is toggle-pair coverage is designed to provide feedback that is likely to expose bugs. The toggle-pair coverages is a metric that is created by monitoring all the flop toggles and extracting all toggle pairs with the total number of bins is n2, where n is the number of flops in the design. The second idea is to select and tune random tests by using the coverage feedback with the help of machine learning algorithms. In this study, two machine learning algorithms that are used are genetic algorithm and clustering, and the comparison of these two is done in the end. The author claims that the results are promising with optimized regressions that have failed consistently more than the non-optimized ones, and have uncovered a larger number of unique failures. To conclude the author states that this methodology can be effective at producing higher rates of failures in a smaller number of simulation cycles, while simultaneously exposing new bugs, through new unique fails.

Chapter 3

Theory

3.1 Machine & Neural Networks

An application of Artificial Intelligence (AI) where a system(often seen as a black-box) is trained to understand the characteristics of a given object in the form of data can be briefly defined as *Machine Learning.* It aims to find natural patterns in the data and learn them in order to produce outputs. The data to any system can take various forms mainly such as numerical data, categorical data, text or time series data. Some of its applications are in image processing, finance and stock market trading, medicinal engineering, text prediction and etc.

Machine learning branches out in three ways of learning, *Supervised Learning*, *Unsupervised Learning* and *Reinforced Learning.* In supervised learning, for a given set of input parameters also called as *Predictors* there is an output assigned to it, *Response.* Taking these sets of data, a model is built by first training it with certain percentage of data and then the new unseen data is introduced and tested against this model to check its predictive accuracy. In short, supervised learning is used when the outputs to the inputs to be trained are known or given.

Supervised learning branches out in two techniques, *Classification* and *Regression.* The method of arranging a group of objects into its respective categories based on similar characteristics can be defined as Classification. Similarly, data can be organized into categories or *classes* based on a similar pattern. Few of the generic examples of classification problem includes classifying whether a person has a heart disease or not, or a given image is a cat or a dog and so on. Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Naive-Bayes (NB), Decision Trees (DT) are some of the basic and commonly used classification algorithms. Regression on the other hand is employed when your output is a continuous numeric data, as in outputs that cannot be classified, but instead need to be calculated. Example, calculating the temperature, or predicting the runs per game in a cricket match or even the stock market values for a company. Some of the regression algorithms are Linear model, Regularization, Step-Wise Regression and etc.

Unsupervised Learning basically works on the principle of self learning by finding patterns and grouping them without pre-existing labels. *Clustering* is one of the most common technique. K-Means, Self Organizing Maps, hidden Markov models and etc are some of the clustering algorithms. Reinforced Learning is when a model trains itself and makes a decision on the correctness of output and based on that it will reward itself. Markov Decision Process (MDP) is one of the most common algorithm used.

Apart from these, Deep Learning (DL) is a special type of machine learning which are based on Neural Networks. They are complex in nature consisting of various layers that contain information. The advantage of DL is that they can achieve higher accuracy and also can handle large data-sets.

As our study is based on classification approaches, this chapter will concentrate on various classification models employed.

3.1.1 Classification

Classification is a process where for a given set of features (inputs or data points), a class (output) is predicted. In other words, predicting a label for a set of data can be termed as classification. Classification of various species of a *Iris* flowers [9] is a classic example. In this example, there are four features namely *sepal length*,*sepal width*, *petal length*, *petal width*. There are three classes or labels, *Setosa*, *Verisicolor* and *Virginica* and have more than 100 observations recorded for these features. With the combinations incorporated with various algorithms, a classification was made.

Similarly, these concepts can be established in the field of hardware design and verification. As the aim is to classify the behavior of the DUT, several classification based algorithms that are implemented in this thesis study have been explained in this chapter, the implementations and the results are discussed in 4 & 5.

3.1.1.1 K-Nearest Neighbors

K-Nearest Neighbors is a type of classification algorithm widely used in pattern recognition. It works on a simple assumption of similarity. When a new observation to be classified is introduced, the classifier first searches for the nearest known observation(s) and based on that, categorize the new observation with that of the known one. The known observations within its vicinity are termed as *Neighbors* and the term 'vicinity' is what defines K. The minimum value of K is 1, which means that for a new data point, it'll only search for 1 nearest neighbor as seen in figure 3.1. The disadvantage of having K as 1 is the probability of the new observation being misclassified is high. To overcome this, K needs to be assigned with a value greater than 2. Now, for example, when K is defined as 3, the new observation now looks for 3 known observations or neighbors and then the majority class among these neighbors will be assigned as the class of the new observation. But when the algorithm searches for the neighbors, the distance between the test vector and a neighbor is measured. This distance is another deciding factor when assigning a class to a test observation. There are many techniques to measure the distance which are *Euclidean Distance*, *Manhattan Distance*, *Hamming Distance*, *Minkowski Distance*, *Chebychev* and etc [16]. Euclidean Distance is the most commonly used distance measure. It is the square root of the sum of the squared differences between the known data point and the unknown data point in n dimensions starting from $i = 1$. It is given by the formula,

$$distance(x_k, x_{uk}) = \sqrt{\sum_{i=1}^{n} (x_{unknown} - x_{known})^2} \tag{3.1}$$

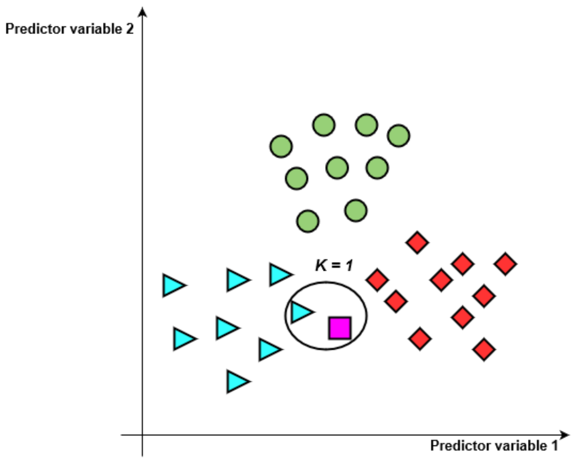


Figure 3.1: KNN algorithm with K = 1
[15]

Figure 3.1 depicts how the KNN algorithm works when it the number of neighbors are defined as 1. The blue, red and green points on the graph represent 3 different classes while the pink point is a test vector which is yet to be assigned a class.

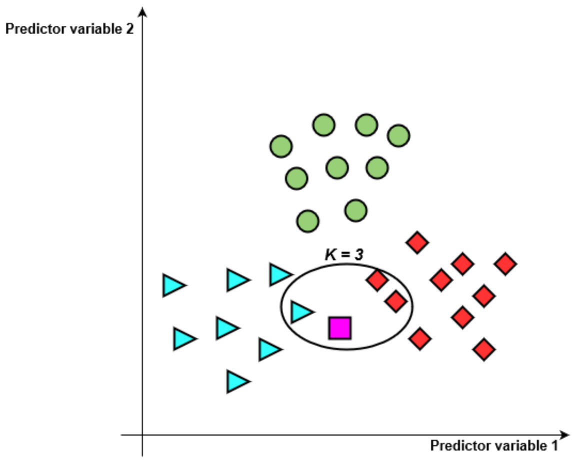


Figure 3.2: KNN algorithm with K = 3
[15]

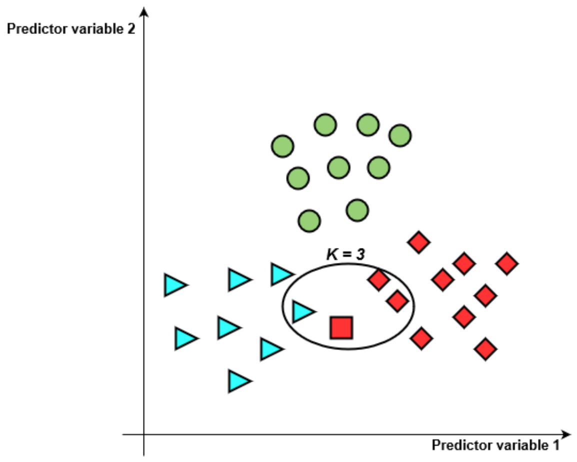


Figure 3.3: New observation assuming the class of majority. [15]

As explained above, figures 3.2 & 3.3 depict when the neighbors are set to 3. The latter shows the previously pink colored test vector assumes the class of the red ones due to majority presence of the class.

3.1.1.2 Decision Trees

Decision Trees (DT) are a type of supervised learning based classification algorithm. As the name suggests, they're trees in hierarchical fashion but instead of branching out upwards they branch out in downward direction based on every decision they make. The principle operation behind a DT is that it splits the given working set into subsets repetitively till it reaches a point where the working set can no longer be split. It creates a top down structure with conditions answered with a 'yes' or a 'no'. Functionally, they are quite similar to flowcharts and are simple yet a powerful algorithm. Decision trees are known for their high accuracy as they have the ability grow deeper in order to make better decisions.

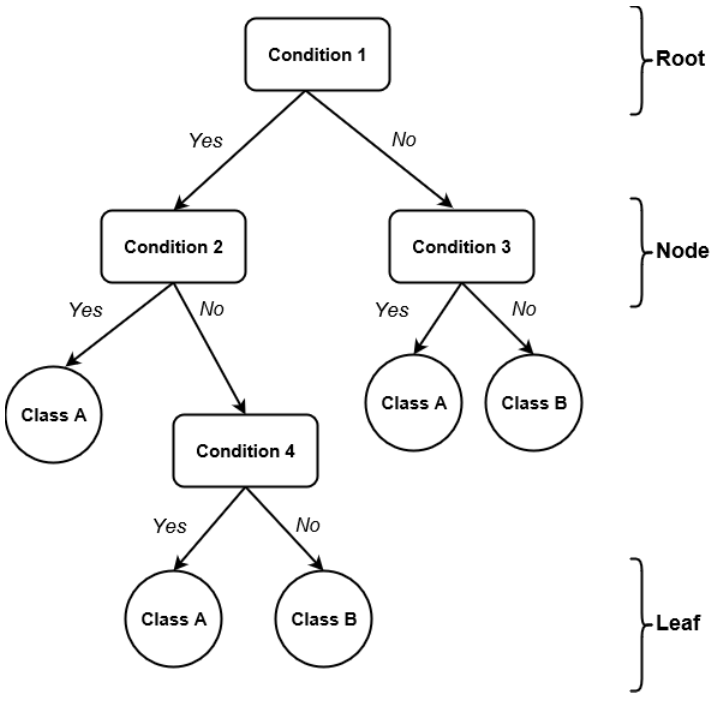


Figure 3.4: A simple binary decision tree.

Figure 3.4 gives us a basic representation of a two-class decision tree. A decision tree has roughly three main components.

- **Root** : This is the origin of the tree. As mentioned earlier, from this point on wards, the tree begins to split grows downward.
- **Node** : These are intermediate conditional points in the tree where certain decisions are made which contribute to the depth of the tree and as well as the outcome.
- **Leaf** : A leaf is basically considered as the last stage of the branch which basically holds the outcome for that respective branch.

3.1.2 Neural Networks

A Neural Network is a chain of neurons connected to each other in the form of layers which perform certain mathematical operations on the inputs provided and produce an output. It is modelled to operate in a way the human brain performs. They are the building blocks of any given neural network. There are various types of NN as of today. The most common NN is Convolutional Neural Network (CNN) due to its ability to solve complex problems. Its mainly utilized and favored in the field of Image processing. The other NN that has been gaining recognition is the Recurrent Neural Network (RNN), often used for Natural Language Processing (NLP) such as text and speech recognition.

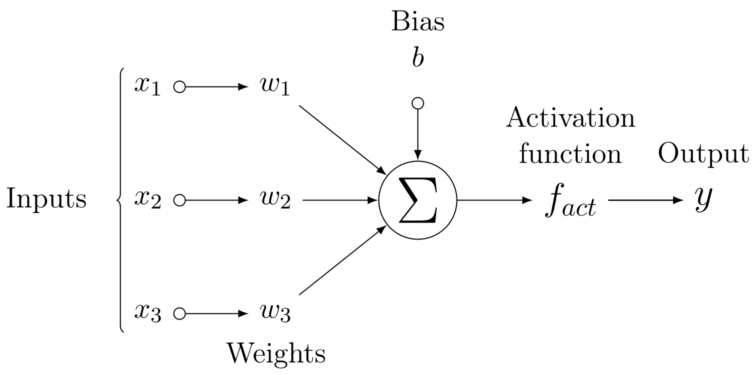


Figure 3.5: The inside of a simple neuron.

Figure 3.5 shows the basic architecture of a single neuron. Each input to the neuron is first multiplied by a parameter called *weight*. These two products are later added with a bias value.

$$p = (x_3 \times w_3) + (x_2 \times w_2) + (x_1 \times w_1) + b \tag{3.2}$$

For a given set of predictors, it's respective numerical quantity/value can possibly vary in ranges which can be of a major challenge for the model to learn the behavior. To overcome this, the sum, p is passed through an activation function such that all the numerical values are now translated with in a fixed range, i.e., between -1 to 1. Sigmoid functions are the most common functions used.

$$y = f(p) \tag{3.3}$$

The sigmoid function of this neuron is,

$$y = \frac{1}{1+e^{-(p)}} \tag{3.4}$$

Once the transfer function is calculated, the networks are trained and tested to measure the correctness of the model. The incorrectness or the error in prediction can be called as *Loss*. For any given network, it is very crucial to know how well the network has learned. Cross-Entropy and Mean-Squared Error (MSE) are some of the loss functions used.

In order to calculate the error for a classification problem, a square of the comparison between true output and predicted output which is called the *squared error* is performed. The Mean-Squared Error, an average of all the squared errors is computed which is given by,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{True} - y_{Pred})^2 \tag{3.5}$$

where, 'n' stands for the samples or observations, while y_{True} and y_{Pred} indicate the true output and the predicted output.

In the next section, FeedForward Neural Network have been discussed briefly.

3.1.2.1 FeedForward Neural Network

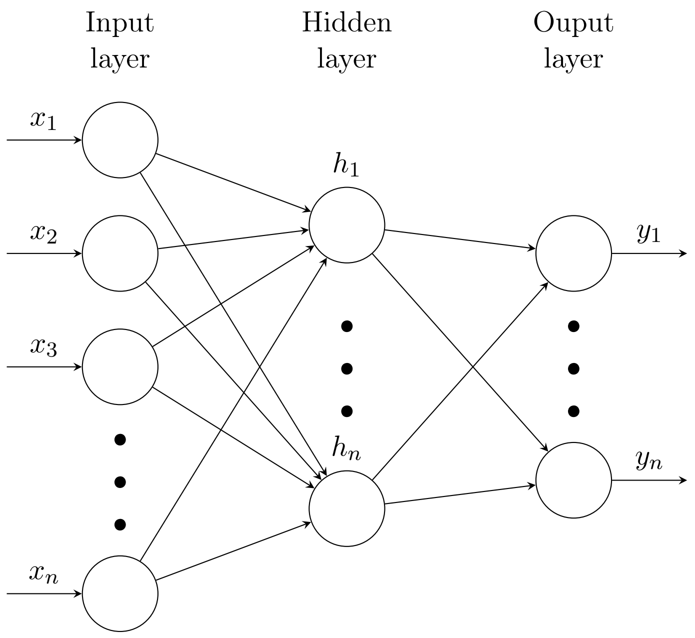


Figure 3.6: A simple feedforward network

They are a basic type of a neural network. It is alternatively termed as *Forward-Pass Computation* or a *Multilayer Perceptrons.* The basic structure consists of input neurons which are basically dependent on the features. There is a minimum of 1 hidden layer consisting of several hidden neurons. As the name states, this layer is invisible. The structure ends with an output layer consisting of output neuron(s). Both the input and the output layers are visible. FeedForward Neural Network (FFNN) are usually applied to text data, image data and most importantly for us, the time series data.

FFNN is a two step process. From figure 3.6, the first step is to calculate the values of the hidden layers with the help of the weights and the inputs. The second step is to calculate the values of the output layer with the help of the freshly calculated weights and the synapses form the hidden layers.

With reference to the formula 3.2,

$$p_n = x_{i,j} \times w_{i,j} \tag{3.6}$$

And lastly, for the output layer,

$$y_n = f(h_i, h_j) \tag{3.7}$$

The output activation function can be either a *Softmax* function or a *Linear* function. Usually, for a classification problem, a softmax function is preferred.

3.1.2.2 Recurrent Neural Network

As machine learning and deep learning are about performing certain computations on matrices, the basic and yet complex NNs such as the FeedForward Neural Network and Convolutional Neural Network (CNN) treat every row in a given matrix as an observation along with its respective output.

Unlike the traditional method, there are few use cases where a class is to be assigned to a group of observations. That is, when the observations on row n are dependent on the (n-1) row, this can form a group of rows or can be seen as a batch of data which describe a certain behaviour. It is safe to say that the aforementioned networks are not quite capable of remembering or retaining the previous information which brings us to Recurrent Neural Network (RNN).Recurrent Neural Network (RNN) have addressed this issue of long-term dependencies, particularly the Long Short Term Memory (LSTM) networks. First introduced by Sepp Hochreiter & Jürgen Schmidhuber [11], they're designed to remember the previous information. The basic working principle behind this network is to produce an output for a group of interdependent data.

3.1.2.3 Long Short Term Memory

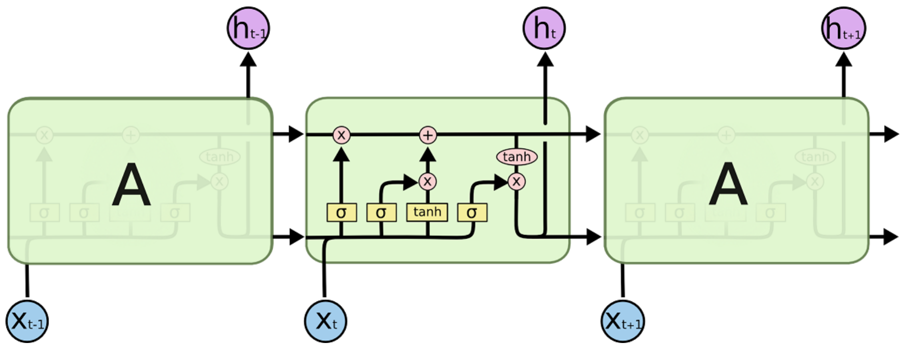
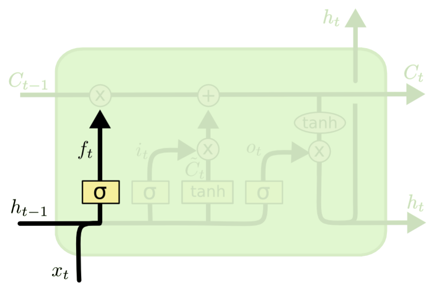


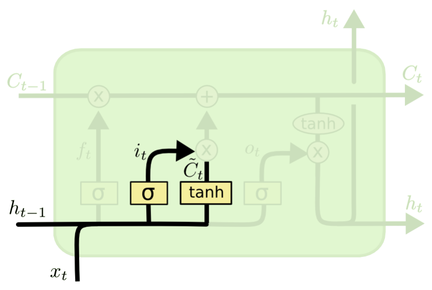
Figure 3.7: A basic architecture of LSTM.
[22]

The idea behind a Long Short Term Memory network is similar to that of an RNN, with their outputs feedback. They are copies of the same network cascaded together as shown in figure 3.7. Each network in an LSTM architecture consists of several layers each having a specific operation. There are three gates namely *forget gate*, *input gate* and an *output gate*, and a *memory cell* in this network [22] that contribute to remembering the previous sequences and predicting an output as shown in figure 3.8.

With reference to figures 3.8, the top most horizontal path is called the *cell state*. It is basically a channel that carries information provided to it by the gates. All the gates are sigmoid function layers. The input gate controls the amount of new information that should be stored in the memory cell, while the forget gate decides how much of information should be discarded from the memory based on the previous and current inputs, and as well as the memory information from the previous network.



(a) The forget gate.



(b) The input gate.

Figure 3.8: The operation of forget and input gates [22]

The sigmoid function for the forget gates is described by the formula :

$$f_t = \sigma(W_f \times ([H_{t-1}], x_t) + b_f) \tag{3.8}$$

W_f represents the weight of the forget gate, while H_{t-1} and X_t represent the output of the previous network and the input to the current network. Lastly, b_f is the bias value of the forget gate. This formula will output either a '1' or '0' which will then later get multiplied with C_{t-1} to determine whether the value from the previous cell state should be retained or not. Similarly for the input gate,

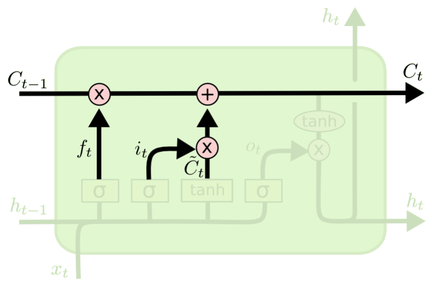
$$i_t = \sigma(W_i \times ([H_{t-1}], x_t) + b_i) \tag{3.9}$$

Since the input gate controls the information to be added, the inputs to the network are passed through a hyperbolic tan function which generates a new set of information, C'_t which is later multiplied with the sigmoid function of the inputs. In other words,

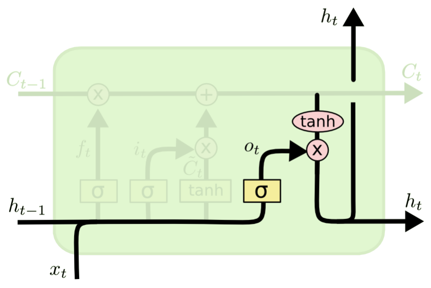
$$C'_t = \tanh \times (W_c \times [H_{t-1}, x_t] + b_c) \tag{3.10}$$

Equations 3.9 and 3.10 now update the cell state of the current network as seen in equation 3.11 and figure 3.9a,

$$C_t = (i_t \times C'_t) + ((f_t \times C_{t-1}) \tag{3.11}$$



(a) cell state.



(b) output gate

Figure 3.9: Updating cell state and predicting output
[22]

Now with information present in the cell state, the network needs to output its prediction and this is done through the output gate. As seen in the figure 3.9b, the information in the cell state is first passed through a hyperbolic tan function and in parallel, the inputs the network are passed through another sigmoid function with respective weights and biases. Finally, the output of this sigmoid function and the output of the hyperbolic tan function are multiplied. This product, $\mathrm{H_t}$ is the output prediction of the network which will then be sent to the next network. Equations 3.12 and 3.13 explain the same mathematically.

$$O_t = \sigma(W_o \times [h_{t-1}, x_t] + b_o) \tag{3.12}$$

$$H_t = O_t \times \tanh(C_t) \tag{3.13}$$

To summarize, a theoretical understanding of basic classifiers implemented were explained. An introduction to neural networks such as feedforward and recurrent neural networks were provided along with why the need for it. With reference to this, the practical implementation in the MATLAB environment has been done in chapter 4. The following chapter will discuss about the Advanced Peripheral Bus protocol. Lastly, all the equations in this subsection were referred from [22].

3.2 Advanced Peripheral Bus

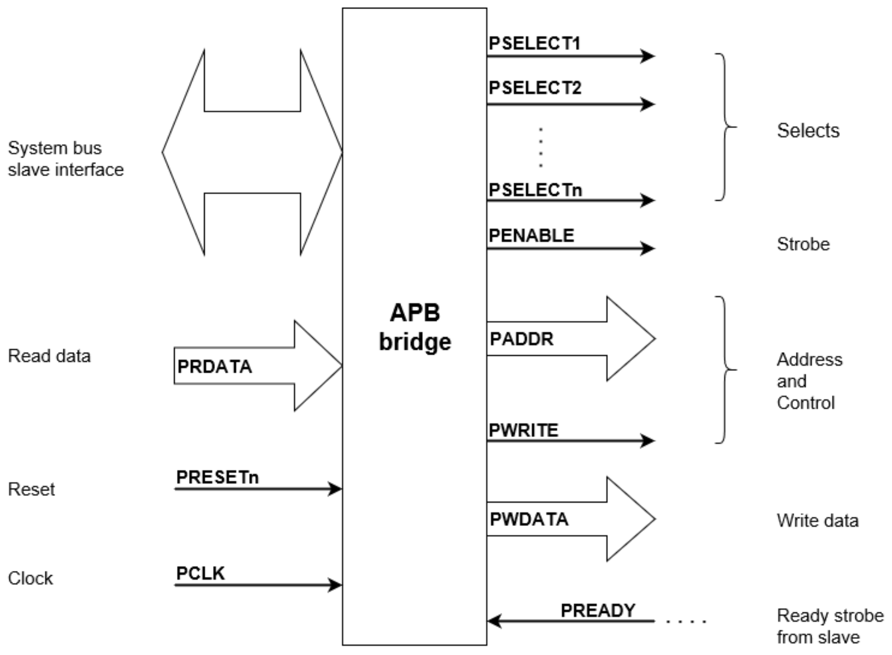


Figure 3.10: APB Master Interface.

Image source: [1]

Advanced Peripheral Bus (APB) [1], a member of the *Advanced Microcontroller Bus Architecture (AMBA)* family, is a bus architecture which handles write and read operations. Each operation is sampled at the rising edge of the clock and takes a minimum of two clock cycles. An APB consists of a *Master* and a *Slave* interface. Few of the main features are low power consumption, low-cost interface and low complexity and is mainly used to interface for low bandwidth peripherals. Apart from the APB, *Advanced High Performance Bus (AHB)* and *Advanced Extensible Interface (AXI)* along with their respective variants are also a part of the AMBA family.

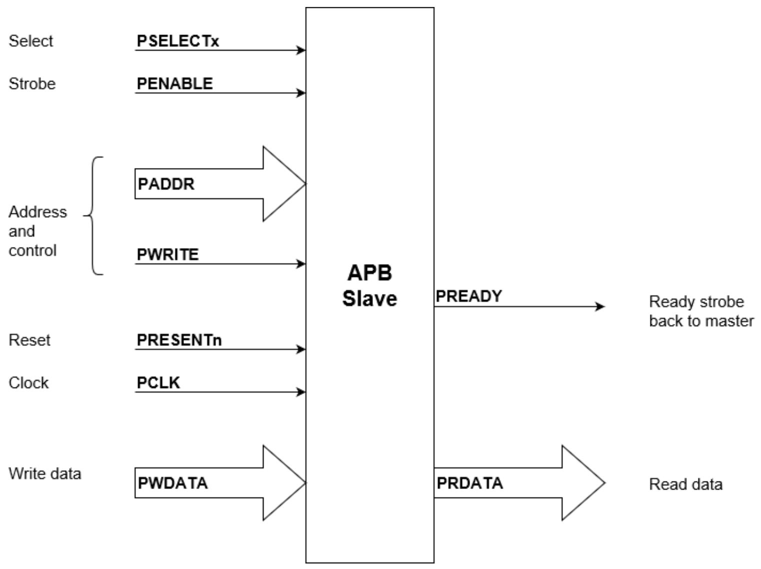


Figure 3.11: APB Slave Interface.
[1]

The operation of the APB is explained in detail in the next section.

3.2.1 State Machine

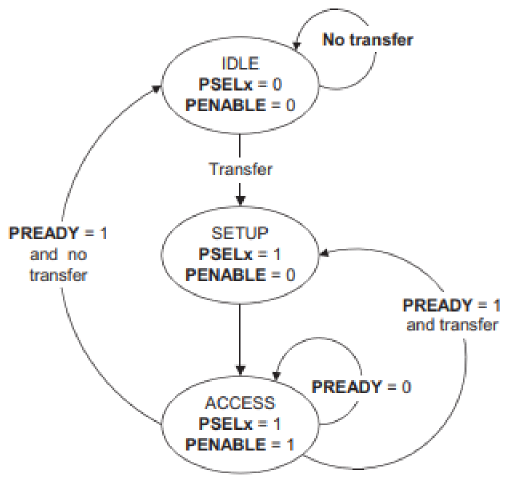


Figure 3.12: State machine of an APB.
[1]

The operation [1] of the APB mainly begins with the *IDLE* state every time a transfer request is raised it goes to the *SETUP* state. For every APB, there can be multiple slave interfaces and in order to initiate with the transfer request, the master needs to select a respective slave and to do so, a handshaking signal, `PSELECT` is triggered HIGH. Consequently in the next rising edge, APB enters the *ACCESS* state. When in this state, the master triggers another signal, `PENABLE` which enables the slave. `PENABLE` signal should be HIGH only when in access state. Also during this state, a signal from the slave to the master, `PREADY` is triggered LOW as long as the transaction is in progress. Once the slave has completed the transaction, the `PREADY` signal is triggered HIGH indicating that it has one, completed the transfer and two, is ready for the next transfer request. Also, during that one clock cycle, `PENABLE` will remain HIGH. If there are no further transfers then the APB returns to the *Idle* state or returns to *Setup* for further transfer(s) and the process repeats. These three signals define the behavior of the slave. The following sections will explain the type of each transaction with the help of another important signal, `PWRITE`.

3.2.2 Write Transaction

3.2.2.1 No-Wait Condition :

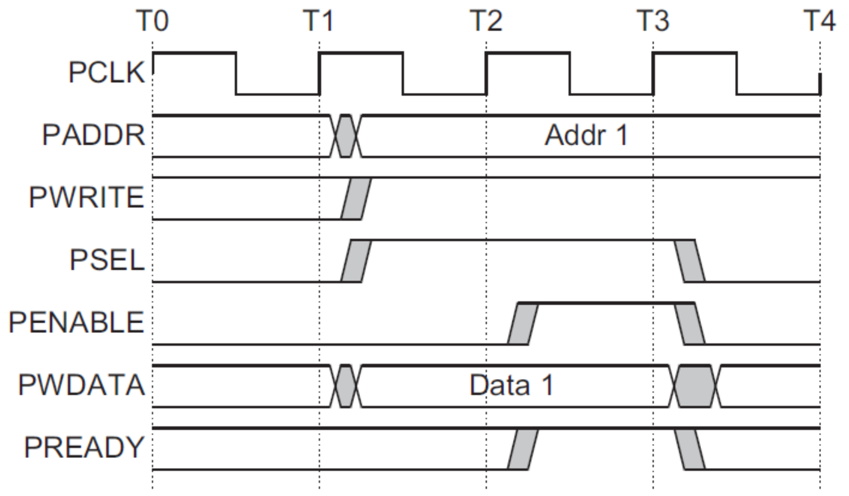


Figure 3.13: A Write Operation with No-Wait.
[1]

To execute a write transaction request, the corresponding signal `PWRITE` needs to be triggered HIGH which can be observed at clock cycle T1 from figure 3.13. Also at T1, `PSELECT` (*PSEL*) is triggered HIGH while the `PENABLE` is LOW. So, in comparison with figure 3.12 the APB at clock cycle T1 is in *Setup* state.

During the clock cycle T2, signals `PREADY` and `PENABLE` are triggered HIGH. This indicates that the APB is now in *Access* state and since `PREADY` is HIGH, the write transaction ends here. The `PWDATA` gets written in the respective `PADDR`

(*Address*) value. The minimum number of clock cycles required for an APB transaction is two.

3.2.2.2 Wait Condition :

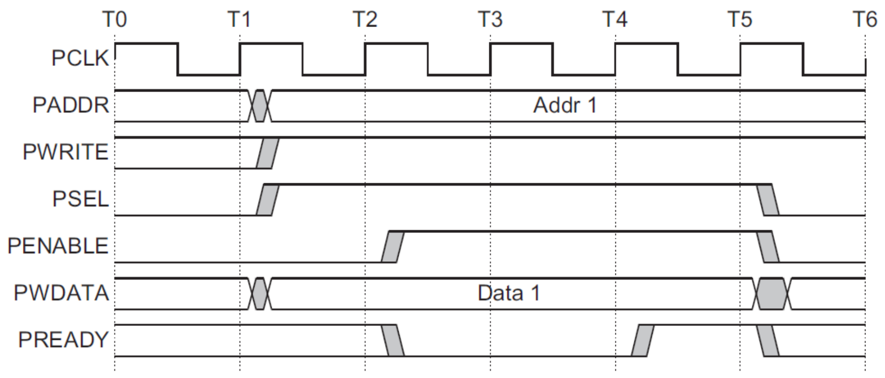


Figure 3.14: A Write Operation with Waits.
[1]

The Wait condition works in similar fashion as compared to the No-Wait condition. The APB enters the *Setup* state in clock cycle T2 as shown in figure 3.14. In the consequent clock cycles, `PENABLE` goes HIGH and `PREADY` is LOW from T2 till T4. Therefore, for two clock cycles the APB is waiting to complete the transaction. At T4, `PREADY` goes HIGH indicating that the slave has completed the write transaction. While the minimum number of clock cycles for a transaction to complete is two, there is no fixed number of maximum clock cycles it requires to complete the transaction.

If there exists any undesired toggling during the transaction, the APB protocol breaks and hence the transaction fails.

3.2.3 Read Transaction

3.2.3.1 No-Wait Condition :

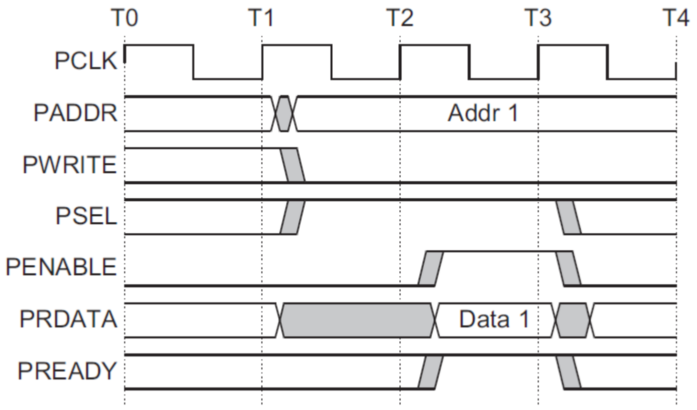


Figure 3.15: A Read Operation with No-Wait.
[1]

3.2.3.2 Wait Condition :

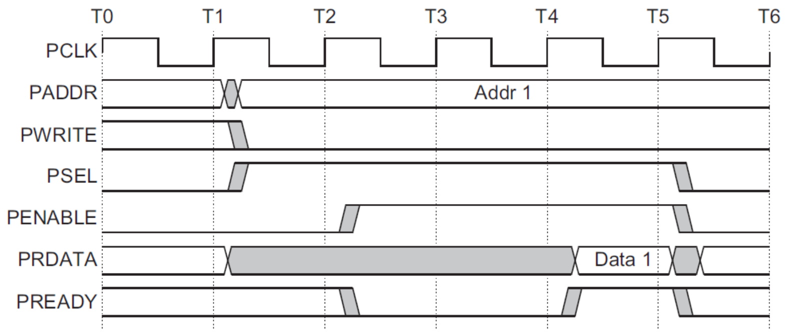


Figure 3.16: A Read Operation with Waits.
[1]

The read operations for both No-Wait and Wait condition is similar to the write operation, the only difference is in the `PWRITE` signal. While the write operation requires the `PWRITE` signal to be HIGH, for a read operation, it is expected to be LOW. `PSELECT`, `PENABLE` and `PREADY` have the same behavior as that of the figure 3.13.

3.3 SystemVerilog Assertions

3.3.1 Constrained Random Stimulus Generation

One of the traditional approaches of verifying a given hardware is to construct directed tests with tailor made stimulus to check the functionality of the same. These input stimuli are later simulated and verified manually through waveforms to compare the obtained results with the expected ones [28]. Generating input stimuli manually can be cumbersome for huge hardwares with complex logic. It requires not only time and labor, but the knowledge of all the possible inputs combinations. Generally in such situations, the chances of missing out on verifying the corner cases are quite high.

This is where randomization of input vectors comes into picture. Unlike, directed tests, randomized tests not only generate random stimuli, but the chances of testing the corner cases are quite high and lastly, this opens up new types of bugs that wouldn't have been covered with the traditional method. But again, completely randomizing the input vectors might slower the process of verification due its wide range of possible combinations [28].

```
initial
begin
  repeat(300)
  begin
  @(posedge clk);
       psel = $urandom_range(0,1);
       pwrite =  $urandom_range(0,1);
       penable =  $urandom_range(0,1);
       pready =  $urandom_range(0,1);
  end
end
```

In the above listing, our four main signals of interest are randomly generated. They have been defined to toggle either LOW or HIGH with the help of the SystemVerilog function `$urandom_range`. This function returns a flat distribution of bits in the unsigned format [28].

Constraining these random vectors with *weights* helps fasten the process [28]. In other words, these weights dictate the percentage of times a particular bit should occur. This increases the hits without sacrificing on other combinations.

```
//generating randomized values with weights.

initial begin
repeat (300)
begin
@(posedge clk);
std::randomize(psel) with {psel dist {0 := 1, 1 := 9} ;};
std::randomize(pwrite) with {pwrite dist {0 := 4, 1 := 6} ;};
std::randomize(penable) with {penable dist {0 := 5, 1 := 5}
    ;};
std::randomize(pready) with {pready dist {0 := 5, 1 := 5} ;};
end
```

```
end
```

In the above listing, the four signals are randomly distributed with respective weights for each bit. For example, `psel` has been defined such that the ’0’ occurs 10% while ’1’ occurs 90% of the simulation time. Similarly, `pwrite` has been defined with 40% of ’0’ and 60% of ’1’ occurrences. This is crucial as it is important to capture both types of the APB transaction, the write and read. By varying either of the percentages for `pwrite` will affect the number of writes and reads occurring during that simulation run. Likewise, `psel`, `penable` and `pready` will affect the number of occurrences of either of the transactions.

With these random inputs generated automatically with constraints, verifying them manually is very laborious. A simple yet, powerful and most used verification methodology, SystemVerilog Assertions can be put to use to capture the nature of every output obtained.

3.3.2 SystemVerilog Assertions

Simply put, assertions are way to check and debug the behavior of a design by defining constraints, conditions and checkers against the given specification over a period of time to ensure it never violates the same [19], [29], [4]. They are one of the extensively used verification methodologies to find bugs in designs as they have the ability to find the root of the problem. Assertions are written by design engineers and verification engineers, both. SimVision by Cadence and ModelSim are few of the tools that monitor assertions. Cadence SimVision was used to debug assertions for the thesis work. With reference to its user guide, the tool follows a state machine to output the status of the assertions. The states are *Inactive*, *Active*, *Finished* and *Failed*. These following states have a certain priority defined from Failed having the highest priority to Inactive having the least.

SystemVerilog Assertions (SVA) are two types, *Immediate* assertions and *Concurrent* assertions. The main difference is that, immediate assertions neither depend on clock or reset while concurrent assertions do. With concurrent assertions, they not only check the condition on every clock, but once triggered they can be constrained to check for a certain number of consequent clock cycles as seen in figure 3.17.

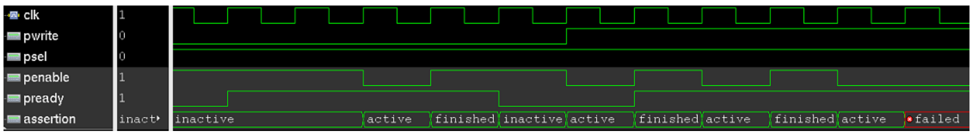


Figure 3.17: A simple waveform of APB with assertions.

Properties, *Sequences*, *Implication*, *Functions* and *Operators* are the main components for any given SVA [7]. For basic understanding, the following components have been explained.

- **Property**: Properties are a way of defining a condition based on sequences defined that if condition A has occurred then check if condition B has occurred in either the same clock cycle or in the next clock cycle(s). The following listing gives a brief understanding of the same.

```
//Definition of a property.

property example;
@(posedge clk) A |-> B;
endproperty

ExProp : assert property(example) else $warning("This
    property has failed.");
```

In the above listing, a property named 'example' has been defined. 'A' is called the *Antecedent* and 'B' is called the *Consequent*. The operator '|->', known as *overlapping* operator in between them implies that if an event 'A' has occurred at clock cycle T1(say) then event 'B' will also occur at the same clock cycle T1. Similarly, instead of '|->', a *non-overlapping* operator '|=>' can be used to check if the consequent occurs in following clock cycles after the occurrence of the antecedent. The *assert* keyword will check this property at every rising edge of the clock. A given property will be executed only if it is asserted. If the assertion fails then a warning message will be displayed on the console.

- **Sequence**: Sequences, as the name states, it is a definition of the behavior of certain signals.

```
//Declaring sequences.

sequence A;
aa and !(bb);
endsequence

sequence B;
cc and dd;
endsequence
```

From the above listing, a sequence A has been defined indicating that the signal *aa* has to be HIGH while signal *bb* has to be LOW at the same clock cycle. SVA allows you to define multiple signals within a sequence with the help of certain functions and operators with logical *and* being one of them. These sequences can later be in properties as shown in previous listing.

- **Functions**: SVA has certain defined functions that relate to toggling of bits of any given signal.

  - **$rose** : This function states that, for a given clock cycle, the signal should have toggled HIGH from LOW with respect to the previous clock cycle.
  - **$fell** : Similar to $rose, $fell checks if the signal has been toggled to LOW from HIGH with respect to the previous clock cycle.

- **$stable** : This function checks if the signal is stable for certain period of time. If there is any undesired toggling, it will output a fail.

The idea of using assertion based verification approach was one, to understand and compare their effectiveness with respect to machine learning algorithms and two, as the task was a classification based problem, assertions were implemented against the behavior of APB to generate an output label for the training data.

As effective as they are, writing assertions aren't simple. As the complexity of the design increases, defining properties and sequences become quite tricky due to various behavioral constraints and most importantly, identifying and verifying corner cases is quite tedious.

Please refer to the appendix for the SystemVerilog Assertion code written for this thesis work.

Chapter 4

Implementation

4.1 Block Diagram

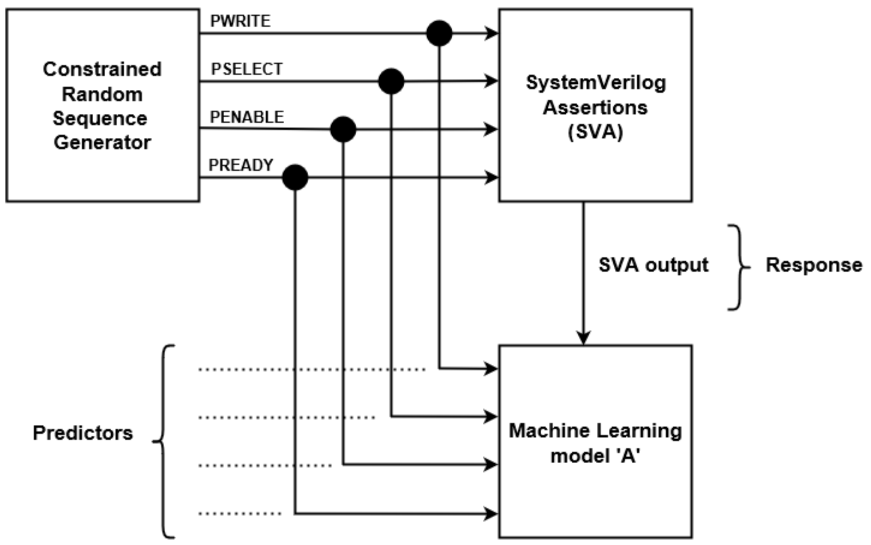

Constrained Random Sequence Generator
PWRITE
PSELECT
PENABLE
PREADY
SystemVerilog Assertions (SVA)
SVA output
Response
Predictors
Machine Learning model 'A'

Figure 4.1: Training a model

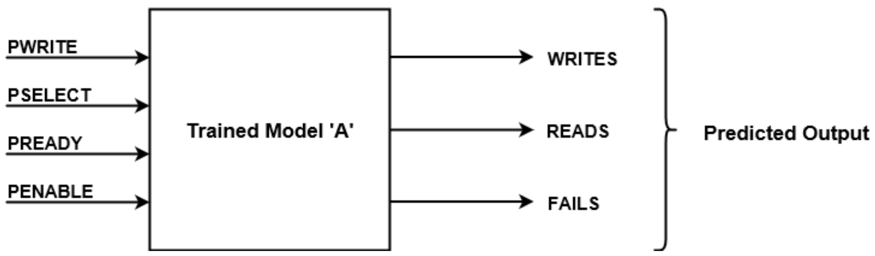


Figure 4.2: Testing a model

Figure 4.1 shows the flow of the implementation. First, random sequences are generated which are then asserted in order to obtain the output class for the training model. The randomized sequences along with the assertion output are used to build the training model. Based on the needs of the algorithm, these sequences are first prepared and modified before constructing the model. Data preparation has been explained in the following sections.

With the training model constructed, the test data is now fed to check the accuracy.

4.2 Data Preparation

For any given machine learning application, data acts as a catalyst and is a major contributor in building a good model. It dictates the performance and efficiency of the same. With that being said, complex problems entails huge amount of data, a small percentage of which might be redundant. The first step in building a training model is to scale the critical data which forms the base of data preparation or *Feature Extraction*.

4.2.1 Classifiers and FeedForward Neural Network

To begin with, randomly generated waveforms asserted with the APB behavior are extracted from Cadence SimVision tool. As described in chapter 4, the signals of interest are `PWRITE`, `PSELECT`, `PENABLE`, `PREADY`. The following figure shows a waveform that has been exported to .csv format.

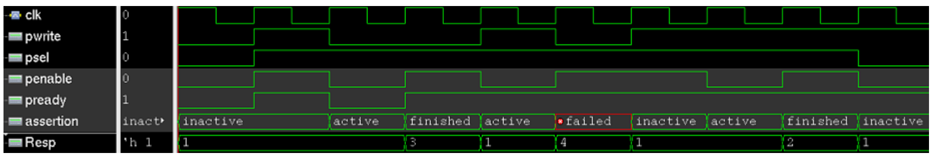


Figure 4.3: A simple waveform of APB with assertions.

Every signal extracted has been sampled at every rising edge of the clock. Table 4.1 is an excerpt from one of the .CSV files exported. Each row in the table represents nth clock cycle. Column `Resp` is the output response for our training model. As seen in the table below, the output response class varies from '1' to '4'. These numbers are equivalent numerical representation of the APB behavior with respect to SVA, in other words, *Classes*. '1' stands for Inactive state, which means that the APB is either in its idle or setup state. Access state of the APB is represented by '2' and '3' with '2' representing a write transaction while '3' stands for a read transaction. class '4' represents a fail case, the point where the APB failed to switch to access state due to undesired toggling of bit(s) or failed to stay in access state.

SimTime	clk	pwrite	psel	penable	pready	Resp
0	1	0	1	0	0	1
5	0	0	1	0	0	1
10	1	0	1	1	1	3
15	0	0	1	1	1	3
20	1	1	1	0	1	1
25	0	1	1	0	1	1
30	1	0	1	1	1	4
35	0	0	1	1	1	4
40	1	1	1	1	1	1
45	0	1	1	1	1	1
50	1	1	1	0	1	1
55	0	1	1	0	1	1
60	1	1	1	1	1	2
65	0	1	1	1	1	2
70	1	0	1	0	0	1
75	0	0	1	0	0	1

Table 4.1: An example of the csv file.

Table 4.1 was one of the raw data obtained and from this, the important features were extracted. The data points of interest are the classes '2', '3' and '4', while the rest are noise. The columns `clk` and `SimTime` do not contribute much to the training model as well since all the signals were sampled and extracted at positive clock edges. Hence, these two columns were altogether deleted. The resulting table is as shown below.

pwrite	psel	penable	pready	Resp
0	1	0	0	1
0	1	1	1	3
1	1	0	1	1
0	1	1	1	4
1	1	1	1	1
1	1	0	1	1
1	1	1	1	2
0	1	0	0	1

Table 4.2: Modifying the table.

Simulation time ranging from 0 nanosecond to thousands of nanoseconds were considered for both training and testing purposes. As explained in 4, an APB transaction consumes a minimum of two clock cycles irrespective of a wait or a no wait condition, there is a dependency of the nth row with the (n-m)th row, where $m \geq 1$. In other words, the values in the current clock cycle are dependent on the values present in the previous clock cycle(s). In order to achieve this, the columns were rearranged such that each row contained the values of the current and the previous clock cycle as shown in table 4.3. The response variable now has the value that of the current clock cycle and not that of the previous clock cycle. The last step in data preparation was to remove the rows for class '1' or invalid class as it training those data points had no relation with the behavior of the APB.

Current clock cycle				Previous clock cyle				
Cwrite	Csel	Cenable	Cready	Pwrite	Psel	Penable	Pready	CResp
0	1	1	1	0	1	0	0	3
0	1	1	1	1	1	0	1	4
1	1	1	1	1	1	0	1	2

Table 4.3: Rearranging the table.

For basic classifiers, the table 4.3 after data preparation can be fed directly where later it is split into training and test sets. But, when it comes to FeedForward or PatternNet(section 4.3.3) in particular, MATLAB expects this table to be converted into a matrix and the output response be converted from numerical indices to its equivalent vectors as shown below.

$$\begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

This preparation method works well irrespective of the algorithm for a no-wait condition of an APB operation provided it is affirmative that the protocol behaves in such fashion constantly. On the other hand, preparing data for a wait state condition of an APB is challenging as there aren't any predetermined or constant number of wait states. The only to way to proceed with this preparation method for unknown wait states is to delete the n-intermediate wait states which is basically creating a hybrid model by forcibly converting all the wait conditions into a no-wait condition. The main disadvantage is, the data is now quite straightforward and deterministic which overrides the need for machine learning. In order to overcome this, an LSTM network was built.

4.2.2 LSTM

For LSTMs, the data preparation method is different when compared with the rest. Unlike the basic classifiers and FFNN, LSTMs basically operate on batches or groups of data. Every row and a column in a batch contributes to the learning curve of the model. The advantage of such a network is that all important data can be considered without any deletion.

The data for LSTM consists of 3 main components. One, the *batch size*. Two, the *time steps* and three, *feature dimension.* Consider the following matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The orientation of data is from left to right, as in 1st column is the first clock cycle, 2nd column is the second clock cycle and so on. Each column of the above matrix is called a time Step while the number of rows represent to the feature dimension. Each row represents a signal, `PWRITE`, `PSELECT`, `PENABLE`, `PREADY` in that respective order. The above matrix can be compared to the following figure.

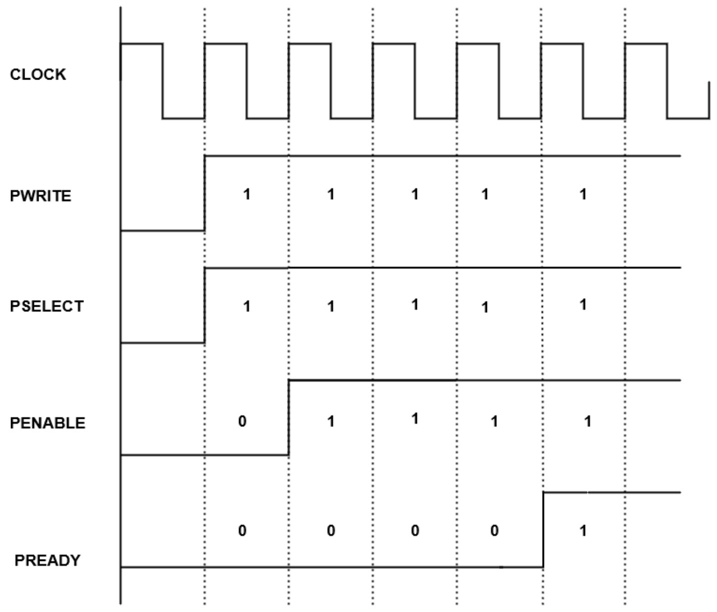


Figure 4.4: Matrix representation of the waveform.

Both, feature dimension and time steps put together result in a sequence or batch size. The above matrix is an example of an APB write transaction with 3 wait states. The entire matrix is called a *Sample* or a *Sequence.* Similarly multiple matrices can be sliced from the waveform and stored in an array along with its response class. For this thesis work, such matrices were created manually for training purposes only, with a combination of patterns. The waveforms were extracted only for testing purposes.

4.3 MATLAB Implementation

In this section, a few important methods and functions used to implement machine learning algorithms in MATLAB have been briefly explained.

First step is to import a complete dataset and later split the obtained data in to training and test subsets. This process is called *Cross Validation* and the function in MATLAB is called `cvpartition`. It creates a cross-validation partition for the given data [13]. General approach in order to split the data is to either have a 60 to 40% or a 70 to 30% ratio of train versus the test data. The `HoldOut` is an input name-pair argument for the `cvpartition` function where the user can input the desired ratio as shown in the following listing.

```
%% Defining the splitting method.
mytable = readtable('apb_nowait.csv')

pt = cvpartition(mytable.Resp, 'HoldOut', 0.65);
traindata = mytable(training(pt),:);
testdata = mytable(test(pt),:);
```

In the above listing, the .csv file is first imported. The `cvpartition` function is called to split the data. The 'mytable.Resp' denotes the response column and based on that, 65% of the data is split for training while the remaining 35% is for testing. The data is split randomly. The `training` and `test` returns the respective indices and stores them in the defined variable.

Due to its randomized nature of splitting, this method turns out to be a disadvantage. Our required data for training purposes will get mixed with test set which will reduce the quality of the model. The approach for this thesis work in terms of data preparation is to have one separate set for training that should be altered only when desired and a separate set of testing data.

In order to overcome this, data was prepared separately in forms of table and directly exported as explained in section 4.2

4.3.1 KNN

In MATLAB, to build a classifier model, the function `fitc*` is used. '*' here represents various types of classification algorithm. For KNN, `fitcknn` is a function that fits a k-nearest neighbor classification model. For example,

```
%%% Defining a knn model
myKNN = fitcknn(PredictorTable, ResponseVariableName);
```

Here, *myKNN* creates a KNN model taking account of the predictors from the table, *PredictorTable* containing various observations with respect to the response variable, *ResponseVariableName* which were obtained during the data preparation stage. There are several optional parameters that can defined one such being `NumNeighbors`. This parameter controls the search vicinity around a new test observation as explained in chapter 3, section 3.1.1.1.

```
%%% Defining a knn model with max number of neighbors.
myKNN = fitcknn(PredictorTable, ResponseVariableName, '
    NumNeighbors', 5);
```

Here the maximum number of neighbors has been defined as 5 which means the classifier will search for 5 nearest neighbors and based on the search and the majority of the classes, the class will be assigned. As described in 3.1.1.1, `Distance`, an optional input name pair describing the distance measure between the unknown and the known point can be defined.

```
%% Defining distance measure
myKNN = fitcknn(PredictorTable, ResponseVariableName, '
    NumNeighbors', 5, 'Distance', 'euclidean');
```

Using the above training model, `myKnn`, predictions can be made using the defined function `predict` for the test data which has been previously held out.

```
%%% Predicting outputs
PredOutput = predict(model, testdata);
```

For any given model, there exists some amount of loss, both for training and test. `resubLoss` in MATLAB stands for Resubsitution Loss. It calculates the training loss of the model defined, i.e, `myKNN`

```
%% Calculating the train loss
trainLoss = resubLoss(model);
disp(['The training error is : ', num2str(trainLoss)])
```

Likewise, the `Loss` function calculates the test loss of the test data with respect to the training model defined.

```
%% Calculating the train loss
testLoss = loss(model, testdata);
disp(['The test error is : ', num2str(testLoss)])
```

4.3.2 Decision Tree

Similar to KNN, there are several MATLAB functions [17] employed to build a decision tree. `fitctree` is a function that returns a binary fitted classification tree. For example,

```
%%% Defining a tree model
myTree = fitctree(PredictorTable, ResponseVariableName);
```

Here, *myTree* creates a binary classification tree using the function `fitctree` taking account of the predictors from the table, *PredictorTable* containing various observations with respect to the response variable, *ResponseVariableName*. As explained in section 3.1.1.2, the depth of a given can be controlled and that it can be done optionally with the help of the function, `MaxNumSplits`.

```
%%% Defining a tree model with maximum number of splits
myTree = fitctree(PredictorTable, ResponseVariableName,  '
    MaxNumSplits', 5);
```

Here the maximum number of splits has been defined as 5 which means the maximum number of branch nodes will be 5. Decision trees are quite sensitive to the data and at times tend to overfit. In order to avoid this, a method called *Pruning* is used. It is a process where the size of the tree is shortened by eliminating the branches that do not provide the power to classify instances. `Prune` is an optional parameter in MATLAB, , which can be turned 'on' or 'off'.

```
%%% Defining a tree model with pruning
myTree = fitctree(PredictorTable, ResponseVariableName,  '
    MaxNumSplits', 5, 'Prune','on');
```

4.3.3 Patternnet

`Patternnet` is a pattern recognition neural network which is a variant of the FFNN in MATLAB. This network was used as the study was related to pattern recognition. The following snippets will give a brief explanation on the algorithm.

First, the dataset is exported after preprocessing the .csv file extracted from the simulations. Preprocessing or the data preparation will be explained in chapter 3.3.

```
load PatternAPBWriteData
XWrite = WinputsTr;
TWrite = WtargetsTr;
```

`XWrite` contributes to the input neurons. Since there are 4 features, there will be 4 input neurons. `TWrite` holds the responses for respective observations from `XWrite`. The classes in `TWrite` will define the number of output neurons. A portion of these variables shall be later used to train and test the model.

The next step is to define the number of hidden layers and neurons by using a MATLAB function `patternnet`.

```
%defining one hiddenlayer with 'm' number of neurons.
WriteNet = patternnet(5);
```

Here, a single hidden layer with 5 neurons has been defined. If required, the number of layers and the neurons can be varied. It is as shown in the following snippet.

```
%defining two hiddenlayers with 'm' and 'n' number of neurons.
WriteNet = patternnet([8 5]);
```

For the thesis study, only one hidden layer was chosen.

Now with the data imported and the hidden neurons defined, next step is to select the training algorithm and the percentage of data which will be used for training and testing.

```
%defining the network parameters
WriteNet.trainFcn = 'trainscg';
WriteNet.divideFcn = 'divideind';
WriteNet.divideParam.trainInd = 1:250;
WriteNet.divideParam.testInd = 251:988;
```

`WriteNet.trainFcn` defines the training function for this particular network. The default training function is Scaled conjugate gradient backpropagation, *trainscg*. `WriteNet.divideFcn` parameter decides how the data should be split. Here, by using *divideind* the train and test data are split based on the indices or rows as shown in the above snippet. The advantage of this over `cvpartition` is that data is not split randomly. Apart from *divideind*, there are several other ways to split the given data set, *dividerand*, *divideint* and *divideblock* are few of them.

The next step is to train and test the network using the above mentioned parameters.

```
%training the network
[WriteNet, Writetr] = train(WriteNet, XWrite, TWrite);
```

Here, `Writetr` basically holds the training record of parameters such as the data division, performance and etc. `train` on the other hand, calls the training function defined in the variable `WriteNet.trainFcn` which, as defined previously is *trainscg* and using this function, it accesses the `WriteNet.divideParam.trainInd` and `WriteNet.divideParam.testInd` to train and test the model. The model here is `WriteNet`. Last but not the least, the outputs are extracted and the performance is calculated.

```
%obtaining the outputs
Wouts = WriteNet(XWrite);
%calculating the performance.
WritePerformance = perform(WriteNet, TWrite, Wouts);
```

4.3.4 LSTM

In MATLAB [14], the LSTM architecture is defined layer by layer. As usual, the data set is imported.

```
load trainlstmdata

XTrain = Xtrain;    %% Train vectors
YTrain = Ytrain;

load testlstmdata

XTest = Xtest;      %% Test vectors
YTest = Ytest;
```

The first layer in the architecture is the `sequenceInputLayer`. This layer basically inputs the sequence data and sets the input size for the network. The second layer is the `lstmLayer` which learns the long term dependencies between sequences. This layer takes in a numeric input which defines the number of hidden units property, `NumHiddenUnits`. This property corresponds to the amount of information it can contain or remember between sequences. This layer also consists of another input parameter that decides the type of sequence prediction i.e., `OutputMode`. It can either predict an output for every sequence or for a group of sequences. In order to select either of them, `OutputMode` can be toggled between `sequence` or `last`.

```
%% Creating layers for the network

layer1 = sequenceInputLayer(4)
layer2 = lstmLayer(30, 'OutputMode', 'last')
layer3 = fullyConnectedLayer(3)
layer4 = softmaxLayer()
layer5 = classificationLayer()

%% Merging all the layers into one.
FinalLayer = [layer1; layer2; layer3; layer4; layer5]
```

`Layer3` connects `Layer2` and `Layer1`. The `FullyConnectedLayer` requires the output size to be defined which is nothing but the number of classes for the network to predict. In the above listing since our output classes are 3, it has been initialized with the same. The `softmaxLayer` applies the softmax function and the `classificationLayer` calculates the cross-entropy loss for the given number of classes.

Now with the basic structure built, the next is to instruct how the architecture should behave. For that, a function called `trainingOptions` is used [18].

```
%% Defining the Options for training

maxEpochs = 70;
miniBatchSize = 5;
learningrate = 0.01;
options = trainingOptions('adam','ExecutionEnvironment','auto'
    ,'GradientThreshold',1,'MaxEpochs',maxEpochs, ...
                            'InitialLearnrate',learningrate,'
    MiniBatchSize',miniBatchSize,'Shuffle', 'never', ...
                            'SequenceLength', 'longest','Plots
    ','training-progress');
```

An `Epoch` is defined as the number of times the entire dataset has been utilized completely to train and test the model. `MiniBatchSize` defines how many number of minibatches are supposed to be created. Minibatches are created to split the training data into smaller batches to fasten the training process. The minimum number of mini batch is 1. Due to this splitting, some sequences might get split in halfway or need extra sequences to maintain the size and in order to handle this, a name-value pair argument, `SequenceLength` is defined with either 'shortest' or 'longest'. While 'shortest' truncates the sequences, 'longest' pads extra dummy data which can act as noise. Hence, this property should be carefully used.

The most important training parameter is `InitialLearnrate`. As the name suggests, it controls the rate at which information is learned over time. This parameter also decides the time it takes to train the model. Smaller the learning rate, longer it takes to train. There are various other optional paramters which can be referred from [18].

With the structure and the training behavior defined, the next step is to train and test the model as shown in the following listing.

```
%% Training the network
ApbNet = trainNetwork(XTrain, YTrain, FinalLayer, options);
save trainedlstm

%% Testing the network
ApbPred = classify(ApbNet, XTest,'MiniBatchSize',miniBatchSize
    , 'SequenceLength', 'longest');

%% Creating a confusion matrix
Result1 = confusionmat(YTest, ApbPred)

%% calculating the accuracy
acc = sum(ApbPred == YTest)./numel(YTest)

save lstmresults
```

The `trainNetwork` function takes in the predictors and the responses along with the layers defined and the training options set. The training, in general takes around 5-30 minutes based on the execution environment, number of observations in the training data, the minibatch size, the epochs and the learning rate. With the help of this trained network, *ApbNet*, the test vectors are now classified and their accuracy is calculated. A confusion matrix is also generated for better understanding.

Chapter 5

Results

Accuracy, as the name speaks for itself, the most important parameter that decides how reliable the model is. Accuracy varies from algorithm to another by a certain margin. For classifiers such as KNN and Decision Trees, the accuracy is based on the percentage of the data split for training and testing and other respective parameters such as neighbors and branches. While for neural networks, important parameters such as epochs, learning rate and hidden neurons play an important role apart from data partition. Figure 5.1 is a comparison of average accuracy obtained for each implemented algorithm. For every given algorithm, the prediction accuracy gradually increased as and when the training data along with the various parameters were moderated.

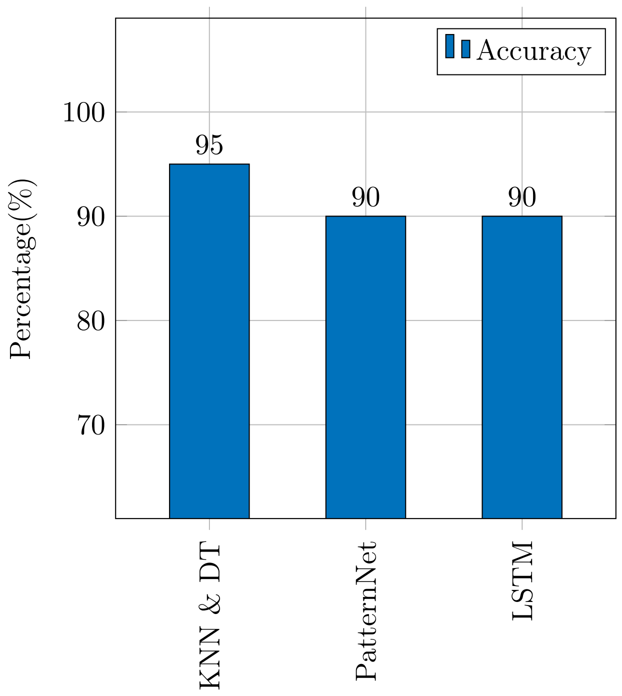
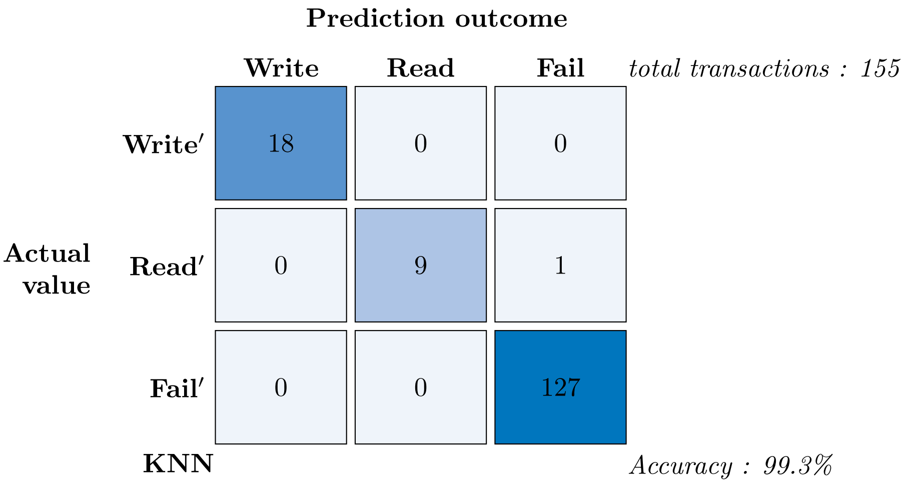


Figure 5.1: Accuracy.

5.1 Classifiers and PatternNet

5.1.1 KNN and DT

Multiple simulations consisting of various APB transactions for obtaining training and test data sets were performed. Due to the method of data preparation as explained in chapter 4, section 4.2.1 the accuracy obtained for KNN and Decision Trees were extremely high. For certain cases, the accuracy obtained was nearly 100%. To put in perspective, there were 0 to a few misclassifications for certain test data. The following confusion matrix from one of the test data gives a better understanding.



Prediction outcome

total transactions : 155

Actual value		**Write**	**Read**	**Fail**
	Write′	18	0	0
	Read′	0	9	1
	Fail′	0	0	127

KNN *Accuracy : 99.3%*

Given the data preparation for KNN and DTs, there exists a direct mapping of features with the class which makes it very simple for the model to classify the new observation. Considering the following table for better understanding.

Current clock cycle				Previous clock cyle				
Cwrite	Csel	Cenable	Cready	Pwrite	Psel	Penable	Pready	Resp
1	1	1	1	1	1	0	1	1
0	1	1	1	1	1	0	1	3

Table 5.1: Understanding direct mapping of data.

The sequence for current clock period is [1 1 1 1] and the sequence for the previous clock period is [1 1 0 1], the output class is '1' which indicates the write operation, should there be a change in a single bit at the least, the class assigned to it is '3', which is a failed transaction. Such a simple and straight forward mapping led to high accuracy for both the classifier algorithms.

5.1.2 PatternNet

PatternNet on the other hand, albeit had the same data preparation method, there was a drop in the accuracy. The highest accuracy obtained for this algorithm was 97.8% with 7 hidden neurons and 57 iterations of epochs. Image 5.2 shows the schematic of the neural network created by MATLAB along with the confusion matrix below.

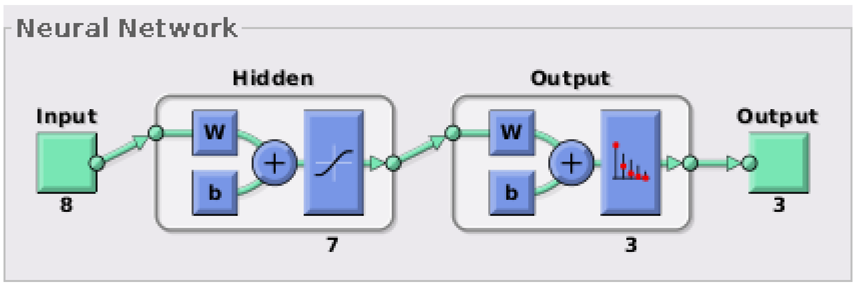


Figure 5.2: Structure of the Neural Network.

With reference to 5.1, PatternNet uses the similar type data sets where there exists a strict direct mapping of features to classes. Comparatively, apart from the data set, it is the structure of the algorithm itself that impacts the accuracy. This is mainly due to the training algorithm used and neurons set. The number of input neurons are fixed based on the features selected and the output neurons based on the number of output classes, but the hidden neurons are to be adjusted based on a trial and error method.

With patternNets being more complex relatively to build and train, it can be concluded that the use-case of this algorithm for the DUT at hand in relation with the dataset is not only unnecessary but rather an overkill. In other words, the outputs can still be predicted with similar datasets by employing the predecessors which are less complex in nature.

		Prediction outcome			
		Write	**Read**	**Fail**	*total transactions : 135*
Actual value	**Write′**	18	0	3	
	Read′	0	9	0	
	Fail′	0	0	105	
	FFNN				*Accuracy : 97.8%*

Lastly, a main disadvantage of using the above three algorithms is the loss of valuable data, the wait states. With the exclusion of the wait states, an incomplete behavior of the APB is trained and predicted. This exclusion also narrows the range of dataset which in a way or two, trains almost all possibilities of the features.

To summarize, taking note of the disadvantages, the above algorithms had to be sidelined and the LSTM algorithm, which has the ability to handle time series data was employed. The following section discusses the results obtained for the same.

5.2 LSTM

From figure 5.1, the average accuracy obtained for LSTM is 90%. LSTMs are complex in nature and many of its network paramters contribute to the accuracy. Similar to PatternNets, hidden neurons in an LSTM matter as well. To begin with, the training and test data set initially included small groups of both Write and Read transactions, of both the No-Wait and the Wait state variants. The following confusion matrix was obtained from one of the tests that were performed with parameter values *Epoch = 40*, *Hidden Neurons = 25* and *Learning Rate = 0.001*. The test data consisted a total of 60 transactions out of which 7 were misclassified, the accuracy obtained was 88.33%.

Prediction outcome

Total transactions : 60

		Prediction outcome		
		Write	**Read**	**Fail**
Actual value	**Write′**	0	3	1
	Read′	0	11	0
	Fail′	0	3	42

LSTM1 *Accuracy : 88.33%*

As the training data progressed quantitatively, the hidden neurons were adjusted along with the network parameters. Increasing the number of hidden neurons does not always guarantee higher accuracy. At the same time, lowering the hidden neurons does not dampen the accuracy as well. Fewer hidden neurons lead to *Underfitting* while too many might lead to *Overfitting*. The highest percentage of accuracy obtained was 90% for a randomly generated test data consisting multiple Write and Read transactions with various wait states. This test was performed with parameter values *Epoch : 40*, *Learning Rate : 0.001* and *Hidden Neurons : 30*. The following confusion matrix briefly depicts the outcome.

Prediction outcome

		Write	**Read**	**Fail**	*Total transactions : 60*
	Write′	2	1	1	
Actual value	**Read′**	0	11	0	
	Fail′	0	4	41	
	LSTM2				*Accuracy : 90%*

Apart from hidden neurons, few network parameters such as the epochs, learning rates, sequence length and mini-batches played an important role as well. As explained briefly in chapter 4, epochs are the number of times the network accesses the training data from start to end for learning while learning rates indicates the rate at which the network learns the features as in the time taken for it learn all the observations. Similarly mini-batches are employed to split the training data into smaller batches to facilitate the batch-wise learning and process it faster. The down side with mini-batches is for a given size of a training data set, if the number of batches are too high then it might lead to data loss as it will slice it into smaller groups.

With the LSTM neural network being the preferred algorithm, the main challenge was to improvise the accuracy of the model. The key factor for a better accuracy is the nature of the data and in this case, it is the number of Wait states. An important criteria of selecting the training data was to ensure that the APB transactions of various operations were approximately equally distributed. By doing so, the classifier learns the behavior in order to make a fair classification. If there existed more number of observations for one particular class in comparison with the rest, it would lead to improper learning as the classifier would fail to distinguish new observations. This imbalance is called *Undersampling.*

Apart from accuracy and complexity, the training computation time and the effort required to prepare the data is important too, a comparison of which can be found in the following section.

5.3 Comparison of Algorithms

Algorithm	Complexity of ML model	Data Preparation effort
KNN & Decision Trees	Low	Low
PatternNet(FFNN)	Medium	Medium
LSTM	High	High

Table 5.2: A brief comparison of various models.

Algorithm	Implementation time	Training and Testing time
KNN & Decision Trees	Low	Low
PatternNet(FFNN)	Medium-High	Medium
LSTM	High	High

Table 5.3: A time comparison of various models.

The complexity of various models were compared based on the number of parameters required to implement a trained model of APB with the nature of respective datasets. For example, KNN was much easier to implement mainly as the data prepared for it couldn't capture the entire behavior of the APB. An alternative was to exclude the wait states of APB which made the dataset simpler for KNN model to learn requiring fewer parameters such as the number of neighbors to be searched and the type of measuring distance between them. This resulted in lower implementation time in terms of setting up the model and, training and testing it.

On the other hand, LSTM itself is a complex neural network which is mainly comprised of multiple layers combined along with various training parameters as explained in chapter 4. For such a model, preparing data consumes a lot of time and effort as it mainly focuses on feature extraction and scaling them to meet the model's requirement. Based on the quality and the size of the datasets, the network layers and the training parameters need to be adjusted in iteration until the desired accuracy is obtained. For any user, a pre-requisite is to have a considerable amount of fundamental knowledge about LSTM and its respective parameters. This contributes to the overall implementation time of this network.

With this understanding of how machine learning can be employed in the field of hardware verification, the following chapter summarizes the thesis work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The aim of the thesis was to investigate how the concepts of Machine Learning could be used to verify an APB protocol. APB protocol is widely used in modern SOCs to facilitate the communication between IPs, hence it was chosen as the DUT.

With the lack of prior knowledge in the concepts of SVA, ML and the functionality of the APB, a considerable amount of time was spent initially in the form of background study to understand these concepts. With that, a framework of simpler ML algorithms like KNN and DT were developed. A reference model of the APB was developed to mimic its behaviour using SVA and constrained random stimuli generation. The four control signals obtained from this reference model *pselect*, *pwrite*, *penable* and *pready* were considered as input class and the behavior it represented was assigned as the output class.

The dataset was split with 60% for training and 40% for testing. These models were trained without introducing the wait-state transactions which resulted in 100% accuracy. In other words, it could precisely classify the failed transactions along with read and write. Next, training datasets with transactions containing *'n'* number of wait states were included and this was the first challenge encountered during the study. As the APB could have an unknown *'n'* number of wait states, preparing the dataset to meet the requirement was practically difficult. A work around was to delete these wait states which defeats the purpose of verifying the APB as this leads to loss of important data. This drawback lead to the implementation of a neural network, the FFNN which again posed the same challenge that of KNN and DT.

The further study of neural networks, introduced us to another neural network, the LSTM. This algorithm suited the requirements to train and verify the full functionality of APB without the loss of important data. The advantage of LSTM is its capability to learn and the remember the time series data. An LSTM model was trained and tested with datasets containing various write, read and fail transactions with various wait states. Initially, datasets with fewer number of transactions were trained, which provided promising results, but as the test data set progressed quantitatively, the model started to provide interesting results. Mainly, with this progression, the LSTM model could still classify the transactions yet the number of misclassifications began to increase. With several iterations to improvise the model, the average accuracy obtained was 90%.

Given how expensive it is to fabricate a chip, SOCs are expected to be verified completely considering all its functional scenarios and corner cases. With accuracy of 90% obtained for LSTM, the gap of 10% is extremely costly. As an immediate future work, the accuracy can be improved if the model is trained with larger sets of data along with fine tuning the model parameters. A drawback of LSTM is that the framework of the model itself is complex and it is required from an engineer to have in-depth knowledge of it.

The investigation concludes this study with the belief that ML algorithms are not far from being established as a verification methodology. Although, this comes with a question of reliability which is directly dependent on the accuracy of predictions provided by the built learning model. Lastly, the study also concludes that the time taken to develop a learning model is dependent on the DUT and the type of functionalities required to be verified. In the case of APB, KNN and DT consumed lesser time to implement due to their simpler framework compared to LSTM.

6.2 Future Work

With the investigation in this thesis as foundation, there is a huge scope for machine learning to be established as new verification standard. As a future study, complex bus protocols such as the AXI and the AHB could be pre-trained with LSTM to be used as reference models to verify the DUT. This can be further extended to verify other DUTs by employing more ML algorithms based on the requirement. Apart from classification based learning, other methods such as Regression or Reinforcement based learning could be used to achieve complete coverage by constraining the stimuli.

Another scope for the future work is to implement machine learning models in Python instead of MATLAB. Python has been used widely to develop ML models and for its applications in various sectors today mainly due to its extensive support of libraries and ease of use. MATLAB is a powerful tool yet it is extremely resource hungry and expensive. An interesting study would be to select a DUT and develop ML algorithms on both the tools to perform a comparison. This study could be helpful for SOC industries to determine which tool could suit them based on their requirement.

Bibliography

[1] ARM. “AMBA 3 APB Protocol”. In: *APB documentation* (2003-2004).

[2] Lida Bai and Lan Chen. “Machine-Learning-Based Early-Stage Timing Prediction in SoC Physical Design”. In: *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, pp. 1–3.

[3] Peter Böhm and Tom Melham. “Design and Verification of On- Chip Communication Protocols”. In: (2015).

[4] Eduard Cerny et al. *SVA: The power of assertions in systemVerilog.* Springer, E-Book ISBN: 978-1-4419-6600-1, 2015.

[5] Veena S. Chakravarthi. *A Practical Approach to VLSI System on Chip (SoC) Design.* Springer, 2020.

[6] Wen Chen et al. “Challenges and trends in modern SoC design verification”. In: *IEEE Design & Test* 34.5 (2017), pp. 7–22.

[7] Doulos. *SystemVerilog Assertions.* `https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/`.

[8] Eman El Mandouh and Amr G Wassal. “Estimation of formal verification cost using regression machine learning”. In: *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT).* IEEE. 2016, pp. 121–127.

[9] Ronald A Fisher. “The use of multiple measurements in taxonomic problems”. In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[10] Harry foster. *ASIC/IC Resource Trends by Mentor Graphics.* `https://blogs.mentor.com/verificationhorizons/blog/2016/10/04/part-8-the-2016-wilson-research-group-functional-verification-study/`. 2016.

[11] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[12] Han Ke, Deng Zhongliang, and Shu Qiong. “Verification of AMBA bus model using SystemVerilog”. In: *2007 8th International Conference on Electronic Measurement and Instruments*. IEEE. 2007, pp. 1–776.

[13] MATLAB. *Cross-Validation Partition.* `https://se.mathworks.com/help/stats/cvpartition.html`. 2019.

[14] MATLAB. *Deep Learning with Time Series, Sequences and Text.* `https://se.mathworks.com/help/deeplearning/ug/long-short-term-memory-networks.html`. 2019.

[15] MATLAB. *Machine Learning with MATLAB.* `https://matlabacademy.mathworks.com/`. 2019.

[16] MATLAB. *MATLAB Documentation for Statistics and Machine Learning Toolbox.* `https://se.mathworks.com/help/stats/fitcknn.html`. 2019.

[17] MATLAB. *MATLAB Documentation for Statistics and Machine Learning Toolbox.* `https://se.mathworks.com/help/stats/fitctree.html`. 2019.

[18] MATLAB. *Training Options for deep learning neural network.* `https://se.mathworks.com/help/deeplearning/ref/trainingoptions.html`. 2019.

[19] Ashok B Mehta. “ASIC/SoC functional design verification”. In: *Publ. Springer* (2018).

[20] Tom Mitchell. *Machine Learning.* New York: McGraw Hill. ISBN 0-07-042807-7. OCLC 36417892., 1997.

[21] James Obert and Tom J.Mannos. “Semi-Supervised Learning and ASIC Path Verification”. In: *2018 First International Conference on Artificial Intelligence for Industries (AI4I)* (2019).

[22] Christopher Olah. *Understanding LSTM Networks.* `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. 2015.

[23] Priyanka Choudhury Perumalla Giridhar. “Design and Verification of AMBA AHB”. In: *2019 1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE)* (2019).

[24] R. R. Khandelwal Prince Gurha. “SystemVerilog Assertion Based Verification of AMBA-AHB”. In: *2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)* (2016).

[25] Edgar Romero et al. “Support vector machine coverage driven verification for communication cores”. In: *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2009, pp. 147–152.

[26] Sincy Ann Saji and K Sivasankaran. “Test suite for SoC interconnect verification”. In: *2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)*. IEEE. 2017, pp. 1–6.

[27] Stan Sokorac. “Optimizing random test constraints using machine learning algorithms”. In: *Design and Verification Conference (DVCon)*. 2017.

[28] Chris Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2012.

[29] systemverilog.io. *SystemVerilog Assertions*. `https://www.systemverilog.io/sva-basics`. 2018.

Printed by Tryckeriet i E-huset, Lund 2022

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-863
http://www.eit.lth.se