Advancing Design Verification: Leveraging Machine Learning for Enhanced Validation Processes

Thesis submitted in partial fulfillment of the requirements
for the award of the degree of

Master of Technology

by

Sreedevi M
MA21M025

Under the Supervision of

Dr.Sivaram Ambikasaran
Assistant Professor, IIT Madras
Dr.Sri Vallabha Deevi
Tiger Analytics

DEPARTMENT OF MATHEMATICS
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
Chennai-600036

May, 2024

CERTIFICATE

This is to certify that the thesis entitled **"Advancing Design Verification: Leveraging Machine Learning for Enhanced Validation Processes"**, submitted by **Sreedevi M** to the **Indian Institute of Technology Madras** for the award of the degree of **Master of Technology** in **Industrial Mathematics and Scientific Computing**, is a record of the original, bona fide research work carried out by her under my supervision and guidance during the academic year 2023-2024 in the Department of Mathematics, IIT Madras. The thesis has reached the standards fulfilling the requirements of the regulations related to the award of the degree. The results contained in this dissertation have not been submitted in part or in whole to any other University or Institute for the award of any degree or diploma to the best of our knowledge.

[illegible]

Prof. Dr.Sivaram Ambikasaran,
Department of Mathematics,
Indian Institute of Technology Madras.

[illegible]

Dr.Sri Vallabha Deevi,
Tiger Analytics,
Chennai, India

DECLARATION

Sreedevi M
MA21M025
Date: 10/05/2024
Place: IIT Madras

ACKNOWLEDGEMENTS

I take this opportunity to acknowledge and express my gratitude to all those who supported and guided me during the thesis work. I am extremely thankful to my supervisors Prof. Dr.Sivaram Ambikasaran, Department of Mathematics, Indian Institute of Technology Madras and Dr. Sri Vallabha Deevi, Tiger Analytics for their motivation and tireless efforts to help me gain deep knowledge of the research area and support me throughout the life cycle of my M. Tech. Thesis work. Lastly, I would like to thank the Department of Mathematics, IIT Madras, for allowing me to work in this exciting field. Their resources, facilities, and academic environment have greatly contributed to my learning and research journey.

Sreedevi M
Department of Mathematics
IIT MADRAS

Abstract

In the semiconductor industry, rigorous design verification is essential to ensure the accuracy and functionality of integrated circuits before fabrication. However, as designs become increasingly complex, traditional regression methods face challenges in achieving optimal efficiency and coverage. This paper explores integrating machine learning (ML) techniques to enhance design verification processes within the semiconductor industry. Specifically, ML algorithms and methodologies can transform key processes such as test bench generation, enhancing the efficiency and effectiveness of design verification procedures. Through a series of comprehensive experiments and analyses, we showcase the effectiveness of SeqGAN in streamlining test bench creation, enhancing regression performance, and ultimately increasing coverage. Our results underscore the potential of ML to expedite validation procedures, elevate semiconductor design quality, and pave the way for new advancements in design verification methodologies.

Contents

Abbreviations

ML	**M**achine **L**earning
RL	**R**einforcement **L**earning
DV	**D**esign **V**erification
RTL	**R**egister **T**ransfer **L**ogic
FSM	**F**inite **S**tate **M**achine
IID	**I**ndependent and **I**dentically **D**istributed
DUT	**D**esign **U**nder **T**est
BPTT	**B**ack **P**ropagation **T**hrough **T**ime
NLP	**N**atural **L**anguage **P**rocessing
GAN	**G**enerative **A**dversarial **N**etworks
SeqGAN	**Seq**uence **G**enerative **A**dversarial **N**etworks

Chapter 1

Introduction

In recent years, the convergence of artificial intelligence (AI) and scientific research has heralded a new era of innovation and discovery. In the domain of machine learning, computer programs are tasked with completing specific tasks, and their ability to improve performance over time by learning from experience defines their effectiveness. This learning process involves making judgments and forecasts based on historical data. In the broader field of artificial intelligence, machine learning encompasses automatic adaptation with minimal human intervention, while deep learning, a subset of machine learning, employs neural networks to emulate the human brain's learning process. Even though deep learning necessitates more extensive data during training, it inherently showcases adaptability to novel circumstances and possesses the capability to rectify its shortcomings [1].

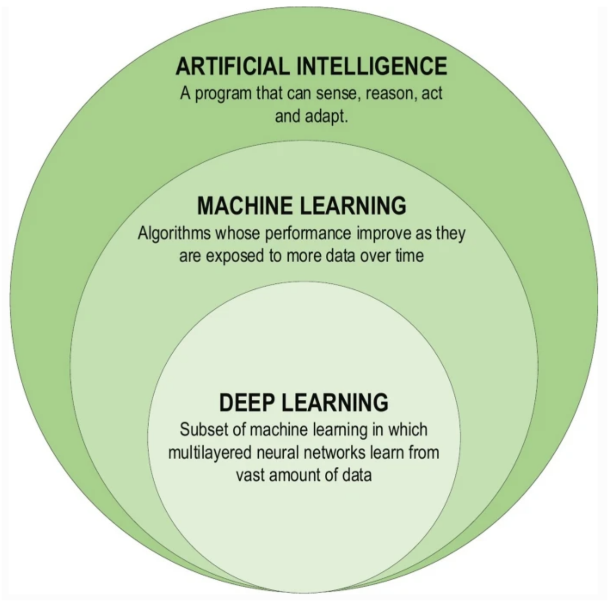

ARTIFICIAL INTELLIGENCE
A program that can sense, reason, act and adapt.
MACHINE LEARNING
Algorithms whose performance improve as they are exposed to more data over time
DEEP LEARNING
Subset of machine learning in which multilayered neural networks learn from vast amount of data

FIGURE 1.1: Artificial Intelligence (AI) and its sub fields[2]

Artificial intelligence, with its ability to accelerate experimental simulations, process massive datasets, and validate theoretical hypotheses, has emerged as an indispensable tool across a multitude of scientific fields [3].

Machine learning involves mapping input to output using a predetermined world representation (features) tailored for each task. Conversely, deep learning seeks to represent the world as a nested hierarchy of automatically detected concepts, leveraging the architecture of deep learning models to accomplish this goal. Deep learning models possess several advantages over traditional machine learning models, including a greater number of learning layers and a higher level of abstraction. The three primary machine learning approaches are Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

1.1 Machine Learning Approaches

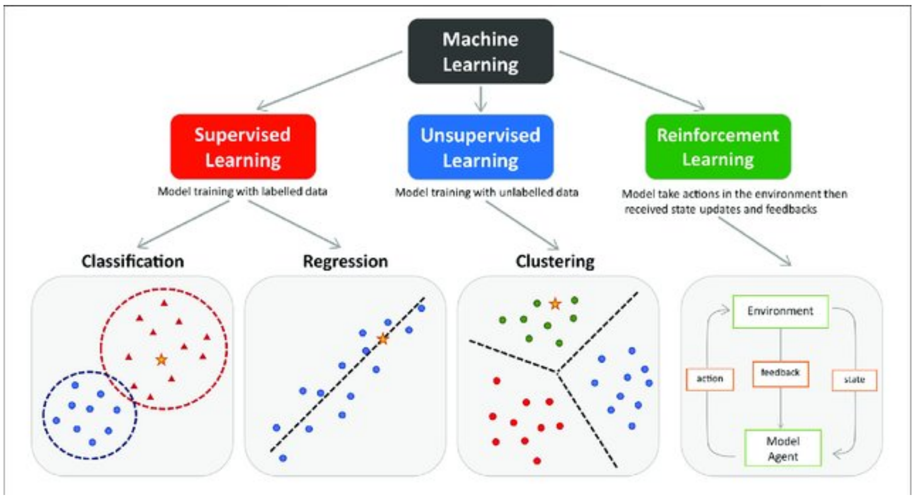


FIGURE 1.2: Main types of Machine Learning[4]

- **Supervised Learning:** Supervised Learning involves training a learning algorithm using labeled data, where each data point has input features as well as the output value. The algorithm iteratively adjusts its predictions to minimize the error between actual outputs and its predictions. This approach typically manifests in two forms: classification and regression. In classification, input data is categorized into distinct output classes, while in regression, the input data is mapped to continuous output values. The primary goal in both cases is to identify patterns and structures in the input data that facilitate accurate and effective predictions. Industries such as sales, commerce, and finance commonly employ supervised machine learning algorithms for tasks like forecasting and decision-making. Popular algorithms in supervised learning include Linear Regression, Decision Tree, Random Forest and KNN.

- **Unsupervised Learning:** Unsupervised Learning trains the learning algorithm using input data without labeled outputs. Unlike supervised learning, which relies on paired input and output data, unsupervised learning aims to uncover the underlying structure or pattern of distribution within the data. This approach is commonly used for association and clustering tasks and finds applications in digital advertising and marketing to analyze customer-centric data and personalize services. Without labeled output data, unsupervised learning identifies similarities within the dataset and groups data accordingly. For instance, consider an unsupervised learning algorithm assigned to analyze a dataset comprising images of different cats and dogs. Operating without prior training on the dataset, the algorithm independently identifies unique characteristics within the images by clustering them according to their similarities.

- **Reinforcement Learning:** Reinforcement Learning is a method where an agent learns by interacting with its environment, rather than being given direct instructions. Through this process, the agent seeks to maximize its total reward over time, receiving feedback in the form of rewards for its actions. This approach finds common application in fields such as Robotics. In this domain, robots are empowered to influence their decisions by manipulating diverse objects within their visual range.

1.1.1 DL and Generative Adversarial Networks

Deep learning(DL), rooted in the principles of artificial neural networks, serves as a fundamental component of modern machine learning methodologies. Its capacity to learn from data, both supervised and unsupervised, has propelled its widespread adoption across numerous domains, including autonomous vehicles, computer vision, natural language processing, recommendation systems, bioinformatics, and medical image analysis. In many cases, deep learning models have exhibited performance on par with, or even surpassing, human experts, highlighting their revolutionary potential in advancing scientific exploration. At the heart of this transformative shift lie advanced technologies such as deep learning and generative adversarial networks (GANs), which take inspiration from the intricate workings of the human brain.

Generative Adversarial Networks (GANs), a category of machine learning frameworks, have emerged as a powerful tool for generating diverse and high-quality data samples. Operating on the principles of adversarial training, GANs utilize two neural networks— a generator and a discriminator— locked in a zero-sum game. This innovative approach has revolutionized fields such as healthcare, finance and banking, offering comprehensive solutions for many real-world challenges [[5],[6]].

However, the evolution of GANs extends beyond traditional applications, as evidenced by recent breakthroughs in sequence generation tasks. The advent of Sequential GANs (SeqGANs) has unlocked new frontiers in creative endeavors such as poem composition, speech-language generation, and music composition [7]. By conceptualizing sequence generation as a sequential decision-making process, SeqGANs have overcome limitations inherent in discrete token-based data, paving the way for unprecedented advancements in AI-driven creativity.

This work introduces a novel approach to address the challenges associated with creating comprehensive test benches for Finite State Machines (FSMs) in the semiconductor industry. While FSM coverage is traditionally tracked using code coverage tools, this work centers on optimizing the test bench generation process to achieve maximum coverage. Minimizing runtime and enhancing efficiency reduces dependence on costly commercial tools, ultimately reducing overall expenses.

In the semiconductor industry, the creation of test benches plays a crucial role in verifying the functionality and reliability of FSM-based designs. Traditional approaches to test bench generation often depend on basic statistical techniques, which can generate a large number of test benches to achieve the desired coverage. Consequently, this leads to extended run times on commercial EDA tools for a given design[8]. Additionally, running the code through these tools repeatedly can be time-consuming, leading to inefficiencies in the verification process. To address these challenges, this work presents a method that leverages machine learning techniques, specifically the Sequential Generative Adversarial Network (SeqGAN), to automate the generation of test benches for FSMs. By utilizing SeqGAN, we aim to not only improve line coverage but also reduce the runtime required on costly commercial tools and streamline the verification process.

In the subsequent sections of this work, we provide a detailed overview of the methodology employed to train the SeqGAN model and generate test benches for FSMs. We also present experimental results demonstrating the effectiveness of our approach in achieving higher coverage compared to traditional methods. Finally, we discuss the implications of our findings for the semiconductor industry and highlight opportunities for further research in this area.

Chapter 2

Design Verification

Semiconductors serve as the foundation of modern electronics, powering devices ranging from smartphones and laptops to autonomous vehicles and IoT devices. The design verification process ensures that semiconductor designs function correctly under all possible conditions. This involves testing various aspects such as functionality, performance, power consumption, and safety, to meet the demanding requirements of today's tech-driven world.

2.1 Understanding Design Verification

Design verification is a pivotal aspect of product development, often consuming up to 80% of the total project timeline[9]. Its primary goal is to ensure that the design meets system requirements and specifications. By undergoing design verification, the quality of the design is established, thereby enhancing the project's chances of success by identifying potential errors in both the design and system architecture. The objective is to exhaustively simulate all functions while meticulously investigating any potential erroneous behavior. The design verification process encompasses several key steps, including Testbench development, Simulation, Functional Verification, Coverage Analysis, Debugging, and Iteration.

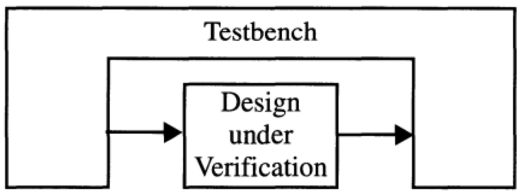


FIGURE 2.1: Figure shows how a test bench interacts with a design under verification (DUV)[10].

The term "testbench" typically refers to simulation code utilized to generate a predetermined input sequence for a design, optionally observing its response. The testbench is responsible for providing inputs to the design and monitoring its outputs. The primary challenge in verification lies in determining the appropriate input patterns to supply to the design and understanding the expected output of a correctly functioning design when subjected to those input patterns[10].

During simulation, the design is tested using the test bench to ensure it behaves as expected and produces the desired outputs for various input scenarios. Simulation tools analyze the chip's behavior and flag any issues or discrepancies.

Functional verification is a pivotal stage aimed at ensuring the correct functionality is used under various operating conditions. It involves testing the design to verify whether it behaves according to its intended specifications and functional requirements.

The final stage is debugging and iteration. If any issues or bugs are identified during verification, engineers debug and refine the design, iterating the process until it meets the desired functionality and quality standards.

Verification completeness is a critical aspect of ensuring the reliability and functionality of a design. However, achieving exhaustive verification is often impractical due to the large

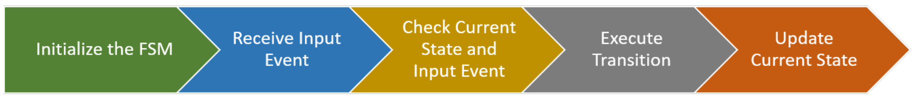
number of possible scenarios. This is where coverage metrics play a crucial role. Coverage metrics provide a systematic way to assess the thoroughness of the verification process by tracking how much of the design has been exercised by the testbench. Here, coverage analysis quantifies the percentage of the source code that is executed when a specific test suite runs. Higher test coverage indicates that more parts of the source code are tested during execution, implying a lower likelihood of undetected bugs compared to code with lower test coverage. Various metrics can be utilized to measure test coverage, including the percentage of program subroutines and statements executed during the test suite's execution[11].
Coverage analysis involves various metrics to evaluate the effectiveness of the verification process. Some of the common coverage metrics include:

- Statement Coverage: This metric measures the percentage of statements in the code that are executed during testing. It indicates how much of the code is covered by the test suite.
- Finite State Machine (FSM) Coverage: This metric evaluates the coverage of states and transitions in the FSMs within the DUT. It ensures that all states and transitions have been visited by the testbench, indicating thorough verification of the FSM behavior. Coverage of states ensures that each state in the FSM is reached during testing, while coverage of transitions ensures that all possible transitions between states are exercised.
- Path Coverage: Path coverage measures the percentage of unique paths through the code that is executed during testing. It aims to ensure that all possible execution paths in the code are tested.

The emergence of generative AI presents significant opportunities for advancing technical and business processes in high-tech and semiconductor industries. From optimizing complex system design processes to accelerating time-to-market for new products, generative AI holds unlimited potential for enhancing engineering and manufacturing methodologies.

2.2 Finite State Machines

A Finite State Machine(FSM) is a model of computation based on a hypothetical machine made of one or more states. An FSM is characterized by a set of states, an initial state, input events or stimuli, output actions or responses, and transitions between states based on input events. Only one state is active at any given time. To perform different tasks, the machine smoothly transitions from one state to another. The state changes based on the inputs it receives, which then dictate what the machine does next and what outcome it produces.
A flowchart outlining the operation of a finite state machine (FSM):



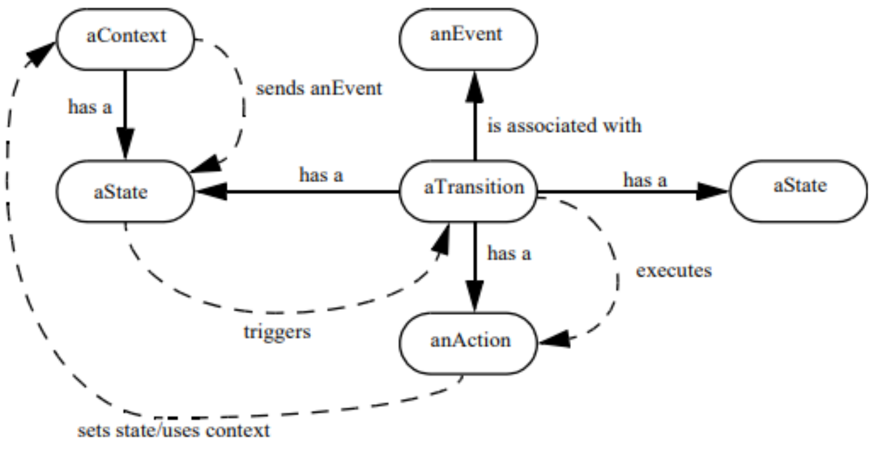


FIGURE 2.2: The FSM framework's components[12]

FSMs have a finite number of states, with specific conditions triggering these transitions, and each state is associated with its own set of behaviors or actions. An FSM generally models the behavior of the control logic. Finite state machines are used in various day-to-day applications without us even realizing it. While we may not directly interact with FSMs, they are employed in the background to control and manage various systems and processes. FSMs find application across various domains, including mathematics, artificial intelligence, gaming, linguistics, and practical systems such as vending machines, traffic lights, elevators, alarm clocks, microwaves, and cash registers. They offer a structured framework for modeling and understanding systems with finite, discrete behavior.

One of the examples of systems that can be modeled using FSMs is an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. This system operates according to FSM principles, seamlessly transitioning between states as you interact with it to ultimately unlock.

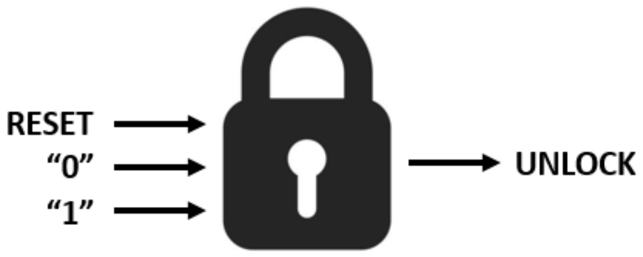


Figure 2.3: An example of FSM: Electronic combination lock

Conceptually, it can be represented as a directed graph, with nodes symbolizing states and edges denoting transitions between these states.

An example of a state diagram for a circuit is shown in Figure 2.4. Here each circle symbolizes a "state," representing a distinct condition in which our machine operates. The upper half of each circle describes that condition, aiding in our understanding of the intended function of our circuit at that state. Arrows denote "transitions" from one state to another, indicating the potential movement based on the current input. Depending on the input received, the machine may transition to a different state.

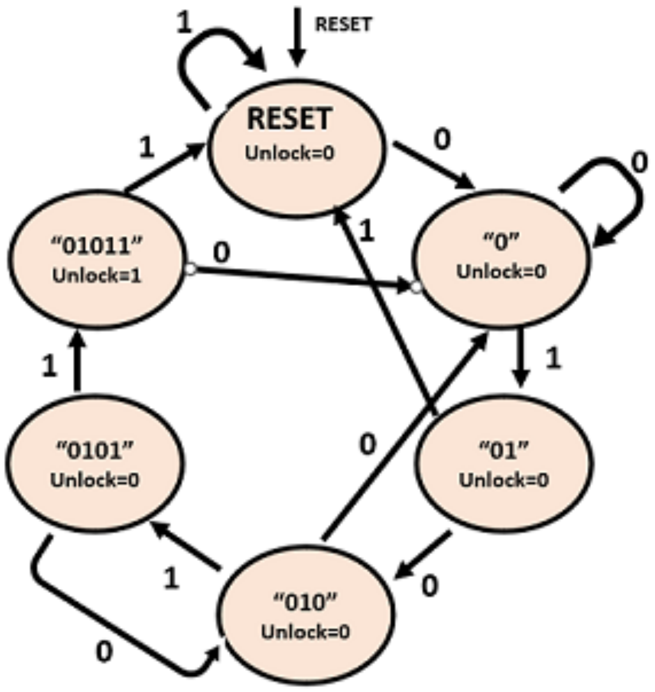


FIGURE 2.4: State transition diagram of the electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination is 01011.

2.2.1 FSM coverage

Finite State Machine (FSM) coverage is indispensable in coverage-driven functional verification, ensuring the reliability and robustness of a design. State machines, with their numerous branches and functional paths, demand thorough testing to prevent critical functionality issues or deadlock scenarios. Coverage metrics play a pivotal role in addressing these risks by ensuring that all states and transitions within the FSM undergo comprehensive verification.

A critical aspect of FSM coverage is visited state coverage, ensuring that every state in the FSM is visited during the simulation. This metric safeguards against leaving any states untested, reducing the risk of overlooking state-specific issues or hidden paths within the design.

Additionally, FSM coverage contributes to risk mitigation by ensuring that critical scenarios or edge cases are adequately addressed, thus minimizing the chances of system malfunctions or errors. It also guarantees functional correctness by verifying that the design behaves as intended under various input conditions and transitions between states correctly.

Moreover, analyzing FSM coverage metrics facilitates quality improvement by identifying areas of the design that require further testing or refinement, leading to enhanced design quality and reliability. In case of errors or unexpected behavior, FSM coverage data aids in debugging the design by identifying untested or inadequately tested states or transitions.

Our analysis predominantly focuses on FSM (Finite State Machine) state coverage.

Chapter 3

Sequence generation using Machine Learning

Sequential data, characterized by its time dependency, finds wide applicability across various business domains such as credit card transactions, medical healthcare records, and stock market prices. Examples of sequential data include time series, gene sequences, and weather data, where each data point's value is dependent on previous observations.

Traditional neural networks assume data is non-sequential, analyzing each data point in isolation. However, for sequential data, models need to capture ordering and context to make accurate predictions. Tasks like time series analysis and natural language processing require models capable of handling long-range dependencies, where distant elements influence predictions[13].

Recently, recurrent neural networks (RNNs) with long short-term memory (LSTM) cells have demonstrated exceptional performance in tasks ranging from natural language and handwriting generation to sequential data generation[14]. Additionally, Generative Adversarial Networks (GANs) have shown promise in generating realistic synthetic data, including sequential data[15].

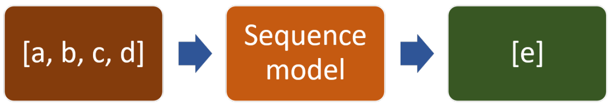


Figure 3.1: Example of a Sequence Prediction Problem

A notable advancement in sequence generation methods is SeqGAN, which effectively trains generative adversarial networks for structured sequence generation via policy gradient. In real-world scenarios such as poems, speech-language, and music generation, SeqGAN has demonstrated excellent performance in generating creative sequences. Additionally, experiments have been conducted to investigate the robustness and stability of training SeqGAN, further enhancing its applicability and effectiveness in various domains[7].

Applications of these models include:

- Time Series Forecasting: Predicting future trends in stock prices, weather patterns, or other time-dependent data.

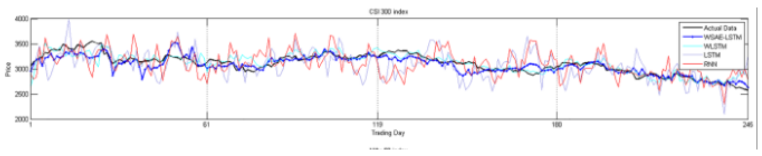


Figure 3.2: Sequential model for Time series[16]

- Natural Language Processing: Extracting semantic meaning from word sequences in tasks like sentiment analysis or machine translation.

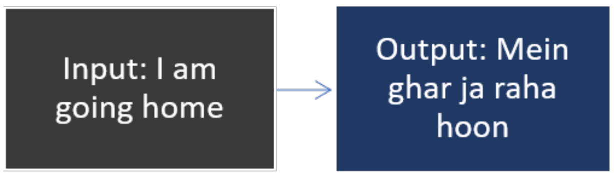


FIGURE 3.3: Machine translation

- DNA Analysis: Classifying gene sequences, predicting gene functions, or identifying mutations.
- Music Generation: Generating new musical compositions based on learned patterns and styles from existing music datasets.

3.1 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a specific type of neural network design that is very useful for finding patterns in sequences of data. They are used for all sorts of sequences like handwriting, genomes, text, and numerical time series[17]. Unlike traditional multi-layer perceptrons, RNNs incorporate connections among hidden units with time delays, facilitating the retention of information about past inputs. This capability enables them to capture temporal correlations between distant events in the data, making them well-suited for modeling sequences in diverse domains.

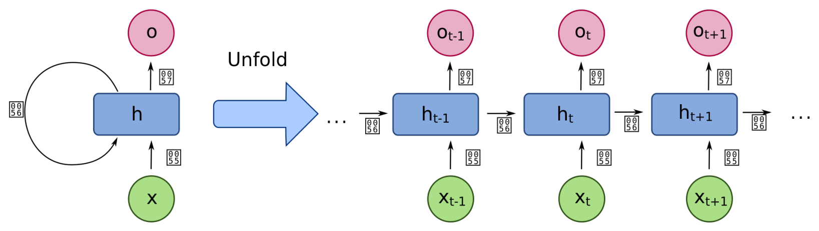


FIGURE 3.4: Compressed and unfolded basic recurrent neural network [18]

This characteristic makes them suitable for processing sequential or time-series data, addressing challenges such as handling variable-length sequences, preserving sequence order, capturing long-term dependencies, and sharing parameters across the sequence.

Overall, RNNs serve as a versatile architecture for detecting patterns in sequential data, encompassing text, time series, and even segmented image patches. Assuming the predictions are probabilistic, a trained network can generate novel sequences by iterative sampling from its output distribution and using each sample as input for the next step. Although the network operates deterministically, the introduction of randomness through sample selection creates a distribution over sequences. This distribution is conditional, as it relies on the network's internal state, which is influenced by previous inputs, thus shaping its predictive distribution [19]. One limitation of standard RNNs is their struggle to retain information about past inputs over long sequences, commonly called the "vanishing gradients" problem. This issue arises due to the nature of backpropagation through time (BPTT) during training, where gradients tend to diminish exponentially as they propagate back through time[20]. As a result, the influence of distant past inputs on the current prediction diminishes rapidly, hindering the model's ability to capture long-range dependencies.

This limitation impacts the RNN's effectiveness in modeling sequences with significant temporal dependencies, as it may fail to consider distant past inputs when making predictions adequately. Consequently, the model's performance may degrade, particularly when tasked with generating or predicting sequences that exhibit long-term dependencies.

3.2 Long Short-Term Memory

Long Short-Term Memory Networks (LSTMs) have emerged as a powerful variant of recurrent neural networks (RNNs), specifically designed to tackle the challenge of learning long-term dependencies. Originally introduced by Hochreiter & Schmidhuber in 1997 [21], LSTMs have been refined and widely adopted due to their effectiveness across diverse problem domains.

LSTMs are engineered to address the long-term dependency problem inherent in traditional RNNs. Unlike standard RNNs, which struggle to retain information over extended periods, LSTMs inherently excel at preserving information for prolonged durations, making them well-suited for capturing complex temporal relationships.

These specialized networks employ Long Short-Term Memory Units (LSTMs), which are meticulously crafted to mitigate the issue of vanishing gradients, allowing RNNs to learn over significantly more time steps, often exceeding 1000. Achieving this feat involves storing additional information outside the conventional flow of neural networks, accomplished through structures called gated cells.

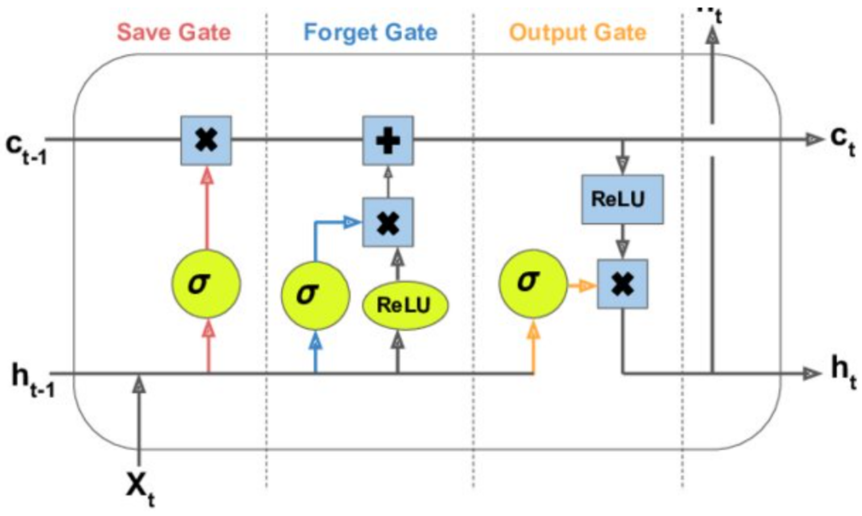


FIGURE 3.5: Single cell of the LSTM [22]

Comprising three crucial components—input gate, forget gate, and output gate—LSTMs meticulously manage the flow of information within each LSTM cell. Utilizing sigmoid functions, these gates regulate the input, retention, and output of information, thereby facilitating the selective storage and retrieval of data crucial for capturing long-term dependencies.

In contrast to simpler architectures like feed-forward neural networks, LSTM networks are computationally more demanding. This increased computational complexity can impose constraints on their scalability, especially when dealing with large-scale datasets or resource-constrained environments.

Moreover, training LSTM networks tends to be more time-consuming compared to simpler models. The computational intricacies involved in LSTM architectures require more data and longer training times to achieve optimal performance, which can be a significant consideration in time-sensitive applications or when computational resources are limited.

In summary, LSTMs represent a powerful tool for processing sequential data, offering solutions to the challenges of learning long-term dependencies and retaining information over extended time intervals. Their versatility and effectiveness make them indispensable across a wide range of applications.

Applications of RNNs and LSTMs: Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs) have a wide range of applications across different domains due to their ability to model sequential data and capture temporal dependencies.

- Language Modeling and Text Generation: RNNs and LSTMs are widely used for language modeling tasks, such as predicting the next word in a sentence or generating coherent text. They have applications in machine translation, chatbots, and natural language understanding.
- Speech Recognition: RNNs and LSTMs are employed in speech recognition systems to convert spoken language into text. They can model the temporal structure of speech signals and effectively recognize phonetic patterns.
- Time Series Forecasting: RNNs and LSTMs are utilized to predict future values in time series data, such as stock prices, weather patterns, or energy consumption. They can capture complex temporal patterns and make accurate predictions based on historical data.
- Music Generation: RNNs and LSTMs can learn the structure of musical compositions and generate new music sequences. They have applications in automatic composition, melody generation, and music recommendation systems.

3.3 Generative Adversarial Network

Generative Adversarial Networks (GANs) stand as a groundbreaking method within deep learning for generating data, whether it be images, text, or other forms of structured data.

Comprising a Generator and a Discriminator, GANs are adept at creating new examples that closely resemble those found in a given dataset. The three main components of GAN can be given as[23]:

Generative: The generative aspect of GANs involves learning a model that describes the process of data generation using probabilistic methods. This model aims to understand and replicate the underlying patterns and distributions present in the training data.

Adversarial: In GANs, the adversarial component entails setting up a competition between two neural networks: the generator and the discriminator. The generator produces synthetic data samples, which are then evaluated by the discriminator to distinguish between real and fake instances. This adversarial process encourages the generator to produce increasingly realistic outputs by learning from the feedback provided by the discriminator.

Networks: GANs utilize deep neural networks as their fundamental architecture for both the generator and the discriminator. These networks leverage artificial intelligence algorithms to learn and improve over time through training. The generator model, in particular, is responsible for generating synthetic data samples that closely resemble real data, contributing to the overall effectiveness of the GAN framework.

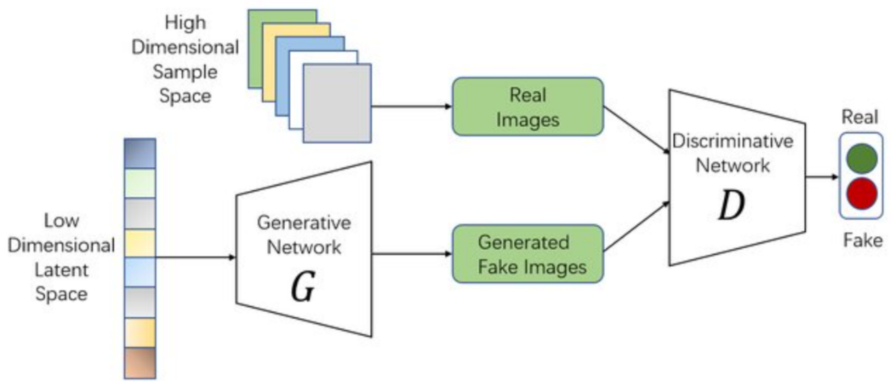


Figure 3.6: Architecture of GAN [24]

The crux of GANs lies in the adversarial relationship between the Generator and the Discriminator. While the Generator aims to produce data that is indistinguishable from real examples, the Discriminator endeavors to accurately classify between genuine and synthetic data. Through iterative learning, it becomes proficient at discerning real from fake data, penalizing the Generator for producing outputs that are deemed implausible. As training progresses, the Discriminator's ability to differentiate between real and synthetic data drives the Generator to enhance the quality of its outputs. This adversarial training dynamic propels both networks towards refinement, ultimately enabling the Generator to generate high-quality, realistic samples that closely emulate those in the original dataset.

By leveraging architectures suited to the specific data domain, GANs have demonstrated their versatility in various applications, ranging from image generation to text synthesis and beyond. Their ability to autonomously identify and replicate patterns in data has made them a powerful tool in the realm of generative modeling within deep learning.GANs have found extensive applications in various fields, including image synthesis, style transfer, text-to-image synthesis, and more. Their versatility and ability to generate realistic data have revolutionized generative modeling.

Generative Adversarial Networks (GANs) to generate sequences, such as text data, present unique challenges compared to generating continuous data like images. GANs are originally designed to generate continuous data, making them less suitable for generating sequences of discrete tokens, such as text. Text generation involves predicting the next token in a sequence based on the previous tokens, which is inherently different from generating continuous data. This discreteness introduces complexities in training GANs for text-generation tasks. Also, GANs typically provide a score or loss for an entire sequence only after it has been fully generated. This poses a challenge when generating sequences incrementally because it's difficult to assess the quality of a partially generated sequence and predict its future score once completed. As a result, determining the quality of intermediate sequences during generation becomes non-trivial.

3.3.1 Sequence GAN

SeqGAN is an extension of Generative Adversarial Networks (GANs) specifically designed for sequence generation tasks, such as text generation. It addresses the challenges of discrete data generation and scoring partial sequences by framing sequence generation as a reinforcement learning (RL) problem and utilizing policy gradient techniques.

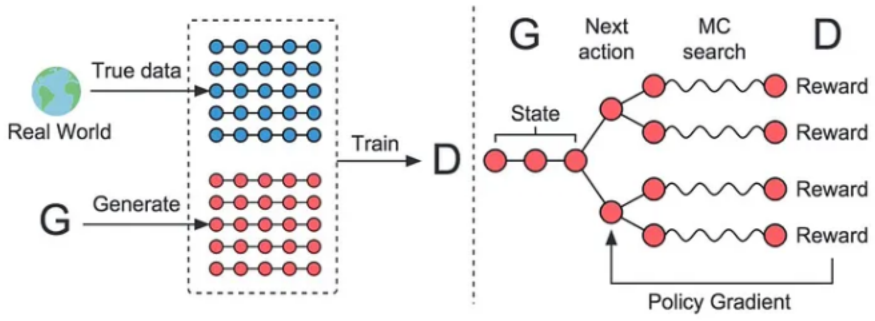


FIGURE 3.7: Architecture of SeqGAN [7]

In SeqGAN, the generator is treated as a Reinforcement Learning agent, where the generation of each token in a sequence corresponds to taking an action. The state of the environment represents the sequence generated so far, and the action is selecting the next token in the sequence.

SeqGAN employs policy gradient optimization to train the generator. Policy gradient methods directly optimize the parameters of the generator's policy based on the expected return (cumulative reward) received from the environment. The objective is to maximize the expected future reward, encouraging the generator to learn a policy that generates high-quality sequences[7]. To estimate the expected return, SeqGAN utilizes a Monte Carlo search approach. After generating a complete sequence, the discriminator is used to evaluate the sequence's quality. The reward signal provided by the discriminator serves as an estimate of the sequence's quality, guiding the generator's learning process. The discriminator is trained separately using standard GAN techniques, providing feedback to the generator by evaluating the quality of generated sequences. SeqGAN balances exploration (trying different actions to discover optimal policies) and exploitation (selecting actions based on current knowledge to maximize reward) during training. The generator in SeqGAN can be implemented using recurrent neural networks (RNNs) such as Long Short-Term Memory (LSTM). These architectures are well-suited for handling sequential data and allow the generator to capture long-range dependencies in the generated sequences.

Overall, SeqGAN with policy gradient combines ideas from reinforcement learning and adversarial training to train a generator model for sequence generation tasks. By optimizing the generator's policy directly using policy gradient methods and leveraging discriminator feedback for reward estimation, SeqGAN effectively addresses the challenges associated with generating discrete sequences like text.

Chapter 4

Enhancing Design Verification

The semiconductor industry relies on rigorous design verification to ensure the accuracy and functionality of integrated circuits prior to fabrication. As designs become more complex, traditional regression methods face challenges in achieving optimal efficiency and coverage. However, the integration of machine learning (ML) presents new opportunities to enhance the verification process in semiconductor design.

By leveraging ML algorithms and techniques, semiconductor designers can transform key aspects of design verification. ML-powered approaches offer enhancements in test vector generation, enabling more thorough and effective testing procedures. These advancements promise to accelerate the validation process and raise the quality of semiconductor designs.

4.1 Leveraging ML for enhanced DV

Understanding how ML can enhance design verification involves exploring the verification flow. Initially, the architecture team constructs a virtual model to evaluate system performance. Subsequently, they develop the RTL model and perform linting to identify coding errors. Static verification ensues to pinpoint structural flaws in the design, followed by formal verification, which offers a more profound analysis, validating critical properties[25]. Concurrently, a test bench is established, and simulations, potentially including emulations, are executed to fulfill verification objectives. Simulation results undergo debugging, with regressions rerun until verification coverage targets are attained.

Simulation serves as a critical step in design verification, uncovering approximately 65% of all bugs[25]. ML emerges as a potent tool to enhance performance in various aspects. Setting up simulation and regression run parameters can be time-intensive and necessitate specialized expertise. As code evolves and regression runs increase, adjustments to settings become imperative for peak performance. ML can address this challenge by automatically generating test benches that offer superior coverage. This method reduces the manual effort needed to achieve coverage closure and improves regression performance.

Test bench generation is a pivotal aspect of design verification, directly impacting coverage and validation effectiveness. ML offers a promising avenue to enhance test bench generation by analyzing the design's behavior and identifying critical paths or potential failure points. ML algorithms like Generative Adversarial Networks (GANs) can utilize past simulation outcomes to guide the creation of test benches focusing on specific design aspects or corner cases. This ML-driven approach enriches overall test coverage, enabling the detection of subtle design issues that conventional methods may overlook.

4.2 Problem Definition

In the semiconductor industry, as systems become more complex, there's a growing need for effective methods to find and fix potential issues efficiently. Despite the various steps involved in verification, like virtual modeling, simulation, and test bench creation, challenges persist in achieving the best performance and ensuring thorough coverage. These challenges include the time it takes to set up simulations, the need for quick regression runs, and the manual work needed for test bench creation. Additionally, ensuring full coverage is both crucial and difficult. This study aims to explore how machine learning (ML) techniques can help overcome these challenges and improve design verification. Specifically, we'll look into automating test bench creation, optimizing simulation settings, and enhancing coverage. By using ML algorithms like Generative Adversarial Networks (GANs), we hope to improve overall test coverage, making design verification more effective and efficient.

4.3 Dataset for Model Training

Datasets are essential in research, forming the basis for analysis and experimentation. They provide the information needed to conduct investigations, gain insights, and test hypotheses. Without reliable datasets, it would be challenging to explore complex topics, develop theories, and confirm findings. In this study, the dataset is crucial as it forms the basis for our analysis and experiments, allowing us to explore the topic and draw meaningful conclusions. Recognizing the hurdles in getting real data, we opted to generate our own synthetic data instead. Given the unavailability of FSM model state data online and that companies couldn't share theirs due to privacy reasons, synthetic data offered a practical solution. By creating synthetic data that mimicked the characteristics of real FSM model states, we were able to proceed with our research despite the limitations posed by the unavailability of authentic datasets. This approach allowed us to explore the intended research questions and draw meaningful conclusions while circumventing the obstacles associated with accessing real data.

In designing the synthetic data, we employed various distribution types to ensure that it closely resembled real-world scenarios. These distribution types included lognormal, normal, uniform, and bimodal distributions. By incorporating a diverse range of distribution types, we aimed to capture the variability and complexity present in real FSM model states. This comprehensive approach enabled us to generate synthetic data that accurately reflected the behavior and patterns observed in actual datasets, thereby facilitating meaningful analysis and experimentation.

The length of FSM model states remained fixed at 5, reflecting the typical complexity observed in real-world systems and ensuring that the synthetic data accurately represented the behavior of actual systems. Our synthetic dataset consisted of a total of 100,000 possible states, with 10,000 of these being feasible states. For distribution parameters, we utilized various distributions such as log-normal, normal, uniform, and bimodal distribution. Specifically, for the bimodal distribution, we considered different scenarios with varying numbers of feasible states, including 2,000, 10,000, and 40,000 feasible states. This approach allowed us to capture the diverse range of potential scenarios and ensure that our synthetic data accurately reflected the complexity of real-world FSM model states.

4.3.1 Dataset Distribution Analysis

Synthetic datasets encompass various distribution types to emulate real-world scenarios. These included:

- **Normal Distribution:** A normal distribution, also known as a Gaussian distribution, exhibits a symmetric bell-shaped curve, representing the probability distribution of values for a random variable. In this distribution, values are equally distributed on both sides of the central tendency, creating a balanced pattern. The empirical rule, often applied to normal distributions, dictates that approximately 68% of values fall within one standard deviation of the mean, 95% within two standard deviations, and 99.7% within three standard deviations. The mean and the standard deviation are pivotal in defining the shape and spread of the distribution.

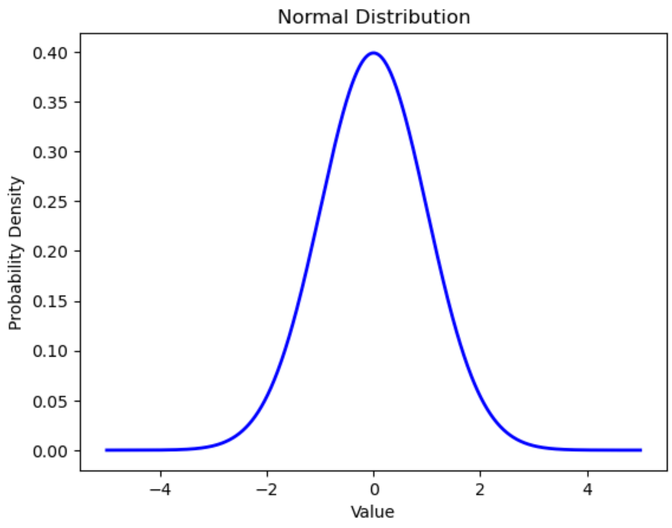


FIGURE 4.1: Normal distribution

In our dataset, we modeled the distribution of feasible states to follow a normal distribution. Specifically, we considered 10,000 feasible states, ensuring that the majority of values clustered around the mean, creating a symmetric pattern. This approach enabled us to simulate realistic scenarios and explore the characteristics of the dataset effectively.

- **Log-normal Distribution:** The Log-normal distribution is characterized by a left-skewed continuous probability distribution featuring a long tail towards the right side. It is derived from a normal distribution using logarithmic mathematics. Specifically, when the logarithm of a random variable follows a normal distribution, the continuous probability distribution of that variable is referred to as a lognormal distribution. Notably, random variables following a lognormal distribution only take positive real values.

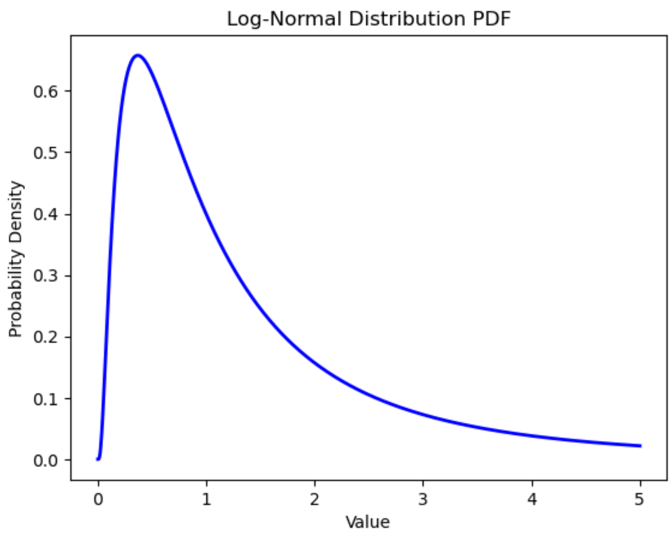


FIGURE 4.2: Log-normal distribution

In our dataset, we modeled the distribution of feasible states to follow a lognormal distribution, with 10,000 feasible states considered. By incorporating the lognormal distribution for our dataset, we aimed to accurately reflect its skewed nature, which typically exhibits a greater concentration of values towards the lower end and a long tail extending towards higher values.

- **Uniform Distribution:** In statistics, a uniform distribution refers to a type of probability distribution where all outcomes are equally likely. In a discrete uniform distribution, outcomes are discrete and have the same probability. This means that each outcome has an equal chance of occurring. The discrete uniform distribution is characterized by its symmetry, where a finite number of values are equally likely to be observed. Specifically, in a discrete uniform distribution with n values, each value has a probability of $\frac{1}{n}$.

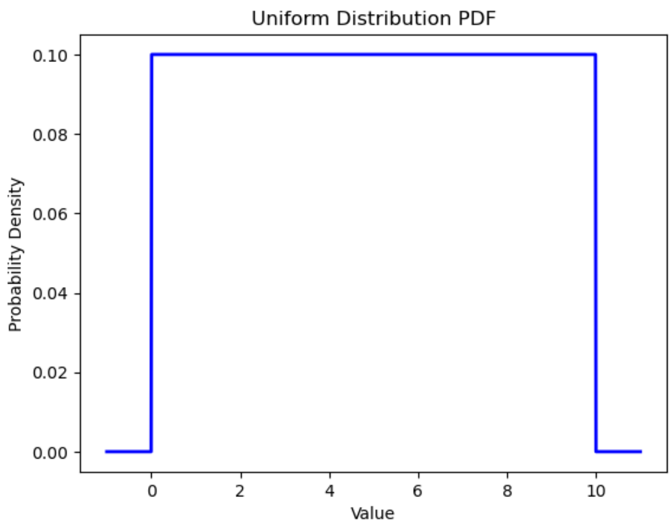


FIGURE 4.3: Uniform distribution

In our dataset, we modeled the distribution of feasible states to follow a uniform distribution, with 10,000 feasible states considered. This means that each of the 10,000 feasible states has an equal probability of occurring, illustrating the characteristics of a discrete uniform distribution.

- **Bimodal Distribution:** A bimodal distribution is defined by its two distinct peaks, representing the modes of the distribution. In continuous probability distributions, these peaks indicate the most frequent values observed in the dataset. Visually, a bimodal distribution displays two separate peaks on a graph. When the peaks of a bimodal distribution have varying heights, the taller peak is labeled as the major mode, while the shorter peak is known as the minor mode.

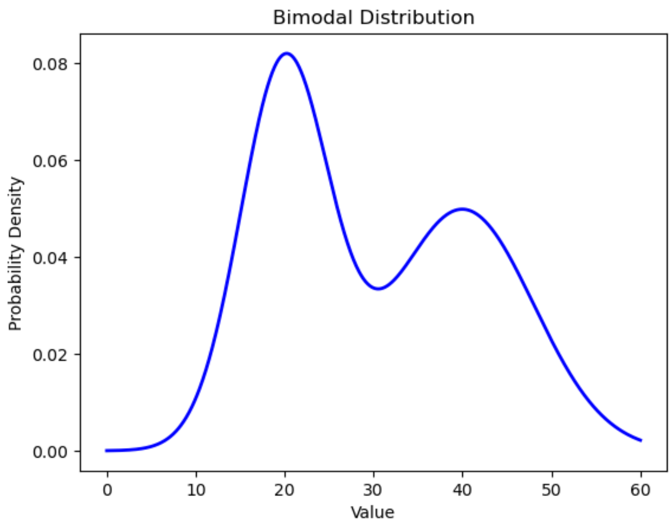


FIGURE 4.4: Bimodal distribution

In our dataset, we aimed to replicate a bimodal distribution by modeling the distribution of feasible states. We considered scenarios with 2000, 10,000, and 40,000 feasible states to observe how the distribution varied with different numbers of states. This allowed us to explore the characteristics of bimodal distributions and their implications in our analysis.

4.4 Methodology

In this section, we detail the methodology utilized to fulfill the objectives outlined in the thesis. To initiate the experimental phase, a variety of synthetic datasets were selected, each representing distinct distributions. These datasets were treated as black boxes, representing the different states within the model under investigation. The test case generation process began with the random generation of test cases across varying sizes. Subsequently, a validation step was implemented to ensure the uniqueness of these randomly generated test cases. Each test case was scrutinized against the synthetic datasets to determine its existence. Test cases identified within the synthetic datasets were then selected to form the initial training data for our SeqGAN model. This meticulous approach ensured that the test cases used for training accurately reflected the behavior of the system under examination.

Through a training process, the SeqGAN model learned to generate sequences of test cases by discerning the underlying patterns and dependencies within the dataset. Consequently, the model could create synthetic test case sequences closely resembling the behavior observed in the original dataset. Within the SeqGAN framework, several scenarios were considered, such as iterating SeqGAN with the same initial test bench, SeqGAN iterations by adding the results from previous iterations to the initial test bench, and SeqGAN iterations where, after a certain number of iterations, a new initial test bench was introduced. These scenarios provided insights into the performance of the SeqGAN model under different conditions and allowed for the identification of optimal strategies for generating test cases. The effectiveness of the SeqGAN model in generating test sequences was evaluated by comparing the generated sequences against those in the synthetic dataset. This evaluation process provided valuable insights into the model's ability to produce relevant test sequences comprehensively covering the functionality of the system under examination.

The performance metric is feasible state coverage percentage, calculated for the unique test cases generated by the SeqGAN model and compared with those created using random generation methods. This comparison facilitated an assessment of which approach yielded superior results.

Chapter 5

Analysis and Results

Our investigation aims to compare the performance of test cases generated by random methods, SeqGAN models, and retrained SeqGAN models using concatenated test cases from previous iterations. We focused on finite state machine (FSM) state coverage, exploring scenarios where feasible FSM states follow different distributions such as bimodal, normal, uniform, and log-normal. By considering different distributions for FSM states, we gained insights into the unique patterns of state occurrence associated with each distribution.

Our approach consists of four different methods for generating test cases:

- **Random Generation Method:** Test cases are randomly generated without following a specific pattern or algorithm. This method serves as a baseline for comparison against more sophisticated techniques.

- **SeqGAN Running Independently:** The SeqGAN model generates test cases iteratively without retraining after each iteration. It is trained one time from the distribution of input data to improve coverage.

- **SeqGAN Retrained Method:** After each iteration, the SeqGAN model is retrained using the concatenated test bench from the current iteration and the previous data. This approach aims to enhance coverage by leveraging both the current and previously generated test cases that are feasible.

- **SeqGAN Renewal Method:** The SeqGAN model is trained with a new test bench created using the random generation method after a finite number of iterations. This method introduces fresh data periodically to adapt the model to changes in the dataset distribution, potentially improving coverage and effectiveness.

We explored various approaches to test case generation, each with its strengths and limitations. By comparing the results obtained from these methods, we assessed the effectiveness and efficiency of each approach in generating test cases for FSMs. This comparison offered valuable insights into the performance and scalability of different test case generation strategies, providing guidance for decision-making processes in testing and verification tasks.

5.1 Results and Interpretation

5.1.1 Bimodal Distribution

A bimodal distribution is characterized by two distinct peaks or modes, indicating that states are concentrated around two central values or ranges. This distribution may arise due to specific characteristics or patterns in the system being modeled, such as alternating modes of operation or distinct phases of behavior. The bimodal distribution can significantly impact the performance of the SeqGAN model in generating test cases. Since SeqGAN relies on learning from the distribution of input data to generate new samples, the presence of a bimodal distribution introduces complexity and challenges. The model must accurately capture the dual peaks or modes of the distribution to effectively generate test cases that cover the full spectrum of state transitions within the FSM.

We have investigated three different scenarios, each characterized by a different number of feasible states within the finite state machine (FSM). These scenarios encompass a variety of conditions and complexities within the FSM, allowing us to explore the performance of SeqGAN across a spectrum of system configurations. These scenarios include:

- Case 1: 2000 feasible states out of 100,000 possible states. i.e., feasible states are 2% of the total possible states.
- Case 2: 10000 feasible states out of 100,000 possible states. i.e., feasible states are 10% of the total possible states.
- Case 3: 40000 feasible states out of 100,000 possible states. i.e., feasible states are 40% of the total possible states.

For each scenario, we examined how adjusting the size of the test bench provided to the SeqGAN model influenced the results. Test benches of varying sizes were considered, each containing unique feasible states: 100, 200, 300, and 500. These test benches were generated using the random generation method and acted as the initial training dataset for the SeqGAN model. With this test bench as training data, SeqGAN was tasked with generating sequences. To compare SeqGAN's performance with random generation, sequences equal in number to the size of the initial test bench generated by SeqGAN were compared with an equal number of test cases created randomly.

5.1.1.1 2% feasible states

The test suite sizes considered include:

1. Test bench containing 100 unique feasible states

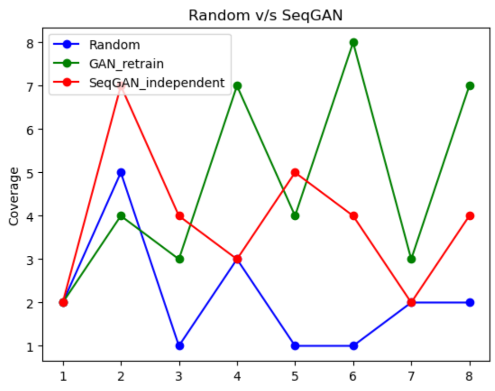


FIGURE 5.1: State Coverage Across Iterations

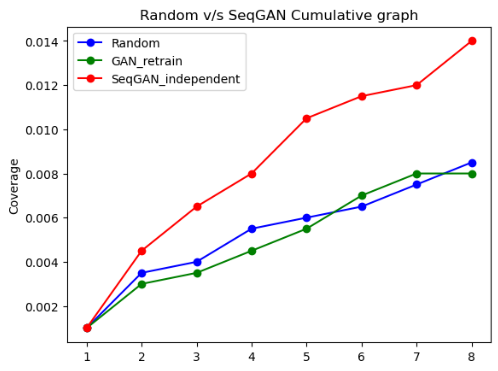


FIGURE 5.2: Cumulative State Coverage Over Iterations

- When utilizing a test bench size of 100, after eight iterations, SeqGAN independently shows an enhancement of 1.4% in coverage, while random generation methods demonstrate a lesser increase of 0.85%. Similarly, SeqGAN, when retrained after each iteration, only yields a modest 0.8% improvement in coverage.
- In comparison, SeqGAN outperforms random generation methods by achieving a 0.5% higher coverage increase after the same number of iterations.
- Despite this slight advantage of SeqGAN over random generation methods, the practice of retraining SeqGAN after every iteration does not exhibit a significant advantage. This is evident from the similar coverage increase observed between SeqGAN retraining and random generation methods. Thus, it suggests that this retraining strategy may not offer substantial benefits in this scenario.

2. Test bench containing 200 unique feasible states

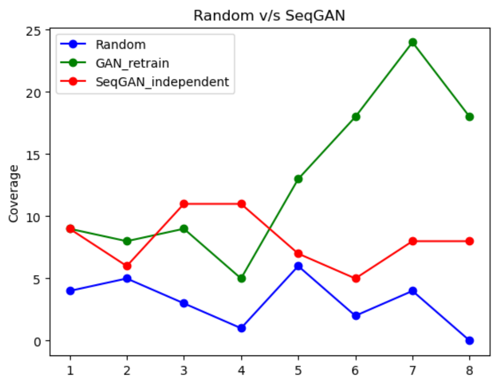


FIGURE 5.3: State Coverage Across Iterations

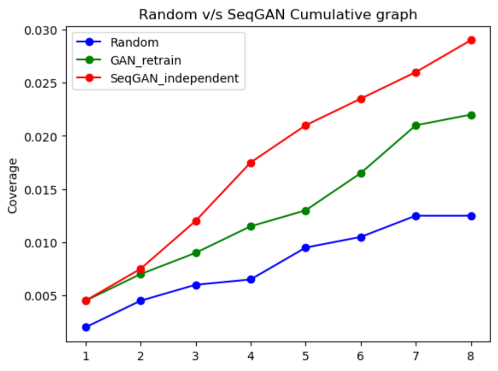


FIGURE 5.4: Cumulative State Coverage Over Iterations

- When the test bench size is 200, after eight iterations of running SeqGAN independently, there is a notable increase of 2.9% in coverage. Conversely, utilizing random generation methods results in a lower increase of 1.25% in coverage over the same period. Furthermore, when SeqGAN is retrained after every iteration, there is an increase of 2.2% in coverage after eight iterations.
- Comparing random test case generation with SeqGAN, there is a substantial 1.65% increase in coverage observed in SeqGAN, indicating its superior performance in terms of coverage improvement after eight iterations. Additionally, it's noteworthy that SeqGAN, when run independently without retraining after each iteration, consistently yields better results compared to SeqGAN with retraining, even though SeqGAN retrain performs better than random test generation.

3. Test bench containing 300 unique feasible states.

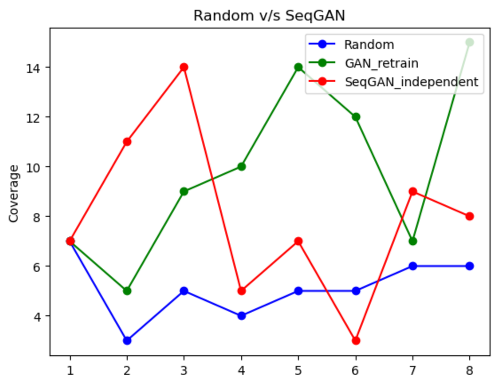


FIGURE 5.5: State Coverage Across Iterations

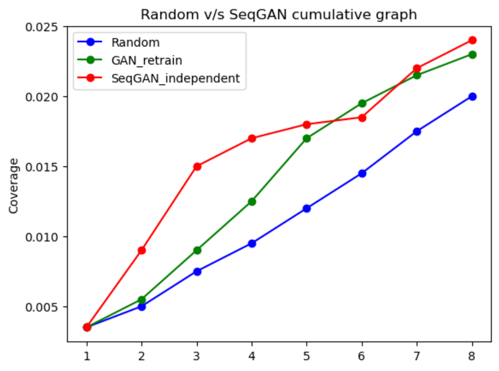


FIGURE 5.6: Cumulative State Coverage Over Iterations

- When the test bench size is 300, after eight iterations of running SeqGAN independently, there is an increase of 2.4% in coverage. In contrast, using random generation methods results in a slightly lower increase of 2.0% in coverage over the same period. Additionally, when SeqGAN is retrained after every iteration, there is an increase of 2.3% in coverage after eight iterations.
- Comparing random test case generation with SeqGAN, there is a modest 0.4% increase in coverage observed in SeqGAN after eight iterations. While SeqGAN retraining after every iteration performs better than the random scenario, it is evident that SeqGAN performs better than both methods in this comparison.

4. Test bench containing 500 unique feasible states.

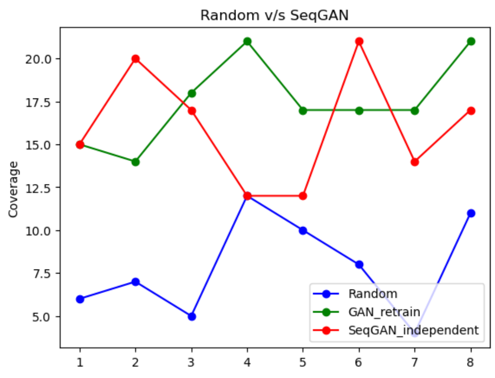


FIGURE 5.7: State Coverage Across Iterations

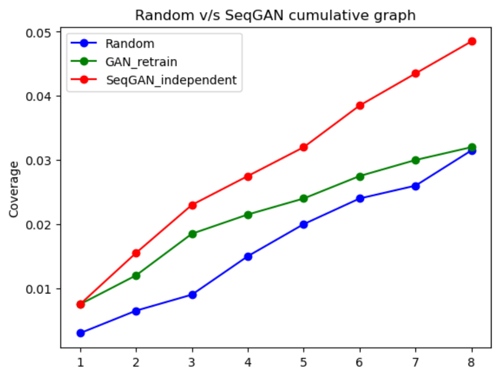
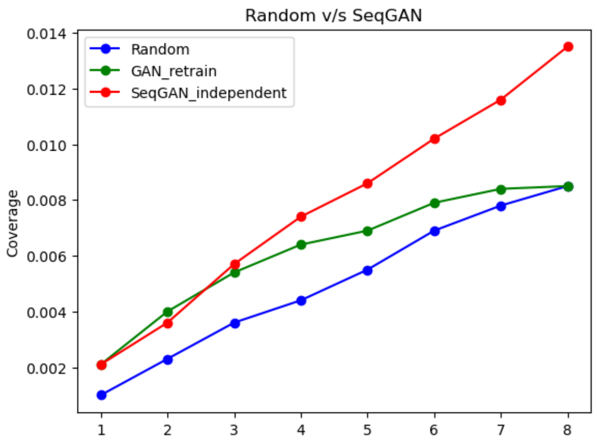


FIGURE 5.8: Cumulative State Coverage Over Iterations

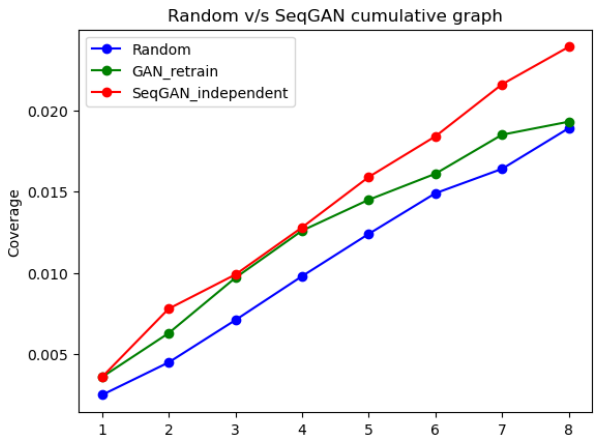
- When the test bench size is 500, after eight iterations of running SeqGAN independently, there is a significant increase of 4.85% in coverage. In contrast, using random generation methods results in a lower increase of 3.15% in coverage over the same period. Additionally, when SeqGAN is retrained after every iteration, there is an increase of 3.2% in coverage after eight iterations.
- Comparing random test case generation with SeqGAN, there is a substantial 1.7% increase in coverage observed in SeqGAN after eight iterations, indicating its superior performance in terms of coverage improvement. While SeqGAN retraining after every iteration is slightly better than random generation methods, it is evident that SeqGAN trained independently outperforms both methods in this comparison.

5.1.1.2 10% feasible states

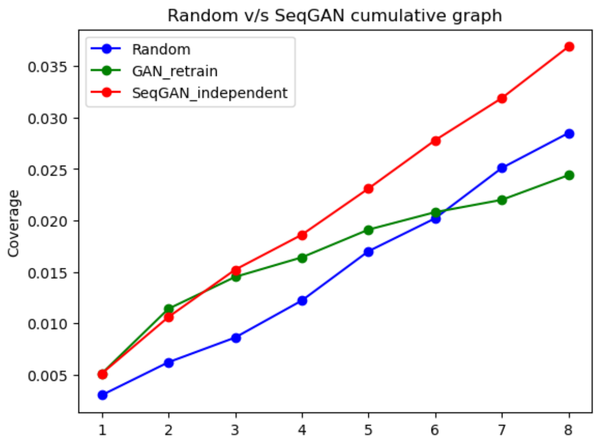
The figures illustrate the cumulative coverage analysis using test benches of varying sizes when 10% of states are feasible.



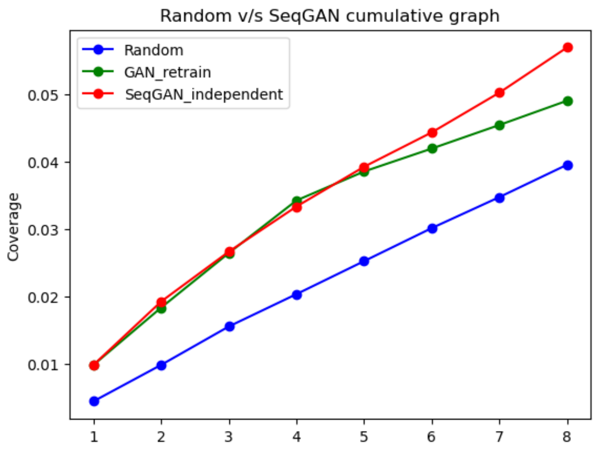
(a) Test bench containing 100 unique feasible states



(b) Test bench containing 200 unique feasible states



(c) Test bench containing 300 unique feasible states



(d) Test bench containing 500 unique feasible states

FIGURE 5.9: Cumulative State Coverage Over Iterations

The table below illustrates the percentage of state coverage after 8 iterations, corresponding to the data in figure 5.9.

% of states covered			
Initial test bench size	Random method(%)	SeqGAN(%)	SeqGAN Re-train(%)
100	0.85	1.35	0.85
200	1.89	2.39	1.96
300	2.44	2.85	3.69
500	3.90	5.69	4.90

Table 5.1: Percentage of states covered after 8 iterations when 10% of states are feasible

In all scenarios, we observed that SeqGAN, when trained using the initial test bench, consistently outperformed the other two methods. SeqGAN, with retraining, also showed superior performance compared to random test case generation. Furthermore, increasing the test bench size resulted in SeqGAN achieving higher coverage rates. Notably, when the test bench size was 100 or 200, both retraining and random generation methods exhibited similar performance after 8 iterations. However, upon increasing the test bench size to 300, SeqGAN retrain began to show noticeable improvement compared to random test generation. Additionally, a comparison between random test case generation and SeqGAN after 8 iterations revealed an increase of 0.8% coverage of states in SeqGAN when the test bench size was 300 or 500.

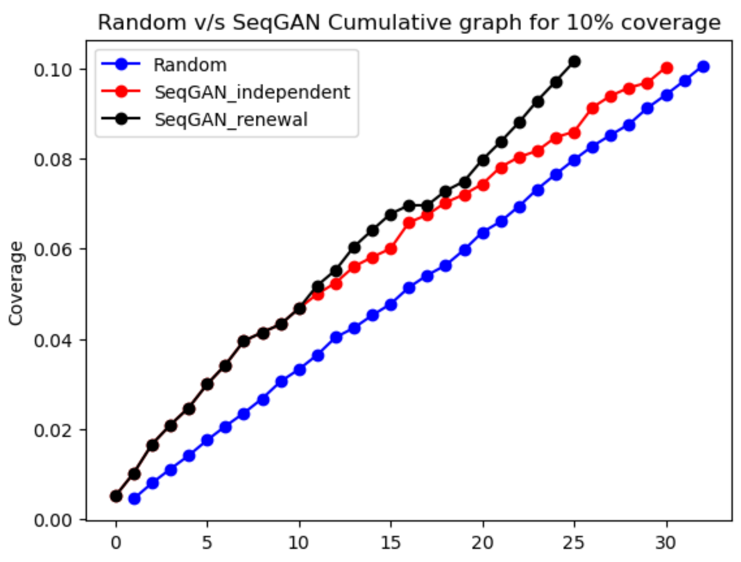
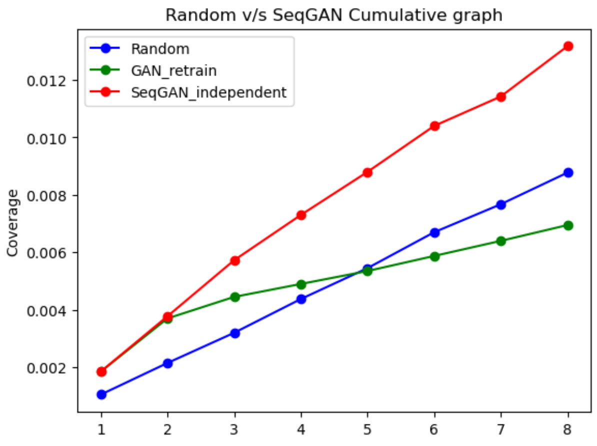


FIGURE 5.10: 10% Cumulative State Coverage Over Iterations

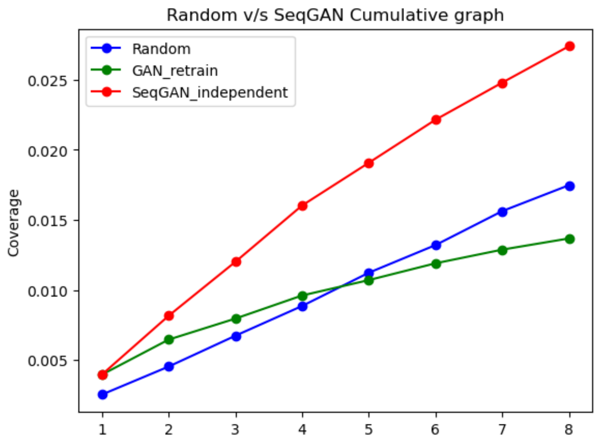
The number of iterations needed to achieve a 10% coverage of FSM transitions provides valuable insights into the efficiency of various approaches. When trained to reach this threshold, SeqGAN, with a new test bench added after every 10 iterations, achieves the target coverage in 26 iterations. In contrast, random generation methods require 32 iterations, while SeqGAN with its initial testbench iterated needs 30 iterations. These findings indicate that SeqGAN with batch renewal surpasses random generation methods in efficiency, requiring fewer iterations to attain the desired coverage threshold. This highlights SeqGAN's effectiveness in producing diverse and effective test cases for FSMs, facilitating quicker convergence toward the desired coverage level. Additionally, the number of iterations could be further decreased by introducing new test benches at shorter intervals.

5.1.1.3 40% feasible states

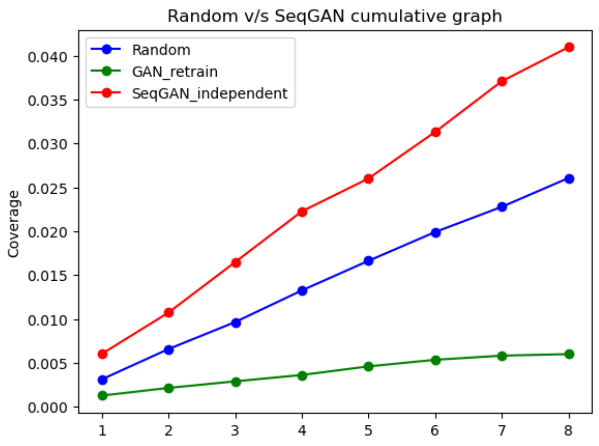
The figures illustrate the cumulative coverage analysis using test benches of varying sizes when 40% of states are feasible.



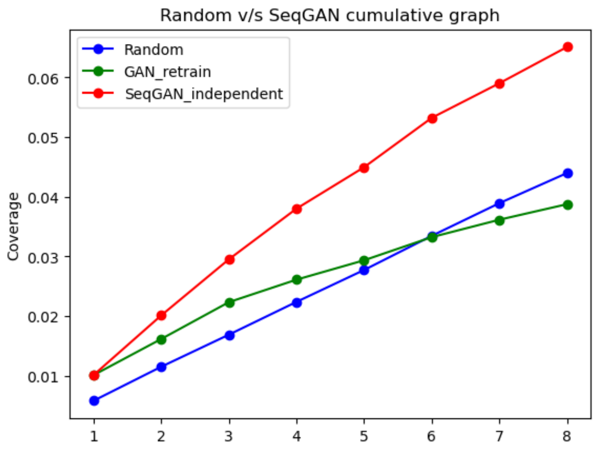
(a) Test bench containing 100 unique feasible states



(b) Test bench containing 200 unique feasible states



(c) Test bench containing 300 unique feasible states



(d) Test bench containing 500 unique feasible states

FIGURE 5.11: Cumulative State Coverage Over Iterations

The table below illustrates the percentage of state coverage after 8 iterations, corresponding to the data in figure 5.11.

% of states covered			
Initial test bench size	Random method(%)	SeqGAN(%)	SeqGAN Re-train(%)
100	0.81	1.30	0.61
200	1.32	2.71	1.71
300	2.60	4.00	0.60
500	4.30	6.51	3.82

TABLE 5.2: Percentage of states covered after 8 iterations when 40% of states are feasible

In all scenarios, we observed that SeqGAN, when trained using the initial test bench, consistently outperformed the other two methods. However, SeqGAN, with retraining, showed poor performance compared to random test case generation. Additionally, a comparison between random test case generation and SeqGAN revealed an increase of more than 2% coverage in SeqGAN after 8 iterations when the test bench size was 300 or 500.

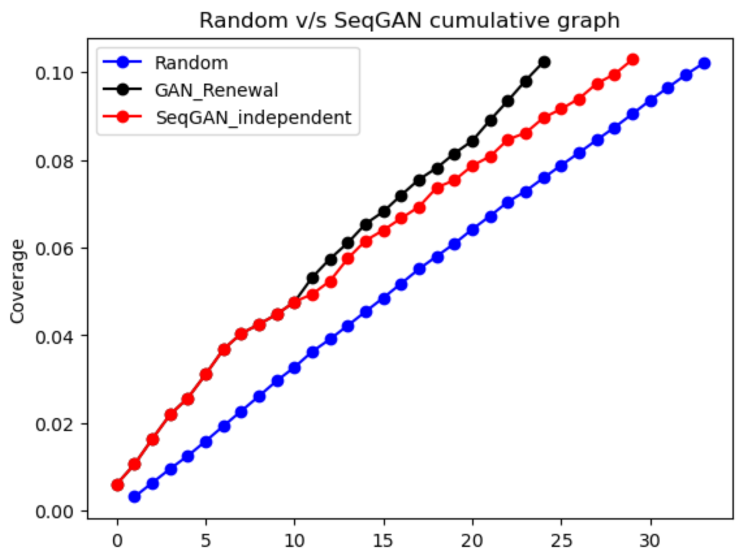


FIGURE 5.12: 10% Cumulative State Coverage Over Iterations

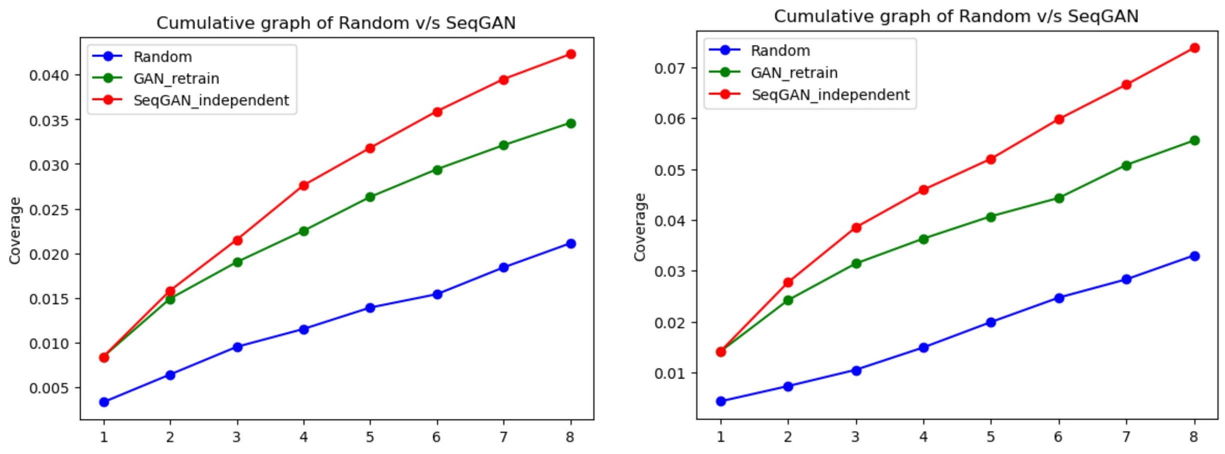
The number of iterations required to achieve a 10% coverage of Finite State Machine (FSM) transitions offers valuable insights into the efficiency of different approaches. SeqGAN, when trained to reach this threshold and with new test benches added after every 10 iterations, achieves the target coverage in 25 iterations. In comparison, random generation methods require 33 iterations, while SeqGAN, with its initial testbench iterated, needs 30 iterations. These findings suggest that SeqGAN with batch renewal outperforms random generation methods in efficiency, requiring fewer iterations to reach the desired coverage threshold.

5.1.2 Normal Distribution

A normal distribution, also known as a Gaussian distribution, is characterized by a single symmetric peak or mode, indicating that states are concentrated around a central value or range. The presence of a normal distribution can impact the performance of the SeqGAN model in generating test cases. Since SeqGAN learns from the distribution of input data to generate new samples, the characteristics of the normal distribution shape how the model understands and generates sequences.

In the scenario characterized by 10,000 feasible states out of a total of 100,000 possible states within the finite state machine (FSM), we examined the impact of varying the size of the test suite fed to the SeqGAN model. This investigation aims to understand how changes in the test suite size influence the performance and effectiveness of the SeqGAN model in generating test cases that adequately cover the FSM transitions.

The figures illustrate the cumulative coverage analysis using test benches of varying sizes when 10% of states are feasible. Also, the feasible states follow a normal distribution.



(a) Test bench containing 300 unique feasible states (b) Test bench containing 500 unique feasible states

FIGURE 5.13: Cumulative State Coverage Over Iterations

The table below illustrates the percentage of state coverage after 8 iterations, corresponding to the data in Figure 5.13.

% of states covered			
Initial test bench size	Random method(%)	SeqGAN(%)	SeqGAN Re-train(%)
300	2.11	4.26	3.46
500	3.30	7.38	5.53

TABLE 5.3: Percentage of states covered after 8 iterations when 10% of states are feasible

In both scenarios, we observed that SeqGAN, when trained using the initial test bench, consistently outperformed the other two methods. However, despite performing marginally better than other random generation methods, retraining SeqGAN after each iteration falls short of the coverage improvement achieved by SeqGAN trained independently. Additionally, a comparison between random test case generation and SeqGAN after 8 iterations revealed an increase of more than 2% coverage in SeqGAN when the test bench size was 300 or 500.

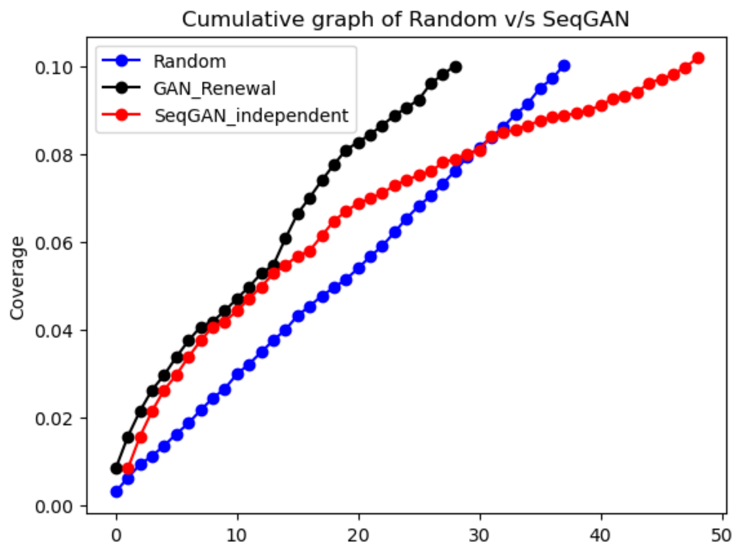


FIGURE 5.14: 10% Cumulative State Coverage Over Iterations

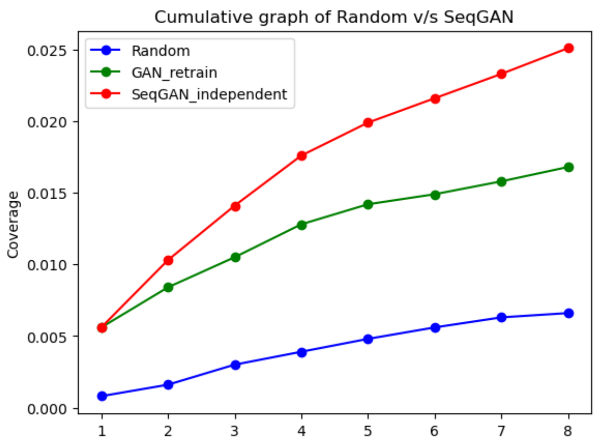
SeqGAN when fresh samples are introduced after every 15 iterations demonstrates improved performance, outperforming SeqGAN iterating on the same initial test bench. Additionally, it surpasses normal test case generation significantly. SeqGAN achieves 10% coverage in just 29 iterations, compared to 39 iterations required for random generation to reach the same coverage. However, SeqGAN with initial test cases performs poorly, exhibiting clear signs of overfitting as it begins producing duplicates after 20 iterations.

5.1.3 Log normal Distribution

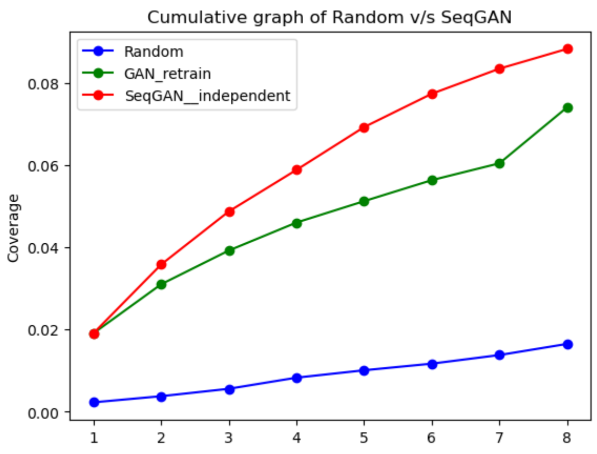
In contrast to a normal distribution, a log-normal distribution is characterized by asymmetric behavior, with a single mode but a skewed tail extending towards higher values. This distribution often emerges in scenarios where the logarithm of the variable follows a normal distribution, resulting in a skewed distribution for the original variable.

In the context of generating test cases for a finite state machine (FSM), a lognormal distribution might represent situations where most states cluster around a central value or mode, but some occasional extreme values or outliers significantly impact the overall distribution.

Within the context of a finite state machine (FSM) featuring 10,000 feasible states out of a potential 100,000, we investigated how altering the size of the test suite provided to the SeqGAN model influences its performance. The figures illustrate the cumulative coverage analysis using test benches of varying sizes when 10% of states are feasible. Also, the feasible states follow a log-normal distribution.



(a) Test bench containing 100 unique feasible states



(b) Test bench containing 300 unique feasible states

FIGURE 5.15: Cumulative State Coverage Over Iterations

The table below illustrates the percentage of state coverage after 8 iterations, corresponding to the data in Figure 5.15.

% of states covered			
Initial test bench size	Random method(%)	SeqGAN(%)	SeqGAN Re-train(%)
100	0.66	2.51	1.61
300	1.66	8.82	7.76

TABLE 5.4: Percentage of states covered after 8 iterations when 10% of states are feasible

In both scenarios, our observations consistently showed that SeqGAN, trained using the initial test bench, outperformed the other two methods. However, despite SeqGAN's better than random generation methods, retraining it after each iteration failed to match the coverage improvement achieved by independently trained SeqGAN.This is most likely due to overfitting since the SeqGAN is retrained on the samples it had produced, thus altering the distribution of newer samples. Furthermore, when comparing random test case generation with SeqGAN, there was a notable increase of over 7.5% coverage in SeqGAN when the test bench size was 300.

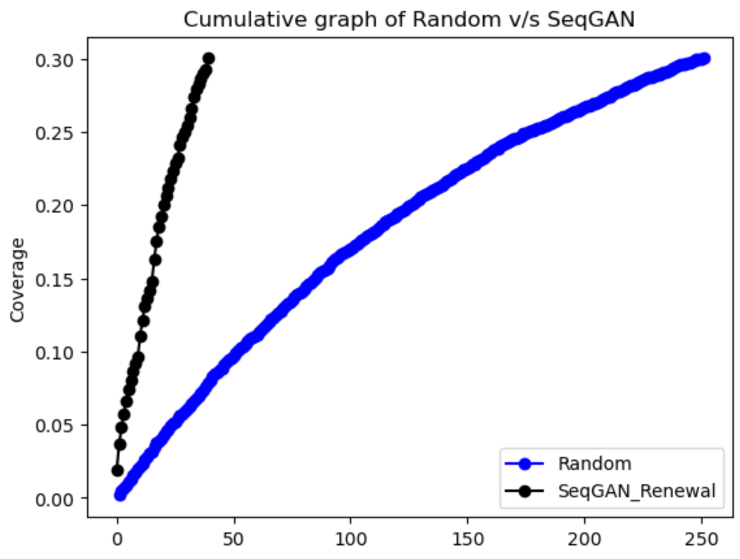


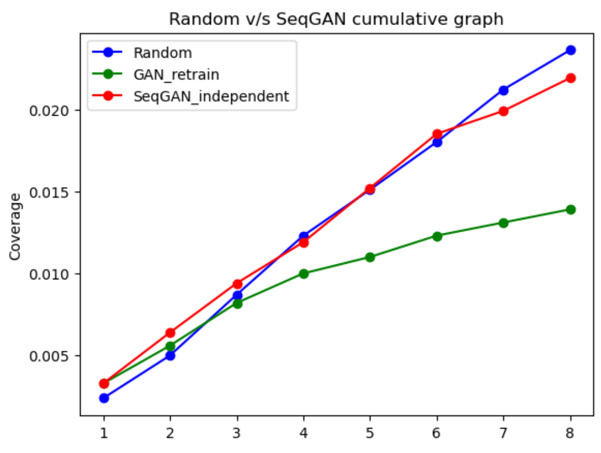
FIGURE 5.16: 30% Cumulative State Coverage Over Iterations

SeqGAN consistently outperforms random generation methods when introducing a fresh sample of the test bench after every 15 iterations. It achieved 30% coverage of states in just 40 iterations, whereas random generation required 251 iterations to reach the same coverage level.This comparison is shown in Figure 5.16

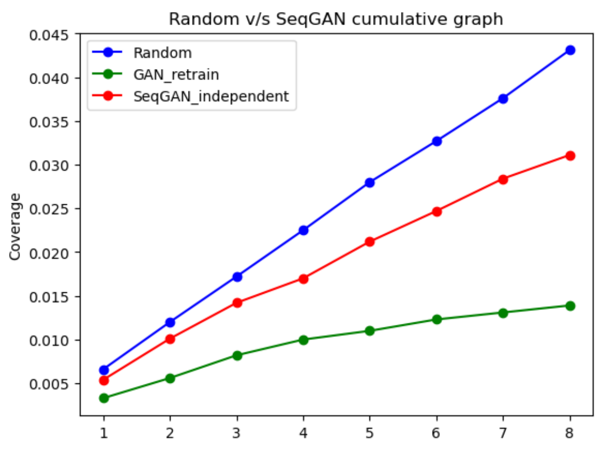
5.1.4 Uniform Distribution

A uniform distribution is characterized by a consistent probability density across its range, resulting in a flat shape. In the context of generating test cases for a finite state machine (FSM), a uniform distribution suggests that states are equally likely to occur within a specified range.

In the scenario featuring a finite state machine (FSM) with 10,000 feasible states out of a total of 100,000 possible states, we explored the impact of varying the test suite size, specifically considering sizes of 300 and 500, on the performance of the SeqGAN model. The figures illustrate the cumulative coverage analysis using test benches of varying sizes when 10% of states are feasible. Also, the feasible states follow a uniform distribution.



(a) Test bench containing 300 unique feasible states



(b) Test bench containing 500 unique feasible states

Figure 5.17: Cumulative State Coverage Over Iterations

The table below illustrates the percentage of state coverage after 8 iterations, corresponding to the data in Figure 5.17.

% of states covered			
Initial test bench size	Random method(%)	SeqGAN(%)	SeqGAN Re-train(%)
300	2.36	2.19	1.39
500	4.31	3.11	0.71

Table 5.5: Percentage of states covered after 8 iterations when 10% of states are feasible

In both cases, when comparing random test case generation with SeqGAN, random generation methods exhibited a notable increase in coverage after eight iterations, indicating a slight edge over SeqGAN methods in terms of coverage improvement. However, SeqGAN retraining performed poorly in this scenario. This suggests that the SeqGAN approach may not be as effective when dealing with uniform datasets, as it struggles to capture the necessary diversity and patterns present in the data required for generating meaningful test cases.

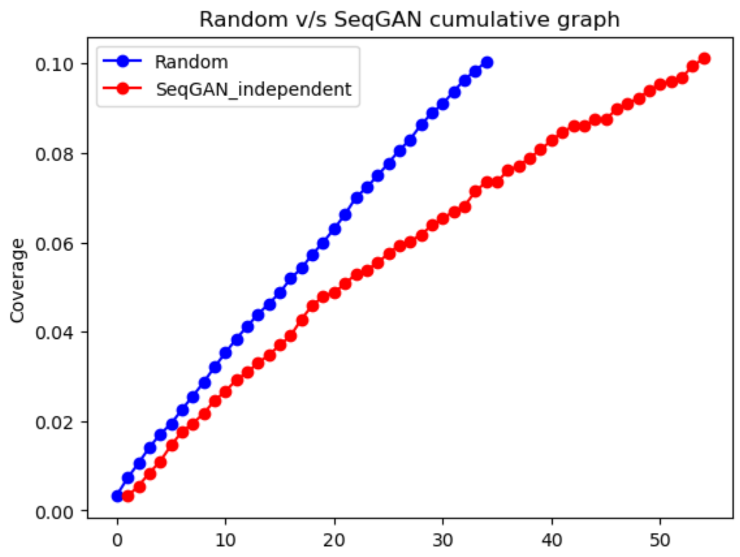


FIGURE 5.18: 10% Cumulative State Coverage Over Iterations

The observation that SeqGAN required 55 iterations compared to the 35 iterations needed by the random test case generation method to achieve 10% coverage indicates its relatively poor performance in this scenario. Furthermore, the graphical representation in Figure 5.18, illustrating SeqGAN's performance from the first iteration, reinforces the notion that the model struggles to produce satisfactory results, even when new test benches are introduced after finite iterations. However, since feasible states in FSM are designed by humans, it is highly unlikely that the set of feasible states follows a uniformly random distribution.

Chapter 6

Conclusions and Practical Implications

6.1 Conclusion

This comprehensive analysis sheds light on the effectiveness and challenges of using SeqGAN for test case generation, particularly in the context of different probability distributions. SeqGAN consistently outperforms random generation methods, demonstrating its superiority in enhancing coverage and efficiency. By examining various combinations of feasible states and test suite sizes, we can gain insights into how the complexity and size of the Finite State Machine (FSM), along with the diversity of the test suite, impact the performance and effectiveness of the SeqGAN model in generating test cases.

The presence of a log-normal distribution indeed presents unique challenges for SeqGAN in test case generation. While the model may effectively capture the central mode, accurately representing tail behavior can be challenging, potentially resulting in coverage gaps for rare or extreme state transitions. Similarly, for normal distributions, SeqGAN generally performs well in capturing the central mode but may struggle with outliers or tail variations, thereby affecting diversity and coverage. However, retraining SeqGAN with new test benches after finite iterations proves effective in addressing this challenge, enhancing the model's ability to capture underlying patterns.

For bimodal distributions, SeqGAN demonstrated robust performance, particularly when new test cases were introduced iteratively. However, when limited to only 2% of feasible states, SeqGAN encountered difficulties in capturing the underlying pattern in the dataset. Despite this challenge, SeqGAN still outperformed random generation techniques in effectiveness, although its performance was slightly less optimal compared to scenarios with a higher percentage of feasible states, such as 10% or 40%.

When confronted with a uniform distribution, SeqGAN encounters distinct challenges compared to distributions like normal, log-normal, or bimodal distributions. Because every state has an equal probability of occurrence, SeqGAN finds it challenging to identify underlying patterns or tendencies within the data. Consequently, generating diverse and representative test cases becomes more difficult, as the model lacks priorities for sequence generation. This limitation can lead to poorer performance of SeqGAN in terms of coverage improvement compared to other distributions where the data is more varied. However, it's worth noting that while it's rare for Finite State Machine (FSM) model states to follow a uniform distribution due to human design, encountering such distributions may still occur in certain contexts or datasets.

This underscores the importance of iterative refinement in machine learning models like SeqGAN, particularly when dealing with complex distributions. Continuous updates to training data and fine-tuning of parameters enable SeqGAN to learn effectively from the distribution of input data, generating sequences that closely resemble the behavior observed in the original dataset. Additionally, varying the number of iterations before providing a new test bench can potentially enhance the performance of SeqGAN. By adjusting this parameter, we can strike a balance between model learning and test bench refinement, enabling SeqGAN to adapt more effectively to the underlying dataset characteristics. Furthermore, this iterative process helps guard against overfitting, ensuring the robustness of the model.

6.2 Study Limitations and Future Prospects

The study's reliance on synthetic data due to privacy policies of companies and restrictions on accessing real-world data introduces certain limitations and considerations. While synthetic data provided a practical workaround to overcome constraints related to data availability, it's important to acknowledge potential disparities between synthetic and real data. These disparities could influence the generalizability and validity of the study findings.

The future prospects section outlines several potential directions for further research based on the findings of the study. Firstly, expanding the number of possible states in finite state machine (FSM) models could provide valuable insights into how more complexity impacts the performance of machine learning (ML) models for test case generation. Additionally, investigating alternative ML approaches, such as transformer-based models, has the potential to offer new insights and enhance the effectiveness of test case generation methods.

Exploring these avenues could contribute significantly to advancing the field of design verification and improving the efficiency and accuracy of semiconductor design processes. Moreover, efforts to validate the findings using real-world data, where feasible, would strengthen the robustness and applicability of the study's outcomes.

References

[1] Mohammad Mustafa Taye. Understanding of machine learning with deep learning: Architectures, workflow, applications and future directions. *Computers*, 12 (5):91, 2023. doi: 10.3390/computers12050091. URL https://doi.org/10.3390/computers12050091.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 3rd edition, 2009.

[3] Laith Alzubaidi, Juncheng Zhang, Ali Jawad Humaidi, et al. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):53, 2021. doi: 10.1186/s40537-021-00444-8.

[4] Machine learning techniques for personalised medicine approaches in immune-mediated chronic inflammatory diseases: Applications and challenges. *Frontiers in Pharmacology*, 12, September 2021. doi: 10.3389/fphar.2021.720694. URL https://doi.org/10.3389/fphar.2021.720694.

[5] Oleksandr Striuk and Yuriy Kondratenko. Generative adversarial neural networks and deep learning: Successful cases and advanced approaches. *International Journal of Computing*, September 2021. doi: 10.47839/ijc.20.3.2278.

[6] A. Aggarwal, M. Mittal, and G. Battineni. Generative adversarial network: An overview of theory and applications. *International Journal of Information Management Data Insights*, 1(1):Article 100004, 2021.

[7] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, 2017.

[8] Bahaa Osman and Mohamed Salem. Functional finite state machine paths coverage using systemverilog. URL `https://www.design-reuse.com/articles/24546/functional-fsm-paths-coverage-systemverilog`.

[9] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng, editors. *Electronic Design Automation*. City, 2009.

[10] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer, 2nd edition, 2003.

[11] The Octat Institute. Coverage in hardware verification. Internal document, October 24 2023. URL `https://www.theoctetinstitute.com/content/sv/coverage-in-hdl-verification/`.

[12] Jilles van Gurp and Jan Bosch. On the implementation of finite state machines. January 1999.

[13] Benoit Liquet, Sarat Moka, and Yoni Nazarathy. *Mathematical Engineering of Deep Learning*. CRC Press, 2024.

[14] Winda Kurnia Sari, Dian Palupi Rini, Reza Firsandaya Malik, and Iman Saladin B. Azhar. Sequential models for text classification using recurrent neural network. 2022.

[15] D Hazra, MR Kim, and YC Byun. Generative adversarial networks for creating synthetic nucleic acid sequences of cat genome. *International Journal of Molecular Sciences*, 23 (7):3701, Mar 2022. doi: 10.3390/ijms23073701.

[16] W. Bao, J. Yue, and Y. Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLoS ONE*, 12(7), 2017. doi: 10.1371/journal.pone.0180944.

[17] RM Schmidt. Recurrent neural networks (RNNs): A gentle introduction and overview. *arXiv preprint arXiv:1912.05911*, 2019.

[18] Wikipedia contributors. Recurrent neural network. Wikipedia. URL https://en.wikipedia.org/wiki/Recurrent_neural_network.

[19] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pages 1310–1318. PMLR, 2013.

[20] Chris Nicholson. A beginner's guide to lstms and recurrent neural networks, 2019. URL https://skymind.ai/wiki/lstm.

[21] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[22] T. Grover, V. Alladi, D. Chamola, D. Singh, and K.-K. R. Choo. Edge computing and deep learning enabled secure multitier network for internet of vehicles. *IEEE Internet of Things Journal*, 8(19):14787–14796, Oct. 2021. doi: 10.1109/JIOT.2021.3071362.

[23] Ian Goodfellow. Generative adversarial networks for text, 2016. URL http://goo.gl/Wg9DR7.

[24] L. Cai, Y. Chen, N. Cai, W. Cheng, and H. Wang. Utilizing amari-alpha divergence to stabilize the training of generative adversarial networks. *Entropy*, 22(4):410, 2020.

[25] Rob van Blommestein. Enhancing chip verification with AI & Machine Learning, July 12 2022. URL https://www.synopsys.com/blogs/chip-design/enhance-chip-verification-with-ai-and-machine-learning.html.

List of Figures

List of Tables