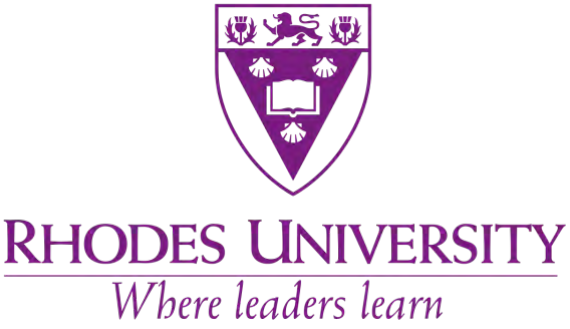
Towards an Artificial Intelligence-based Agent for Characterising the Organisation of Primes



Nicole Oyetunji

A dissertation submitted in fulfilment of the requirements for the degree of Master of Science in Applied Mathematics in the Faculty of Science at Rhodes University.

Supervisors: Dr. Patrice Okouma
Dr. Marcellin Atemkeng

December 2023

ACKNOWLEDGEMENTS

I am deeply grateful to my late mother, Suzette El-Mareé Oyetunji, for her unwavering support and immense sacrifices throughout my journey. She selflessly put her own dreams aside to provide me with the best opportunities in life, and to support me in all of my academic endeavours. She always believed in me, which gave me the strength to persevere during challenging times. Her love, dedication and diligence has shaped me into the person I am today, and I will always cherish her memory and be indebted to her.

My heartfelt appreciation goes to my supervisors, Dr P Okouma and Dr M Atemkeng, for their invaluable guidance and constant support. They believed in me and invested their time and effort to make this dissertation a reality, always going the extra mile. Their care, patience, encouragement, and provision of funding played a crucial role in my success. Furthermore, I extend my gratitude to Professor D Pollney, who helped me to be able to pursue Masters.

I would also like to extend my gratitude to Chris Van Der Spuy for his unwavering support in both my academic pursuits and general well-being. His encouragement and belief in me has been a source of inspiration throughout the years.

I extend my appreciation to my support system, which consists of my mother's church, the mother's prayer group and my mother's friends; Aunty Geraldine Rorke, Uncle Jonathan Rhode and Aunty Nesa. The love, prayers, and unwavering support from this community have been instrumental in my journey.

In addition, I want to thank my dear friends Mbeko Banjatwa, Sakhile Vanqa, Panashe Museki, Khuselwa Tembani, Vongai Chindeka, Sinoxolo Hale, Gina Mthembu, Tawanda Dhliwayo, Nanda Bambiso, Kamvalethu Vanqa, Sisipho Hamlomo, Siphendulwe Zaza, Sinako Gobozi, Aphelele Matshaya, Anele Sikhakhane, Stanley Mbiva, Lelona Stick, Nwabisa Mayeng, and Nwabisa Welaphi. Their friendship, support, and encouragement have been invaluable.

Lastly but most importantly, I would like to thank God for being by my side even at difficult times, and making this possible.

ABSTRACT

Machine learning has experienced significant growth in recent decades, driven by advancements in computational power and data storage. One of the applications of machine learning is in the field of number theory. Prime numbers hold significant importance in mathematics and its applications, for example in cryptography, owing to their distinct properties. Therefore, it is crucial to efficiently obtain the complete list of primes below a given threshold, with low relatively computational cost. This study extensively explores a deterministic scheme, proposed by Hawing and Okouma (2016), that is centered around Consecutive Composite Odd Numbers, showing the link between these numbers and prime numbers by examining their internal structure.

The main objective of this dissertation is to develop two main artificial intelligence agents capable of learning and recognizing patterns within a list of consecutive composite odd numbers. To achieve this, the mathematical foundations of the deterministic scheme are used to generate a dataset of consecutive composite odd numbers. This dataset is further transformed into a dataset of differences to simplify the prediction problem. A literature review is conducted which encompasses research from the domains of machine learning and deep learning. Two main machine learning algorithms are implemented along with their variations, Long Short-Term Memory Networks and Error Correction Neural Networks. These models are trained independently on two separate but related datasets, the dataset of consecutive composite odd numbers and the dataset of differences between those numbers. The evaluation of these models includes relevant metrics, for example, Root Mean Square Error, Mean Absolute Percentage Error, Theil U coefficient, and Directional Accuracy. Through a comparative analysis, the study identifies the top-performing 3 models, with a particular emphasis on accuracy and computational efficiency.

The results indicate that the LSTM model, when trained on difference data and coupled with exponential smoothing, displays superior performance as the most accurate model overall. It achieves a RMSE of 0.08, which significantly outperforms the dataset's standard deviation of 0.42. This model exceeds the performance of basic estimator models, implying that a data-driven approach utilizing machine learning techniques can provide valuable insights in the field of number theory. The second best model, the ECNN trained on difference data combined with exponential smoothing, achieves an RMSE of 0.28. However, it is worth mentioning that this model is the most computationally efficient, being 32 times faster than the LSTM model.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT iii

LIST OF FIGURES viii

LIST OF TABLES ix

LIST OF ABBREVIATIONS AND/OR ACRONYMS x

1 GENERAL INTRODUCTION 1

1.1 Introduction . . . 1

1.2 Aims and contributions . . . 3

1.3 Approach and Methodology . . . 3

1.4 Dissertation Outline . . . 4

2 CONSECUTIVE COMPOSITE ODD NUMBERS AND PRIMES 5

2.1 Mathematical foundations . . . 5

2.2 Main Results . . . 10

2.3 Data . . . 13

2.3.1 COCOON Data . . . 14

2.3.2 Difference Data . . . 17

2.4 Our Research Thrust . . . 22

2.5 Conclusion . . . 23

3 MACHINE LEARNING AND DEEP LEARNING: AN OVERVIEW 24

3.1 Introduction . . . 24

3.2 Neural Networks . . . 24

3.2.1 What is an artificial neuron? . . . 25

3.2.2 Activation Functions . . . 25

3.2.3 Perceptrons . . . 26

3.2.4 Multilayer Perceptrons . . . 28

3.2.5 Cost Functions . . . 29
3.2.6 Gradient Descent and Backpropagation . . . 31
3.3 Metrics for Model Evaluation . . . 36
3.4 Conclusion . . . 38

4 RECURRENT NEURAL NETWORKS 39
4.1 Introduction . . . 39
4.2 Traditional Recurrent Neural Networks . . . 39
4.2.1 Variations of Gradient Descent . . . 41
4.2.2 Adaptive Moment Estimation Optimiser (Adam) . . . 42
4.2.3 Backpropagation Through Time . . . 43
4.3 Long Short-Term Memory Networks . . . 45
4.3.1 Architecture . . . 45
4.3.2 Types of LSTMs . . . 48
4.3.3 Variations of LSTM Models . . . 48
4.3.4 Advantages and Disadvantages of LSTMs . . . 49
4.4 Further Background Study on Machine Learning Techniques . . . 49
4.4.1 Error Correction Neural Networks . . . 49
4.4.2 Advantages and Disadvantages of ECNNs . . . 51
4.4.3 Exponential Smoothing . . . 52
4.5 Conclusion . . . 56

5 RESEARCH METHODOLOGY 58
5.1 Introduction . . . 58
5.2 Prediction Model framework . . . 58
5.3 dataset . . . 60
5.3.1 Model Selection . . . 61
5.3.2 Hyperparameter Tuning . . . 62
5.4 Experimental Design . . . 62
5.4.1 Long Short-Term Memory Network . . . 62
5.4.2 Long Short-Term Memory Network with Exponential Smoothing . . . 63

5.4.3 Stacked Long-Short Term Memory Network 63
5.4.4 Error Correction Neural Networks 64
5.4.5 Exponential Smoothing Implementation 64
5.5 Model Evaluation 65
5.6 Conclusion 66

6 RESULTS **67**
6.1 Introduction 67
6.2 Long Short-Term Memory Network 67
6.2.1 Vanilla LSTM 67
6.2.2 Stacked LSTM 74
6.2.3 LSTM with Exponential Smoothing 76
6.3 Error Correction Neural Network 79
6.3.1 ECNN with Exponential Smoothing 82
6.4 Model Evaluation 86
6.4.1 Top Performing Models 89

7 DISCUSSION AND CONCLUSION **91**
7.1 Introduction 91
7.2 Study Overview 91
7.3 Performance Investigation 92
7.4 Final Model Evaluation 93
7.5 Implications of Research 93
7.6 Concluding Remarks: Limitations and Future Work 94

REFERENCES **99**

LIST OF FIGURES

2.1 Generating COCOON data given some threshold N. 14
2.2 The first 5 million COCOONs. 15
2.3 A closer look at COCOON data. 16
2.4 Distribution of 2's in the first 5 million COCOONs using a bin size of 5000. 18
2.5 Distribution of 4's in the first 5 million COCOONs using a bin size of 5000. 19
2.6 Distribution of 6's in the first 5 million COCOONs using a bin size of 5000. 20
2.7 Density of Prime Numbers in the first 5 million positive integers. 21

3.1 A single neuron (Hayken, 2014). 24
3.2 Activation Functions. 26
3.3 A Perceptron with m inputs (Staudemeyer and Morris, 2019). 26
3.4 Output Function of a Perceptron. 27
3.5 Hyperplane separating two different classes. 27
3.6 Multilayer Perceptron with n inputs (Staudemeyer and Morris, 2019). 28
3.7 Neural network trained using backpropagation (Hayken, 2014). 32

4.1 Repeating unit of an RNN (Shiri *et al.*, 2023). 40
4.2 Repeating unit of an LSTM (Shiri *et al.*, 2023). 45
4.3 Unfolded Error Correction Neural Network (Mvubu *et al.*, 2020). 50

5.1 Machine Learning Prediction Model Framework. 59
5.2 Stacked LSTM Model Architecture. 63

6.1 LSTM: RMSE for varying timesteps for the top 5 models. 69
6.2 Predicted vs Actual for an LSTM trained on COCOON data. 70
6.3 Predicted vs Actual for an LSTM trained on COCOON data. 71
6.4 Predicted vs Actual for an LSTM trained on Difference data. 72
6.5 Predicted vs Actual (scaled back to COCOON representation) for an LSTM trained on Difference data. 73
6.6 Predicted vs Actual for a Stacked LSTM trained on COCOON data. 74
6.7 Predicted vs Actual for a Stacked LSTM trained on Difference data. 75

6.8 Predicted vs Actual for LSTM with exponential smoothing, trained using COCOON data. 76
6.9 Predicted vs Actual for LSTM with exponential smoothing, trained using Difference data. 77
6.10 Predicted vs Actual (scaled back to COCOON representation) for LSTM with exponential smoothing, trained using Difference data. 78
6.11 Training vs Validation loss for ECNN trained using COCOON data. 79
6.12 Predicted vs Actual for ECNN trained using COCOON data. 79
6.13 Training vs Validation loss for ECNN trained using Difference data. 80
6.14 Predicted vs Actual for an ECNN trained on Difference data. 81
6.15 Predicted vs Actual (scaled back to COCOON representation) for ECNN trained on Difference data. 81
6.16 Training vs Validation loss ECNN with exponential smoothing, trained using COCOON data. 82
6.17 Predicted vs Actual for ECNN with exponential smoothing trained with COCOON data. 83
6.18 Training vs Validation loss for ECNN with exponential smoothing, trained using Difference data. 83
6.19 Predicted vs Actual for ECNN with exponential smoothing , trained using Difference data. 84
6.20 Predicted vs Actual (scaled back to COCOON representation) for ECNN with exponential smoothing, trained using Difference data. 85
6.21 Root Mean Square Error of different models deployed on various datasets. 87
6.22 Mean Absolute Percentage Error of different models deployed on various datasets. . 88
6.23 Theil U Coefficient of different models deployed on various datasets. 88

LIST OF TABLES

2.1 Complete list of COCOONs below 1000. 15

5.1 An example of a Difference dataset for COCOONs below 1000. 60
5.2 Architecture of the LSTM model. 63
5.3 Architecture of the ECNN model. 64
5.4 Architecture of the ECNN with Exponential Smoothing. 64

6.1 Values for LSTM hyperparameters in the Grid Search. 68
6.2 A summary of the top 5 LSTM models showing the model hyperparameters and test RMSE. 68
6.3 Summary of the best LSTM model. 70
6.4 Predictive accuracy of different models using MAPE. 86
6.5 Predictive accuracy of different models using RMSE. 86
6.6 Predictive accuracy of different models using DA. 87
6.7 Predictive accuracy of different models using Thiel U. 87
6.8 Statistical Summary of the Residuals of the Top 3 Models. 90
6.9 Model Runtime on Google Colab. 90

LIST OF ABBREVIATIONS AND/OR ACRONYMS

Adam	Adaptive Moment Estimation
BPTT	Backpropagation Through Time
COCOON	Consecutive Composite Odd Number
CV RMSE	Coefficient of Variation Root Mean Squared Error
DA	Directional Accuracy
ECNN	Error Correction Neural Network
ECNN es	Error Correction Neural Network with Exponential Smoothing
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ME	Mean Error
MSE	Mean squared Error
LSTM	Long Short-Term Memory Network
LSTM es	Long Short-Term Memory Network with Exponential Smoothing
RNN	Recurrent Neural Network
RMSE	Root Mean Square Error
SGD	Stochastic Gradient Descent
SSE	Sum of Squares Error

CHAPTER 1

GENERAL INTRODUCTION

1.1 INTRODUCTION

Machine learning has become a rapidly expanding area of research over the past few decades (Aggarwal *et al.*, 2022). With its inception in the late 1950's when the first machine learning model was developed, and it's rapid boom since the start of the 21^{st} century (Fradkov, 2020). Recent advancements in computational power and data storage have led to the development of machine learning algorithms that are able to efficiently process large data (Aggarwal *et al.*, 2022). One of the applications of machine learning is in the field of number theory. For instance, in a study conducted by Amir *et al.* (2022), supervised learning algorithms are used in the classification of real quadratic fields.

Prime numbers have always been an exciting topic for mathematicians, and with the power of machine learning techniques, innovative approaches related to prime number research are being discovered (Stekel *et al.*, 2018). For instance, in their recent study, Stekel *et al.* (2018) use deep learning techniques to predict the number of Goldbach partitions, given some even number. The Goldbach conjecture states that every even integer bigger than 2, can be represented as the sum of two primes, with each prime pair constituting a Goldbach partition. This research demonstrated the superior performance of deep learning techniques compared to analytical methods in the estimation of Goldbach partitions.

A prime number is a natural number greater than one, that is only divisible by one and itself. Prime numbers are significant in mathematics and have unique properties (Dawson, 2015). The Fundamental Theorem of Arithmetic dictates that any composite integer can be expressed as a product of prime numbers in only one way, aside from the permutations resulting from rearranging the prime factors (Dawson, 2015). This fact has implications, particularly in the field of encryption (Zaman, 2023). Primes are important, hence, there is a need for efficient methods to generate a complete list of primes below a given threshold.

The application of machine learning algorithms in the field of number theory is still in its early stages. However, despite increasing interest in this area, there are still gaps in research that need to be addressed. Prime numbers are notoriously difficult to predict because there is no known formula that can generate them efficiently (Grant and Ghannam, 2019). Machine learning is a powerful tool for finding patterns and regularities in data, researchers have used machine learning algorithms to predict prime numbers by analysing their distribution and patterns (Pylov *et al.*, 2023).

Various known methods exist to obtain the full list of prime numbers below a specified limit, with each method having its own advantages and disadvantages (Khairina, 2019). The Sieve of Eratosthenes is an ancient algorithm which was first used by Greek mathematician Eratosthenes to find the complete list of primes up to some limit (Mothebe, 2023). It does this by iteratively marking all proper multiples of each prime below the specified limit as composite, with all unmarked entries after the process is complete being prime (Helfgott, 2017). The Sieve of Eratosthenes has a computational cost of $\mathcal{O}(n \log(\log n))$ for a given threshold n (Ning and Kaeli, 2023; Helfgott, 2017). This method is effective when n is small but becomes impractical when the threshold is very large as the complexity grows exponentially.

The Sieve of Sundaram is a variation of the Sieve of Eratosthenes (Bufalo *et al.*, 2023). Like the Sieve of Eratosthenes, this method also involves marking the composite numbers up to a certain limit n. However, instead of marking the multiples of prime numbers, the Sieve of Sundaram marks all numbers which take the form $i + j + 2ij$, where $1 \leq i \leq j$, and $i + j + 2ij \leq n/2$ (Bufalo *et al.*, 2023). The primes are then obtained by subtracting the marked numbers from the list of natural numbers up to n (Bufalo *et al.*, 2023). According to Bufalo *et al.* (2023), the computational complexity of the Sieve of Sundaram is given by $\mathcal{O}(n \log n)$, which makes it impractical for large values of n. Given this drawback, a more efficient method with lower computational complexity was needed, particularly for large values.

The Sieve of Atkin is a modern modification and improvement of the Sieve of Sundaram. This method was first described by A. O. L. Atkin and D. J. Bernstein in 2004 (Martino, 2013). Like the Sieve of Sundaram, this method involves marking certain numbers, but it uses a more sophisticated algorithm to do so. The algorithm involves computing values of quadratic polynomials in two integer variables, and if those values are congruent to 1 modulo 2 or 1 modulo 3, marking those values

(Khairina, 2019). The computational complexity of the Sieve of Atkin is $\mathcal{O}(\frac{n}{\log(\log n)})$ (Martino, 2013), which makes it very efficient for large values of n, as demonstrated through a comparative study conducted by Abdullah *et al.* (2018). The improved computational efficiency of this algorithm stems from its refined sieving method, which minimises the need for brute force computations, resulting in a reduction of the overall number of operations performed.

Each of the methods mentioned have their own advantages and disadvantages. The Sieve of Eratosthenes is efficient for small values of n, but it becomes unfeasible for large values of n (Abdullah *et al.*, 2018). The Sieve of Sundaram has a higher computational complexity than the Sieve of Eratosthenes, and is impractical for large values of n Abdullah *et al.* (2018). On the other hand, the Sieve of Atkin provides a significant improvement, in terms of computational complexity, to both the Sieve of Eratosthenes and Sieve of Sundaram (Abdullah *et al.*, 2018; Khairina, 2019).

1.2 AIMS AND CONTRIBUTIONS

Machine learning algorithms have shown potential in predicting prime numbers, but there are still gaps in research that need to be addressed. Addressing these gaps will not only advance our understanding of prime numbers, but also open up new avenues of research in this area. This research aims to fill some of the gaps in research through the development of new algorithms, based on a deterministic scheme, which predict consecutive composite odd numbers. These numbers serve as a gateway to primes, as given any pair of consecutive composite odd numbers, the primes existing within that interval can easily be obtained based on the difference between the numbers. Ultimately, this research aims to contribute to the advancement of the field of number theory and machine learning.

1.3 APPROACH AND METHODOLOGY

The first stage of the research involves exposing and thoroughly exploring the mathematics behind the scheme centered around Consecutive Composite Odd Numbers, and its relation to primes. This will clearly demonstrate the underlying simplistic structure inherent in the set of COCOONs,

while also providing relatively simple relations to obtain COCOONs based on their last digit. This framework will allow for an adequate dataset to be generated, which will be used in the second phase of the research. The second phase of the research focuses on using machine learning techniques together with the deterministic scheme for the prediction of the $(n+1)^{th}$ COCOON, given any list of n COCOONs. Particularly focusing on two main algorithms of interest; Long Short Term Memory Networks which are detailed by Murugan (2018), and Error Correction Neural Networks (Mvubu *et al.*, 2020; Nandutu *et al.*, 2022). These algorithms have been chosen due to their capabilities in handling sequential data with long-term dependencies, a crucial requirement in this instance given the sequential nature of the data. During this stage, the algorithms will be continuously altered and optimised, using various techniques in order to achieve the highest possible accuracy. Finally, the methods and results will be evaluated. The factors limiting the research will be discussed, together with suggestions for possible further research in this area.

1.4 DISSERTATION OUTLINE

Chapter 2 extensively explores the deterministic scheme and data upon which this research is based, exposing the mathematical foundations of the scheme, as well as identifies mathematical expansions. In Chapter 3, the fundamentals of neural networks are extensively reviewed in order to build a strong theoretical background. Furthermore, Chapter 4 meticulously reviews existing literature and research on machine learning techniques associated with Recurrent Neural Networks. The methodology used to develop two main artificial intelligence agents capable of learning and recognizing patterns within a list of consecutive composite odd numbers, is presented and clarified in Chapter 5. After which, the results of the machine learning models from the different algorithms are presented in Chapter 6. The final chapter discusses and concludes the work, providing suggestions for possible further research in this domain.

CHAPTER 2

CONSECUTIVE COMPOSITE ODD NUMBERS AND PRIMES

2.1 MATHEMATICAL FOUNDATIONS

In this text, the set of natural numbers $\mathbb{N}$, includes 0, and $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$.

Definition 1. Any natural number $c \in \mathbb{N}^*$, is a composite odd number if and only if

1. c is odd,
2. c is not prime.

We note that 1 is neither prime nor composite.

Definition 2. Any m *and* $n \in \mathbb{N}$, are Consecutive Composite Odd Numbers (COCOONs) if and only if there exists no other composite odd number k such that $m < k < n$.

Let C be the set of odd composite numbers $\{9, 15, 21, 25, 27, 33, 35, 39, \cdots\}$, for every $c \in C$ the last digit of c is either 1, 3, 5,7 or 9. For each of the latter, we have:

$$1: \ c_1 = 10n + 1, \ n \in \mathbb{N}^*.$$
$$3: \ c_3 = 10n + 3, \ n \in \mathbb{N}^*.$$
$$5: \ c_5 = 10n + 5, \ n \in \mathbb{N}^*.$$
$$7: \ c_7 = 10n + 7, \ n \in \mathbb{N}^*.$$
$$9: \ c_9 = 10n + 9, n \in \mathbb{N}^*.$$

As c is composite and odd, it can be written in the form $c = p \times q$, where p and $q \in \{2n+1, n \in \mathbb{N}^*\}$. All the different possible forms of c can be obtained from $c_i \times c_j$, where $i, j \in \{1, 3, 5, 7, 9\}$. Since $c_i \times c_j = c_j \times c_i$, only one of them is considered in order to avoid redundant computations. Therefore, all of the different possible forms of c can be obtained by the following:

$$
\begin{aligned}
c_1 \times c_1 &= (10n_1 + 1)(10n_2 + 1) \\
&= 100n_1n_2 + 10n_1 + 10n_2 + 1 \\
&= 10(10n_1n_2 + n_1 + n_2) + 1 \\
&= 10M + 1,\ M \in \mathbb{N}^*. \qquad (2.1)
\end{aligned}
$$

$$
\begin{aligned}
c_1 \times c_3 &= (10n_1 + 1)(10n_2 + 3) \\
&= 100n_1n_2 + 30n_1 + 10n_2 + 3 \\
&= 10(10n_1n_2 + 3n_1 + n_2) + 3 \\
&= 10M + 3,\ M \in \mathbb{N}^*. \qquad (2.2)
\end{aligned}
$$

$$
\begin{aligned}
c_1 \times c_5 &= (10n_1 + 1)(10n_2 + 5) \\
&= 100n_1n_2 + 50n_1 + 10n_2 + 5 \\
&= 10(10n_1n_2 + 5n_1 + n_2) + 5 \\
&= 10M + 5,\ M \in \mathbb{N}^*. \qquad (2.3)
\end{aligned}
$$

$$
\begin{aligned}
c_1 \times c_7 &= (10n_1 + 1)(10n_2 + 7) \\
&= 100n_1n_2 + 70n_1 + 10n_2 + 7 \\
&= 10(10n_1n_2 + 7n_1 + n_2) + 7 \\
&= 10M + 7,\ M \in \mathbb{N}^*. \qquad (2.4)
\end{aligned}
$$

$$
\begin{aligned}
c_1 \times c_9 &= (10n_1 + 1)(10n_2 + 9) \\
&= 100n_1n_2 + 90n_1 + 10n_2 + 9 \\
&= 10(10n_1n_2 + 9n_1 + n_2) + 9 \\
&= 10M + 9,\ M \in \mathbb{N}^*. \qquad (2.5)
\end{aligned}
$$

$$
\begin{aligned}
c_3 \times c_3 &= (10n_1+3)(10n_2+3) \\
&= 100n_1n_2+30n_1+30n_2+9 \\
&= 10(10n_1n_2+3n_1+3n_2)+9 \\
&= 10M+9,\ M \in \mathbb{N}^*. \qquad (2.6)
\end{aligned}
$$

$$
\begin{aligned}
c_3 \times c_5 &= (10n_1+3)(10n_2+5) \\
&= 100n_1n_2+50n_1+30n_2+15 \\
&= 10(10n_1n_2+5n_1+3n_2+1)+5 \\
&= 10M+5,\ M \in \mathbb{N}^*. \qquad (2.7)
\end{aligned}
$$

$$
\begin{aligned}
c_3 \times c_7 &= (10n_1+3)(10n_2+7) \\
&= 100n_1n_2+70n_1+30n_2+21 \\
&= 10(10n_1n_2+7n_1+3n_2+2)+1 \\
&= 10M+1,\ M \in \mathbb{N}^*. \qquad (2.8)
\end{aligned}
$$

$$
\begin{aligned}
c_3 \times c_9 &= (10n_1+3)(10n_2+9) \\
&= 100n_1n_2+90n_1+30n_2+27 \\
&= 10(10n_1n_2+9n_1+3n_2+2)+7 \\
&= 10M+7,\ M \in \mathbb{N}^*. \qquad (2.9)
\end{aligned}
$$

$$
\begin{aligned}
c_5 \times c_5 &= (10n_1+5)(10n_2+5) \\
&= 100n_1n_2+50n_1+50n_2+25 \\
&= 10(10n_1n_2+5n_1+5n_2+2)+5 \\
&= 10M+5,\ M \in \mathbb{N}^*. \qquad (2.10)
\end{aligned}
$$

$$
\begin{aligned}
c_5 \times c_7 &= (10n_1 + 5)(10n_2 + 7) \\
&= 100n_1n_2 + 70n_1 + 50n_2 + 35 \\
&= 10(10n_1n_2 + 7n_1 + 5n_2 + 3) + 5 \\
&= 10M + 5,\ M \in \mathbb{N}^*. \qquad (2.11)
\end{aligned}
$$

$$
\begin{aligned}
c_5 \times c_9 &= (10n_1 + 5)(10n_2 + 9) \\
&= 100n_1n_2 + 90n_1 + 50n_2 + 45 \\
&= 10(10n_1n_2 + 9n_1 + 5n_2 + 4) + 5 \\
&= 10M + 5,\ M \in \mathbb{N}^*. \qquad (2.12)
\end{aligned}
$$

$$
\begin{aligned}
c_7 \times c_7 &= (10n_1 + 7)(10n_2 + 7) \\
&= 100n_1n_2 + 70n_1 + 70n_2 + 49 \\
&= 10(10n_1n_2 + 7n_1 + 7n_2 + 4) + 9 \\
&= 10M + 9,\ M \in \mathbb{N}^*. \qquad (2.13)
\end{aligned}
$$

$$
\begin{aligned}
c_7 \times c_9 &= (10n_1 + 7)(10n_2 + 9) \\
&= 100n_1n_2 + 90n_1 + 70n_2 + 63 \\
&= 10(10n_1n_2 + 9n_1 + 7n_2 + 6) + 3 \\
&= 10M + 3,\ M \in \mathbb{N}^*. \qquad (2.14)
\end{aligned}
$$

$$
\begin{aligned}
c_9 \times c_9 &= (10n_1 + 9)(10n_2 + 9) \\
&= 100n_1n_2 + 90n_1 + 90n_2 + 81 \\
&= 10(10n_1n_2 + 9n_1 + 9n_2 + 8) + 1 \\
&= 10M + 1,\ M \in \mathbb{N}^*. \qquad (2.15)
\end{aligned}
$$

Equations 2.1, 2.8, and 2.15 all lead to composite numbers ending in 1. Equations 2.2 and 2.14 lead to composite numbers ending in 3. Equations 2.3, 2.7, 2.10, 2.11, and 2.12 give us the composite numbers ending in 5. However, this can be represented simply as $10l+5$, where $l = 1, 2, 3...$, which gives us all of the composite numbers ending in 5. Equations 2.4, and 2.9 give us the composite numbers ending in 7. Finally, Equations 2.5, 2.6 and 2.13 give us the composite numbers ending in 9. This provides 11 different formulae to obtain composite numbers based on their last digit:

Last digit of 1:

$$(10n_1+1)(10n_2+1),\ n_1 \in \mathbb{N}^*,\quad n_2 \in \mathbb{N}^* \tag{2.16}$$

$$(10n_1+3)(10n_2+7),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N} \tag{2.17}$$

$$(10n_1+9)(10n_2+9),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N}. \tag{2.18}$$

Last digit of 3:

$$(10n_1+1)(10n_2+3),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N}^* \tag{2.19}$$

$$(10n_1+7)(10n_2+9),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N}. \tag{2.20}$$

Last digit of 5:

$$10l+5,\ l \in \mathbb{N}^*. \tag{2.21}$$

Last digit of 7:

$$(10n_1+1)(10n_2+7),\ n_1 \in \mathbb{N}^*,\quad n_2 \in \mathbb{N} \tag{2.22}$$

$$(10n_1+3)(10n_2+9),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N}. \tag{2.23}$$

Last digit of 9:

$$(10n_1+1)(10n_2+9),\ n_1 \in \mathbb{N}^*,\quad n_2 \in \mathbb{N} \tag{2.24}$$

$$(10n_1+3)(10n_2+3),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N} \tag{2.25}$$

$$(10n_1+7)(10n_2+7),\ n_1 \in \mathbb{N},\quad n_2 \in \mathbb{N}. \tag{2.26}$$

2.2 MAIN RESULTS

Theorem 1. For any pair of consecutive composite odd numbers (COCOONs) N_1 and N_2 ($N_2 > N_1$),

i) $N_2 - N_1 = 2$ or $N_2 - N_1 = 4$ or $N_2 - N_1 = 6$.

ii) If $N_2 - N_1 = 2$, then there is no prime number between N_1 and N_2.

iii) If $N_2 - N_1 = 4$, then there is one and only one prime number between N_1 and N_2.

iv) If $N_2 - N_1 = 6$, then there are two and only two prime numbers between N_1 and N_2.

i) *Proof.* Let us consider a pair of consecutive composite odd numbers, N_1 and N_2, where $N_2 > N_1$. Let S_k be the ordered set of natural numbers between any two odd multiples of 3:

$S_k = \{a \in \mathbb{N} : 3(2k+1) \leq a \leq 3(2k+3) \text{ with } k \in \mathbb{N}^*\}$.

Let us assume that N_1 and N_2 belong to distinct ordered sets of natural numbers between two odd multiples of 3, $N_1 \in S_{k_1}$ and $N_2 \in S_{k_2}$, where $k_1 \neq k_2$. It then follows that:

$S_{k_1} = \{a \in \mathbb{N} : 3(2k_1+1) \leq N_1 \leq 3(2k_1+3) \text{ with } k_1 \in \mathbb{N}^*\}$ and,

$S_{k_2} = \{a \in \mathbb{N} : 3(2k_2+1) \leq N_2 \leq 3(2k_2+3) \text{ with } k_2 \in \mathbb{N}^*\}$, where $k_1 \neq k_2$.

Since $N_1 < N_2$, we can deduce that $N_1 \leq 3(2k_1+3) \leq N_2$. This deduction leads to a contradiction as N_1 and N_2 are **consecutive** composite odd numbers, implying that there cannot be any other composite odd number between them, however, $3(2k_1+3)$ is composite and odd. Therefore, our initial assumption that any pair of COCOONs belongs to distinct ordered sets of natural numbers between two odd multiples of 3, has to be incorrect. This implies that any COCOON pair, N_1 and N_2, where $N_1 < N_2$, always lies within the same S_k.

Since any pair of COCOONs , N_1 and N_2, is always located within the same S_k, we can analyse each element of S_k to identify the potential COCOON pairs and compute their differences. The set S_k contains 7 natural numbers: $S_k = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7\}$, where s_1 and s_7 are composite and odd by definition. This gives us: $\{odd,\ even,\ odd,\ even,\ odd,\ even,\ odd\}$, since s_2, s_4 and s_6 are even, we rule them out as possible composite odd numbers. This leaves us with s_3 and s_5, which are either prime or composite and odd.

This leads us to the following:

Case 1:

Both s_3 and s_5 are prime. Therefore s_1 and s_7 are a COCOON pair, with $N_1 = s_1$ and $N_2 = s_7$. Recall that $s_1 = 3(2k+1)$ and $s_7 = 3(2k+3)$, therefore the set $S_k = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7\}$ can be written as:

$$\{3(2k+1), 3(2k+1)+1, 3(2k+1)+2, 3(2k+1)+3, 3(2k+1)+4, 3(2k+1)+5, 3(2k+3)\}.$$

Computing the difference between the COCOON pair yields:

$$\begin{aligned} N_2 - N_1 &= 3(2k+3) - 3(2k+1) \\ &= 6k + 9 - 6k - 3 \\ &= 6. \end{aligned}$$

Case 2:

s_3 is prime. Therefore s_1 and s_5 are a COCOON pair, with $N_1 = s_1$ and $N_2 = s_5$.

$$\begin{aligned} N_2 - N_1 &= 3(2k+1) + 4 - 3(2k+1) \\ &= 6k + 3 + 4 - 6k - 3 \\ &= 4. \end{aligned}$$

Additionally, s_5 and s_7 are also a COCOON pair, with $N_1 = s_5$ and $N_2 = s_7$.

$$\begin{aligned} N_2 - N_1 &= 3(2k+3) - (3(2k+1) + 4) \\ &= 6k + 9 - 6k - 3 - 4 \\ &= 2. \end{aligned}$$

Case 3:

s_5 is prime. Therefore s_1 and s_3 are a COCOON pair, with $N_1 = s_1$ and $N_2 = s_3$.

$$\begin{aligned} N_2 - N_1 &= 3(2k+1) + 2 - 3(2k+1) \\ &= 6k + 3 + 2 - 6k - 3 \\ &= 2. \end{aligned}$$

Additionally, s_3 and s_7 are also a COCOON pair, with $N_1 = s_3$ and $N_2 = s_7$.

$$\begin{aligned} N_2 - N_1 &= 3(2k+3) - (3(2k+1)+2) \\ &= 6k + 9 - 6k - 3 - 2 \\ &= 4. \end{aligned}$$

Case 4:

Neither s_3 nor s_5 are prime. Therefore s_1 and s_3 are a COCOON pair, with $N_1 = s_1$ and $N_2 = s_3$, $N_2 - N_1 = 2$.

Additionally, s_3 and s_5 are also a COCOON pair, with $N_1 = s_3$ and $N_2 = s_5$.

$$\begin{aligned} N_2 - N_1 &= 3(2k+1) + 4 - (3(2k+1)+2) \\ &= 6k + 3 + 4 - 6k - 3 - 2 \\ &= 2. \end{aligned}$$

Finally, s_5 and s_7 are also a COCOON pair, with $N_1 = s_5$ and $N_2 = s_7$, $N_2 - N_1 = 2$.

Therefore, the difference between any COCOON pair N_1 and N_2 is 2, 4 or 6.

□

ii) *Proof.* Let us look at the set of natural numbers between any two COCOONs N_1 and N_2, $\{n \in \mathbb{N} : N_1 \leq n \leq N_2\}$. Since N_1 and N_2 are natural numbers, if $N_2 - N_1 = 2$ then there exists only one natural number between them,

$$\{N_1, N_1 + 1, N_2\}.$$

Since N_1 is odd, it is of the form $2k+1 \implies N_1 + 1 = 2k + 2$, which is even. Since $N_1 \geq 9$, $N_1 + 1 \neq 2$, and since $N + 1$ is even and not equal to 2, it cannot be prime. Therefore if $N_2 - N_1 = 2$, there is no prime between N_1 and N_2.

□

iii) *Proof.* Similarly,

since N_1 and N_2 are natural numbers, if $N_2 - N_1 = 4$ then there exist only three natural numbers between them,

$$\{N_1, N_1 + 1, N_1 + 2, N_1 + 3, N_2\}.$$

$N_1 + 1$ is even, $N_1 + 2$ is odd, and $N_1 + 3$ is even. Since $N_1 + 1$ and $N_1 + 3$ are even and not equal to 2, they cannot be prime. Recall that N_1 and N_2 are Consecutive Composite Odd Numbers, therefore there are no other composite odd numbers between them, since $N_1 + 2$ is odd and not composite $N_1 + 2$ is prime.

□

iv) *Proof.* Similarly,

since N_1 and N_2 are natural numbers, if $N_2 - N_1 = 6$ then there exist only five natural numbers between them,

$$\{N_1, N_1 + 1, N_1 + 2, N_1 + 3, N_1 + 4, N_1 + 5, N_2\}.$$

$N_1 + 1$, $N_1 + 3$, and $N_1 + 5$ are even. Since $N_1 + 1$, $N_1 + 3$, and $N_1 + 5$ are even and not equal to 2, they cannot be prime. Additionally, $N_1 + 2$ and $N_1 + 4$ are odd, recall that N_1 and N_2 are Consecutive Composite Odd Numbers, therefore there are no other composite odd numbers between them. Since $N_1 + 2$ and $N_1 + 4$ are odd and not composite, they are prime. Specifically, they are twin primes since they differ by 2.

□

The only assumption in this theorem is that N_1 and N_2 are consecutive composite odd numbers. Once that condition is satisfied, the theorem holds.

2.3 DATA

In this research, two datasets were explored: the COCOON data and the difference data. Theorem 1 established that the difference between any COCOON pair is limited to 2, 4, or 6. This discovery introduced a second dataset for investigation alongside the COCOON dataset, namely the dataset of differences. The procedures used to acquire each dataset are detailed in the subsequent sections, where the distribution of the data is also examined..

2.3.1 COCOON Data

The 11 relations obtained in Section 2.1 (Equations 2.16 - 2.26) can be used to generate a dataset of COCOONs given some threshold N, this is shown in Figure 2.1. The algorithm takes in the desired threshold, N, as an input. Equations 2.16, 2.17, and 2.18 are used to compute composite odd numbers ending in 1 (COON1), which are then stored in the list C1. This process is iterated while each equation produces a composite odd number below N. The same procedure is applied to composite odd numbers ending in 3, 5, 7, and 9 using their respective equations. Subsequently, the lists of composite odd numbers categorised by the last digit are merged into a single list, sorted, and duplicates are removed. This results in a complete list of COCOONs below the specified threshold N (odd numbers below N not in this list are considered non-composite and therefore prime).

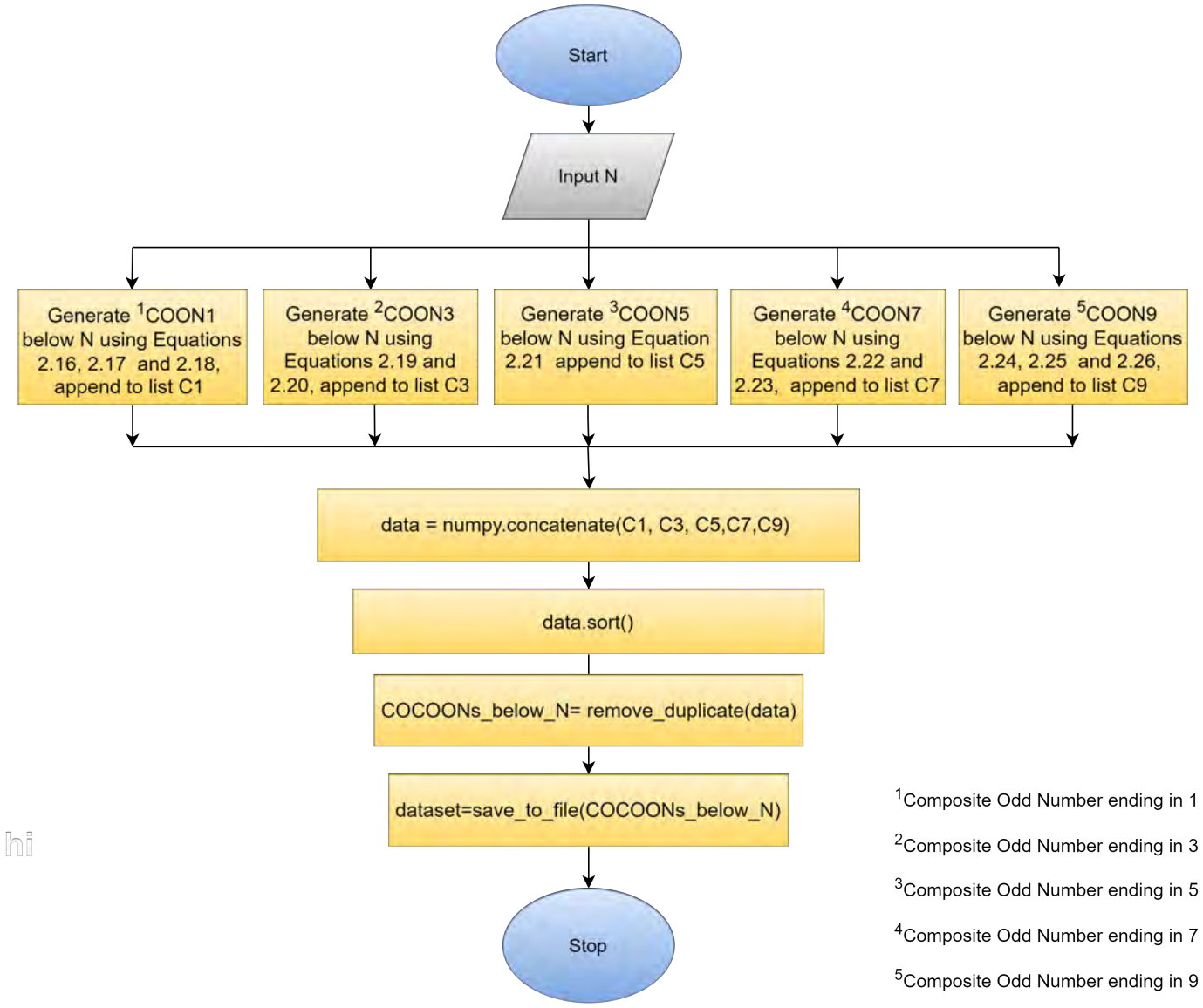


Figure 2.1: Generating COCOON data given some threshold N.

An example of the implementation of the procedure illustrated in Figure 2.1 is shown in Table 2.1, where the procedure was used to generate the complete list of COCOONs below 1000.

9, 15, 21, 25, 27, 33, 35, 39, 45, 49, 51, 55, 57, 63, 65, 69, 75, 77, 81, 85, 87, 91, 93, 95, 99, 105, 111, 115, 117, 119, 121
123, 125, 129, 133, 135, 141, 143, 145, 147, 153, 155, 159, 161, 165, 169, 171, 175, 177, 183, 185, 187, 189, 195, 201,
203, 205, 207, 209, 213, 215, 217, 219, 221, 225, 231, 235, 237, 243, 245, 247, 249, 253, 255, 259, 261, 265, 267, 273,
275, 279, 285, 287, 289, 291, 295, 297, 299, 301, 303, 305, 309, 315, 319, 321, 323, 325, 327, 329, 333, 335, 339, 341,
343, 345, 351, 355, 357, 361, 363, 365, 369, 371, 375, 377, 381, 385, 387, 391, 393, 395, 399, 403, 405, 407, 411, 413,
415, 417, 423, 425, 427, 429, 435, 437, 441, 445, 447, 451, 453, 455, 459, 465, 469, 471, 473, 475, 477, 481, 483, 485,
489, 493, 495, 497, 501, 505, 507, 511, 513, 515, 517, 519, 525, 527, 529, 531, 533, 535, 537, 539, 543, 545, 549, 551,
553, 555, 559, 561, 565, 567, 573, 575, 579, 581, 583, 585, 589, 591, 595, 597, 603, 605, 609, 611, 615, 621, 623, 625,
627, 629, 633, 635, 637, 639, 645, 649, 651, 655, 657, 663, 665, 667, 669, 671, 675, 679, 681, 685, 687, 689, 693, 695,
697, 699, 703, 705, 707, 711, 713, 715, 717, 721, 723, 725, 729, 731, 735, 737, 741, 745, 747, 749, 753, 755, 759, 763,
765, 767, 771, 775, 777, 779, 781, 783, 785, 789, 791, 793, 795, 799, 801, 803, 805, 807, 813, 815, 817, 819, 825, 831,
833, 835, 837, 841, 843, 845, 847, 849, 851, 855, 861, 865, 867, 869, 871, 873, 875, 879, 885, 889, 891, 893, 895, 897,
899, 901, 903, 905, 909, 913, 915, 917, 921, 923, 925, 927, 931, 933, 935, 939, 943, 945, 949, 951, 955, 957, 959, 961,
963, 965, 969, 973, 975, 979, 981, 985, 987, 989, 993, 995, 999

Table 2.1: Complete list of COCOONs below 1000.

Figure 2.2 shows a line plot of the first 5 million COCOONs, at first glance this appears to be a linear plot, however there are kinks in the line.

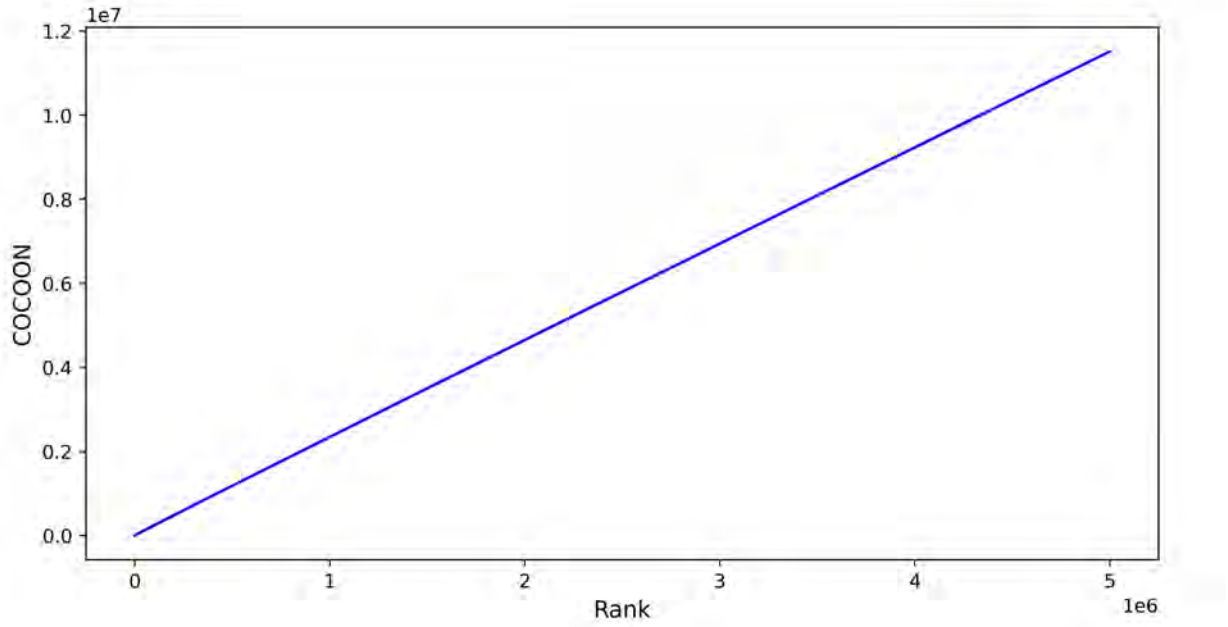


Figure 2.2: The first 5 million COCOONs.

Figure 2.3 shows a much closer look of COCOON data at a smaller scale, with each subfigure closely examining the first 10 COCOONs at varying ranks, visually capturing the non-linearity of the data.

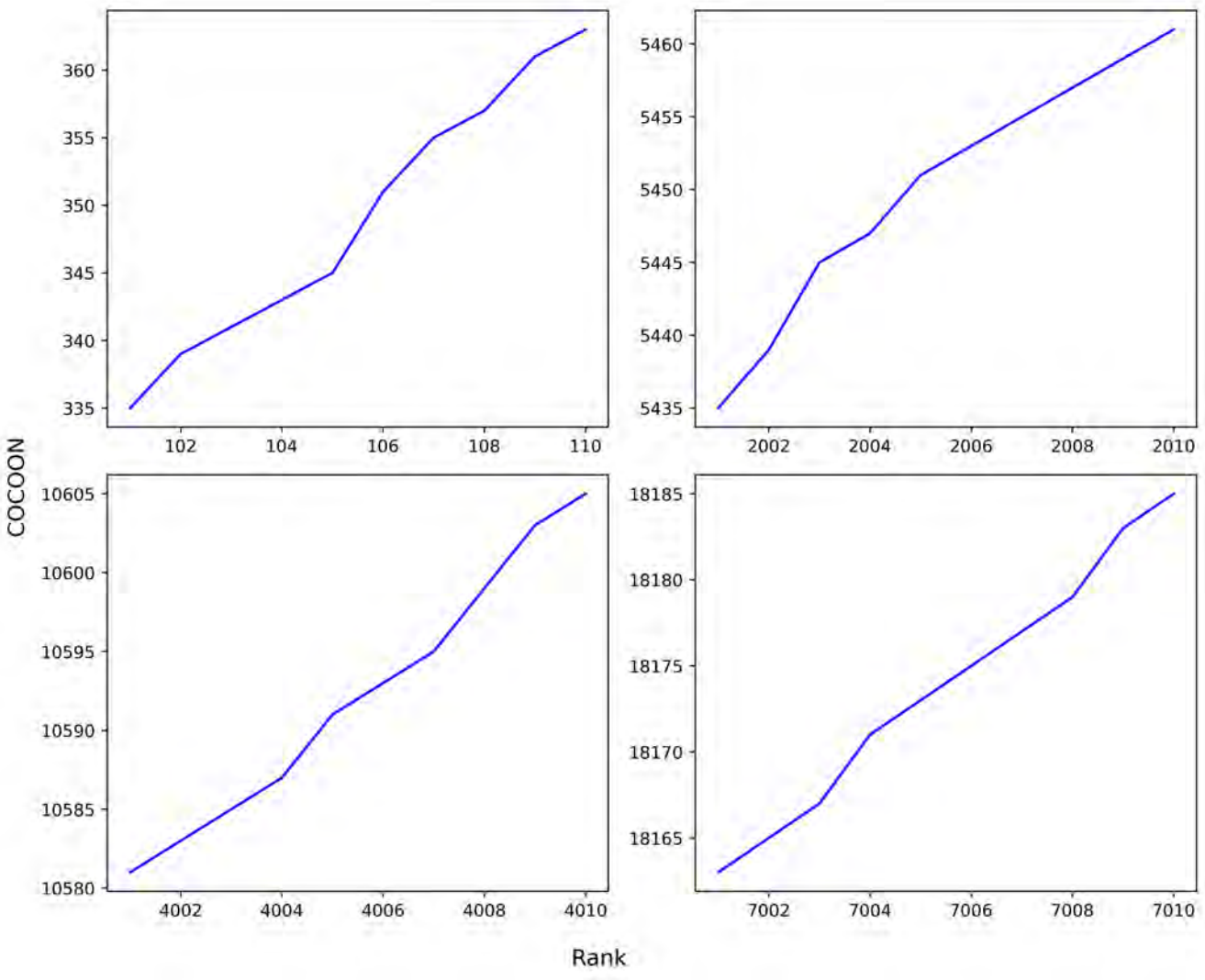


Figure 2.3: A closer look at COCOON data.

2.3.2 Difference Data

The COCOON data is used to obtain the difference dataset, as outlined in Algorithm 1. Given a list of COCOONs, C, the difference between each COCOON pair is computed iteratively across the list's length and stored in an array of differences. This gives us a full list containing difference data corresponding to COCOON data.

Algorithm 1 Computing Difference data

Require: C ▷ list of COCOONs
  $Diff \leftarrow [\,]$ ▷ array to store differences
  $k \leftarrow 0$
  while $k < length(C) - 2$ **do**
    $d \leftarrow C[k+1] - C[k]$ ▷ difference between COCOON pair
    $Diff \leftarrow Diff \cup d$
    $k+ = 1$
  end while

The differences in the COCOON pairs are either 2, 4 or 6, as shown in Section 2.2. Figures 2.4- 2.6 show the distribution of these differences in the first 5 million COCOONs. To generate these plots, the differences in the first 5 million COCOONs were computed as shown in Algorithm 1, these differences were subsequently partitioned into 1000 bins, with each bin containing 5000 entries. Thereafter, the number of 2's, 4's and 6's in each bin was computed. Figure 2.4 shows that the number of COCOON pairs with differences of 2 increase in a logarithmic manner as the bin number gets larger, and since a difference of 2 indicates no primes between the COCOON pair, this means that as the bin number gets larger, the less primes there are in that bin. Figure 2.5 and 2.6 show an exponential decrease in the number of COCOON pairs with a difference of 4 and 6 respectively as the bin number gets larger. Since primes are only found between COCOON pairs with a difference of 4 or 6, a rapid decrease in these differences with increasing bin number implies that primes become rarer and rarer as the bin number gets larger.

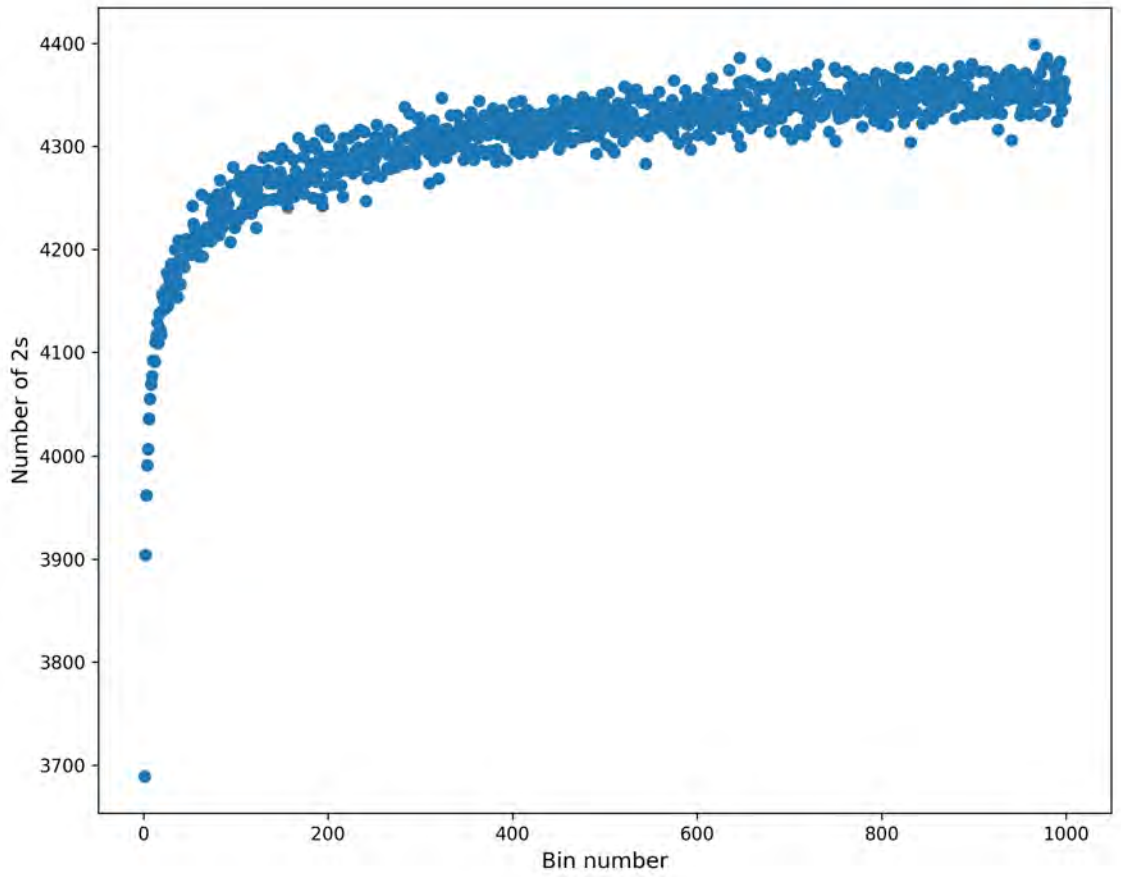


Figure 2.4: Distribution of 2's in the first 5 million COCOONs using a bin size of 5000.

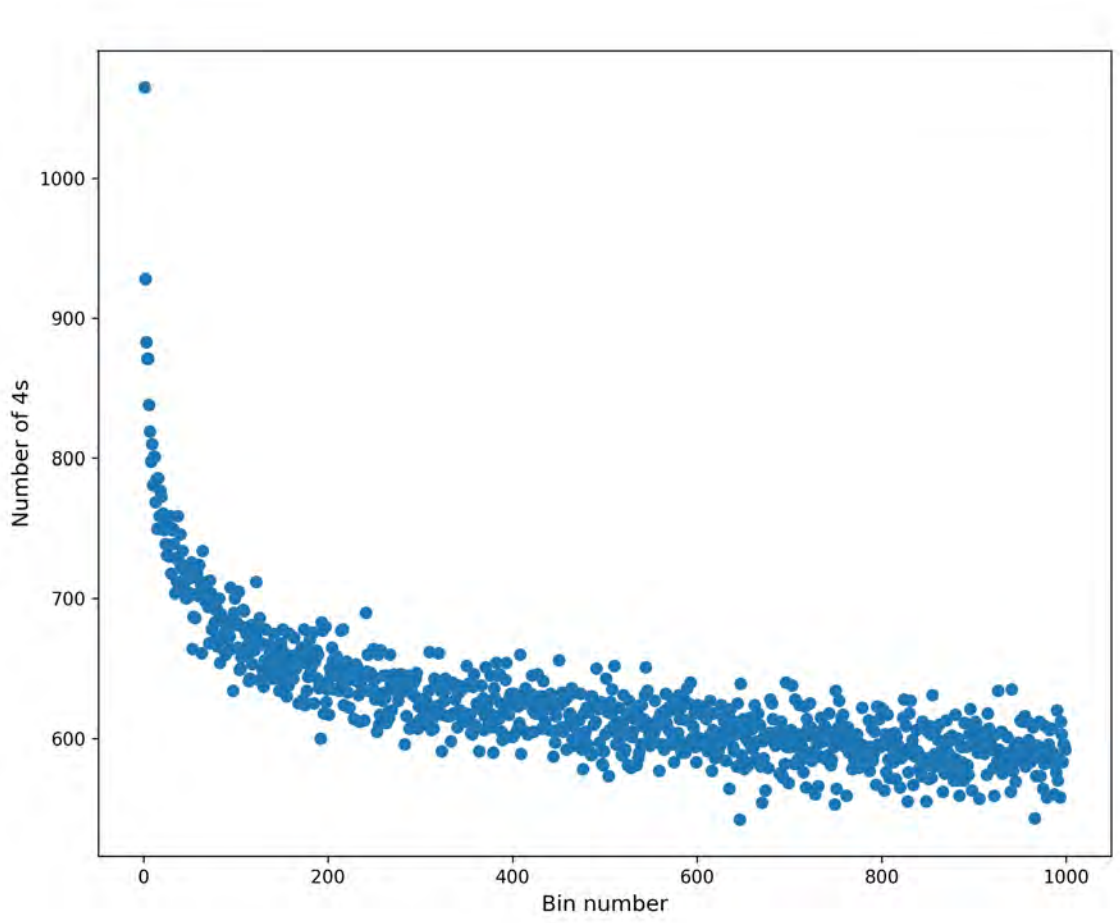


Figure 2.5: Distribution of 4's in the first 5 million COCOONs using a bin size of 5000.

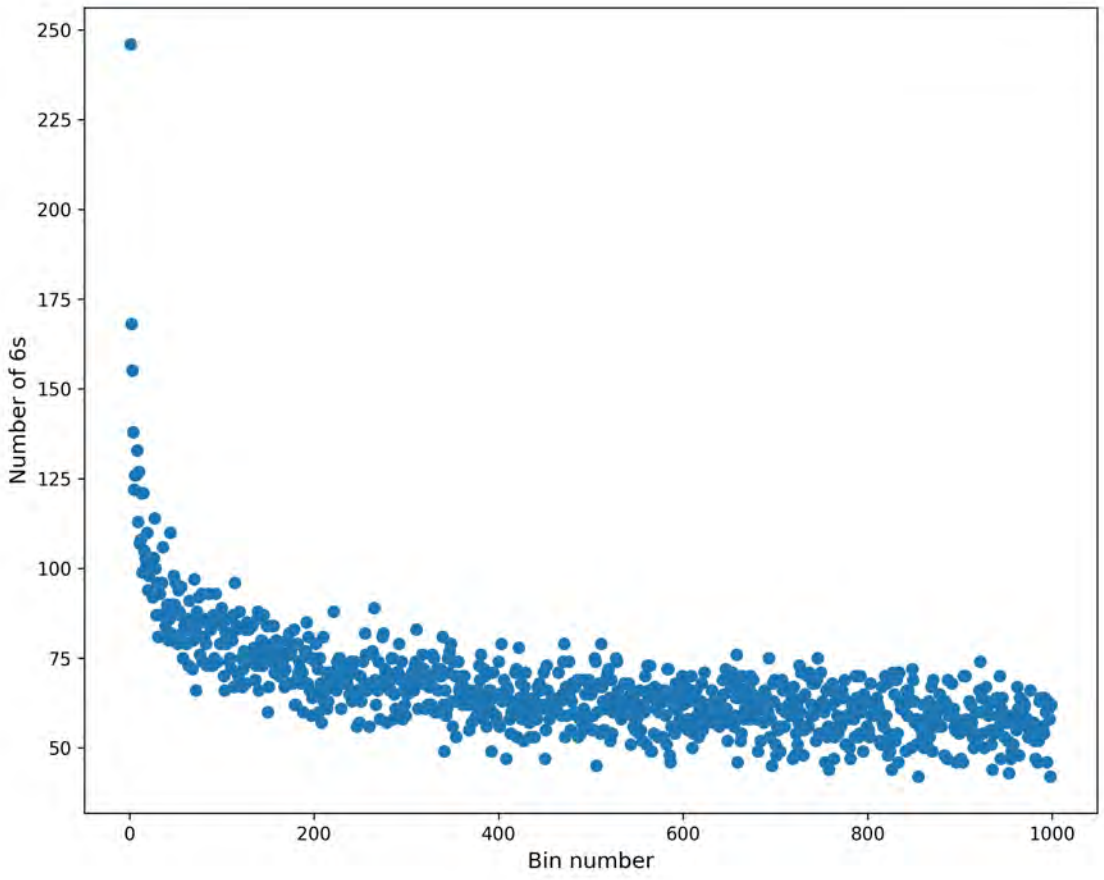


Figure 2.6: Distribution of 6's in the first 5 million COCOONs using a bin size of 5000.

These results are consistent with what we know from existing literature, particularly the Prime Number Theorem, which shows that prime numbers become rarer and rarer as the numbers become larger (Schroeder, 1984). Let $\pi(n)$ be the number of primes numbers less than or equal to n, according to the Prime Number Theorem, $\pi(n)$ is asymptotically equivalent to $n/\ln n$, that is,

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1. \tag{2.27}$$

This gives us an approximation of the number of positive integers less than or equal to n which are prime, $n/\ln n$. With $1/\ln n$ giving us the approximate percentage of these integers, otherwise known as the prime density. Since $\ln n$ is strictly increasing for all $n > 0$, $1/\ln n$ is strictly decreasing, showing that prime numbers become rarer as numbers become larger. This is demonstrated visually in Figure 2.7, which shows the empirical and theoretical density of primes as n increases.

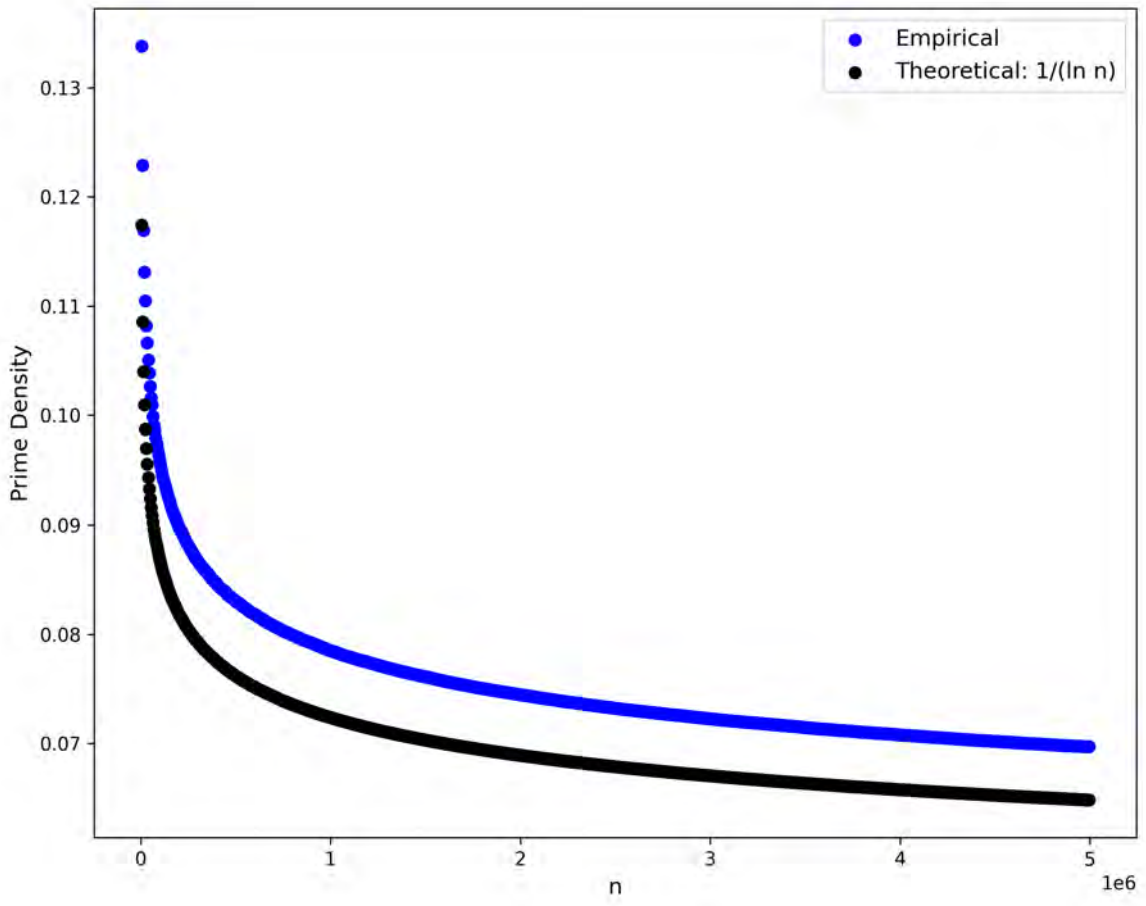


Figure 2.7: Density of Prime Numbers in the first 5 million positive integers.

2.4 OUR RESEARCH THRUST

The key research thrusts are as follows:

1. Exploring various implementations of Recurrent Neural Networks (RNNs), that can accurately predict the next composite odd number given list of n Consecutive Composite Odd Numbers, based on their internal structure.
2. Comparing and refining the accuracy and computational efficiency of the two main algorithms of interest.

In this chapter, we have established the following results:

1. Section 2.1 demonstrates that there are only 11 relations to obtain composite odd numbers based on their last digit, given a specific threshold value. This is shown specifically in Equations 2.16 - 2.26. This enables us to obtain a COCOON dataset, with values ending in 1, 3, 5, 7 and 9, given some threshold value.
2. The difference between any pair of COCOONs is either 2, 4, or 6. This is established in Theorem 1. These findings enable us to obtain a second dataset consisting of differences, to be used in the subsequent phase of the research, in order to simplify the prediction problem.
3. Theorem 1 further establishes that based on this difference, we can easily determine the existence of primes between the COCOON pair and compute them:
   (a) If the difference between N_2 and N_1 is 2, no primes exist between the COCOON pair.
   (b) If the difference is 4, there exists one and only one prime between the COCOON pair, and it can easily be obtained by $N_1 + 2$.
   (c) If the difference is 6, there exist exactly two primes between the COCOON pair, and they can be obtained by $N_1 + 2$ and $N_1 + 4$.

2.5 CONCLUSION

In this chapter, we established the mathematical foundations of the deterministic scheme used in this study (Section 2.1). This involved establishing 11 relations to obtain a COCOON based on its last digit. The implications of these findings were then explored in Section 2.2, where we established and proved the link between COCOONs and primes.

Building upon the insights from Sections 2.1 and 2.2, we introduced the COCOON dataset in Section 2.3. This dataset serves as the basis for obtaining a difference dataset, resulting in two main datasets for our study. This is crucial in order to effectively explore relevant machine learning techniques for the first thrust of the research, which will be discussed in Chapters 3 and 4.

To gain further insights into the data, we examined the distribution of the difference dataset within the first 5 million COCOONs. Our analysis revealed that this distribution aligns with existing literature on the distribution of primes.

CHAPTER 3

MACHINE LEARNING AND DEEP LEARNING: AN OVERVIEW

3.1 INTRODUCTION

This chapter introduces the fundamental concepts of machine learning and deep learning. Neural networks are discussed extensively by closely examining the artificial neuron, activation functions, single and multilayer perceptrons, cost functions, gradient descent and backpropagation in Sections 3.2.1 - 3.2.6 respectively. Finally, metrics for model evaluation are discussed in Section 3.3.

3.2 NEURAL NETWORKS

The human brain is very efficient at learning to recognise patterns, for example, identifying people or objects. It is made up of billions of biological neurons, which are the single processing unit of the brain (Hayken, 2014). These neurons are connected to each other to form a network (Hayken, 2014). A single neuron in the brain is made up of three main components, the dendrites which are a branch of input connections, the cell body which is the processing unit, and the axon which is a branch of output connections (Hayken, 2014). Figure 3.1 shows the basic structure of a neuron with labelled components.

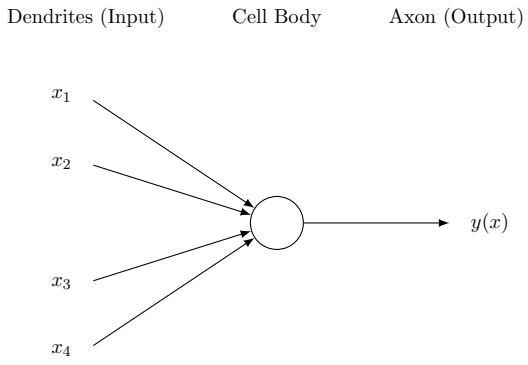


Figure 3.1: A single neuron (Hayken, 2014).

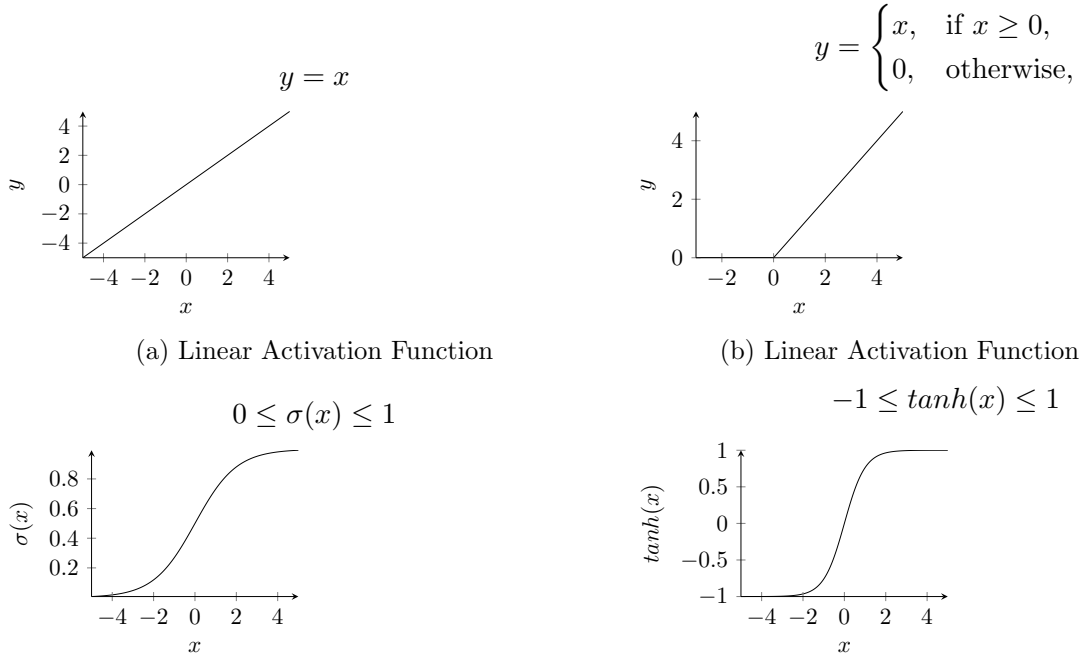
Each neuron takes in input electrical signals via the dendrites, which are processed in the cell body to produce an output signal (Hayken, 2014). The output signal is subsequently sent to the next neuron via the axon and the process is repeated, which enables the brain to learn (Hayken, 2014). A neural network is a collection of connected nodes mimicking neurons in the human brain, which learns to perform a task by analysing data for which there are known inputs and outputs (Hayken, 2014). It is comprised of three main components; an input layer, hidden layers, and an output layer, where each layer is comprised of artificial neurons (Hayken, 2014). The first neural network to be created was a single layer neural network called a perceptron.

3.2.1 What is an artificial neuron?

An artificial neuron is a machine simulation of a biological neuron. Similar to the biological neuron, it is comprised of a set of input nodes, a cell body in which the processing takes place, and an output node (Hayken, 2014). Each input is associated with a real number which gives a measure of the importance of that input in making the final prediction, therefore quantifying the strength of the connection between two neurons. This parameter is known as a weight. The values from the input nodes are multiplied by their weights, and the weighted inputs are added together to give a weighted sum. A constant value is added to provide a better fit, this value is known as the bias (Hayken, 2014). The fundamental operation of an artificial neuron is the summation of the weighted inputs and bias, which is passed into an activation function to get the activation response (output) of the neuron (Dubey *et al.*, 2021).

3.2.2 Activation Functions

An activation function maps the input of the neuron to a specified range, which can be finite or infinite. Numerous functions, discrete and continuous, can be used as activation functions. There are two types of activation functions, linear and non-linear (Dubey *et al.*, 2021). This is shown in Figure 3.2, where Subfigures a-b showcase examples of two commonly used linear activation functions, and Subfigures c-d showcase examples of two commonly used non-linear activation functions.



(a) Linear Activation Function

(b) Linear Activation Function (ReLu)

(c) Sigmoid Activation Function

(d) Hyperbolic Tangent Activation Tangent

Figure 3.2: Activation Functions.

3.2.3 Perceptrons

The perceptron was developed in the late fifties by Frank Roseblatt. It is a single layer neural network which acts as a linear classifier, by taking in multiple inputs and producing a single output (Ahmadi *et al.*, 2020; Staudemeyer and Morris, 2019). Figure 3.3 shows a diagram of a perceptron with m inputs, where x_i is the i^{th} input, and w_i is the weight corresponding to that input.

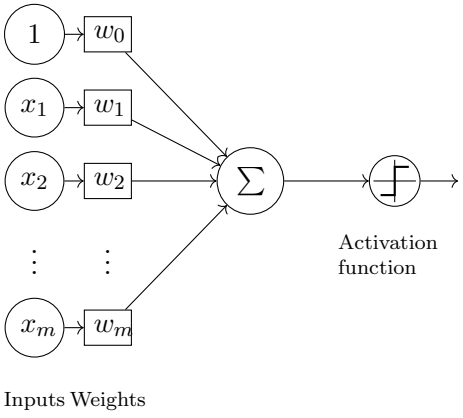


Figure 3.3: A Perceptron with m inputs (Staudemeyer and Morris, 2019).

Let $\boldsymbol{w}$, $\boldsymbol{x}$, and $\boldsymbol{b}$ be vectors containing weights, inputs and bias respectively,

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}. \tag{3.1}$$

The output of a perceptron is computed using a linear combination of the input values:

$$f(\boldsymbol{w}\boldsymbol{x} + \boldsymbol{b}) = f(\sum_i w_i x_i + b_i) \tag{3.2}$$

$$f(\boldsymbol{w}\boldsymbol{x} + \boldsymbol{b}) = \begin{cases} 1, & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} > 0 \\ 0, & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} \leq 0. \end{cases} \tag{3.3}$$

In Equation 3.3, $\boldsymbol{w} \cdot \boldsymbol{x} \ + \boldsymbol{b} > 0$ is the set of all points on one side of a hyperplane, while $\boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} \leq 0$ is the set of all points on the other side of the hyperplane.

Figure 3.4 depicts a graphical representation of the output of a perceptron, illustrating its division into two distinct classes (Schuld *et al.*, 2015; Staudemeyer and Morris, 2019). This is shown in Figure 3.5.

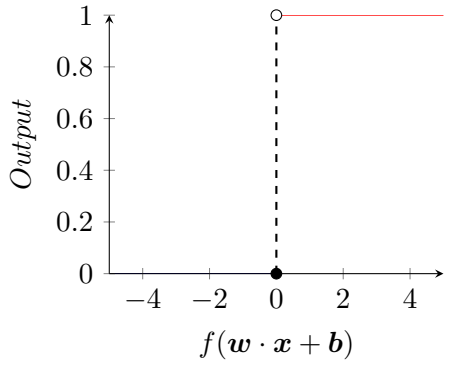


Figure 3.4: Output Function of a Perceptron.

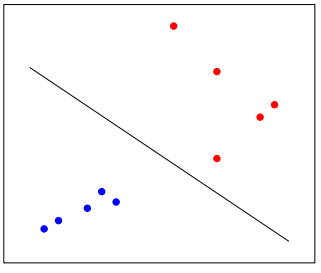


Figure 3.5: Hyperplane separating two different classes.

Perceptrons are simple and relatively easy to train, but are highly limited (Schuld *et al.*, 2015). The activation functions used in perceptrons are discrete and linear, making learning difficult (Schuld *et al.*, 2015). The use of these activation functions meant that perceptrons could not handle linearly inseparable data (Staudemeyer and Morris, 2019). This presents a challenge as it can be difficult to know ahead of time whether a problem is linearly separable or not. Linearly separable data refers to data which can be separated into distinct classes using a straight line, known as a linear decision boundary (Schuld *et al.*, 2015). Multilayer perceptrons were developed to overcome these limitations (Staudemeyer and Morris, 2019).

3.2.4 Multilayer Perceptrons

A multilayer perceptron is a neural network composed of multiple layers of perceptrons. The network comprises of an input layer, which receives input data and processes it in hidden layers (Staudemeyer and Morris, 2019). The hidden layers are situated amid the input and output layer, and are where the learning takes place (Staudemeyer and Morris, 2019). The final layer in the network is the output layer, which uses the output from the last hidden layer to make a prediction (Hayken, 2014; Staudemeyer and Morris, 2019). This architecture is shown in Figure 3.6, where x_i is the i^{th} input, a_i is the activation response in the hidden layer corresponding to that input, and y_i is the final prediction corresponding to input i.

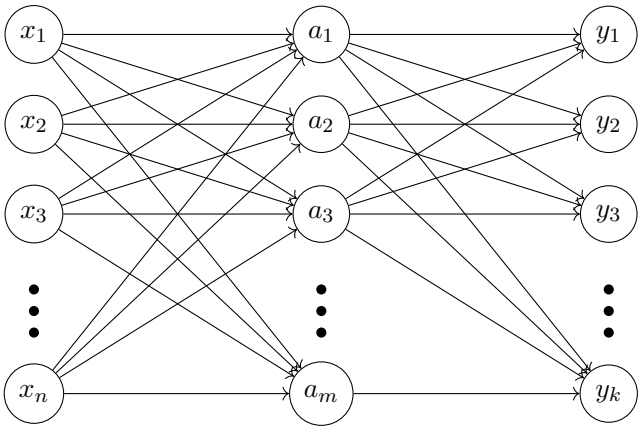


Figure 3.6: Multilayer Perceptron with n inputs (Staudemeyer and Morris, 2019).

During training, the predictions made by the network are compared to the target values, and an error is calculated by taking the difference between the output of the network and the target value (Staudemeyer and Morris, 2019). The overall error in the network can be quantified using a cost function, this is a function which assesses the training performance of a neural network (Ciampiconi *et al.*, 2023).

3.2.5 Cost Functions

The cost function, which is also called the loss or error function, is a function used to optimise a neural network (Ciampiconi *et al.*, 2023). It takes in two parameters; the output calculated by the network and the target output, which are used to compute an error across the entire training set (Ciampiconi *et al.*, 2023). There are different ways to quantify the error based on the nature of the problem. For prediction problems in which the target values are continuous, regression cost functions are used (Ciampiconi *et al.*, 2023). This includes:

1. Mean Error (ME)

   The mean of all the errors across the training set,

$$E = \frac{\sum_{i=1}^{n} y_i - \hat{y}_i}{n}, \tag{3.4}$$

   where y_i is the target value, and $\hat{y}_i$ is the activation response, otherwise known as the predicted value.

   The Mean Error is not commonly used as a cost function as it can be misleading (Morley *et al.*, 2018). The errors can be positive or negative, depending on whether the target value is bigger or less than the predicted value. Making it possible to get a mean error of 0, not because the data does not have errors, but because of cancelling out of error terms in the summation (Morley *et al.*, 2018). This limitation can be addressed several ways, for example, by squaring each error or by taking the absolute value of each error.

2. Mean Squared Error (MSE)

   The mean of all the squared errors across the training set

$$E = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}. \tag{3.5}$$

   The disadvantage of using the Mean Squared Error as a cost function is that errors due to outliers are amplified when they are squared (Ciampiconi *et al.*, 2023). This limitation can be addressed by quantifying the loss as the mean of all the absolute errors (Ciampiconi *et al.*, 2023).

3. Mean Absolute Error (MAE)

   This metric is commonly used as it is relatively simple to understand, and is not as sensitive to outliers as the MSE (Ciampiconi *et al.*, 2023). However, it provides less information about the errors compared to other cost functions. As it only provides information about the average absolute error, it does not provide any information about how the errors are spread.

$$E = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n}. \tag{3.6}$$

4. Sum of Squares Error (SSE)

   As detailed by Willmott *et al.* (2009), another cost function which is commonly used in the optimisation of machine learning algorithms is the Sum of Squares Error, as it is relatively simple and differentiable. However, the drawbacks of this metric is that it is sensitive to outliers and is biased towards larger sample sizes (Willmott *et al.*, 2009).

$$E = \sum_{i}^{n}(y_i - \hat{y}_i)^2. \tag{3.7}$$

After an appropriate cost function has been selected, the error across the training set is calculated. This error is subsequently used to adjust the weights in the network, which are continuously updated during this process via the gradient descent algorithm (Gui *et al.*, 2023; Hayken, 2014). This is a form of supervised learning, as the network is trained using known outputs (labelled data). In supervised learning, the network is given inputs and corresponding targets, and the weights are updated to minimise the difference between the outputs obtained from the network and the target values (Tibaldi *et al.*, 2023). However, in unsupervised learning, the network is only given the inputs, it identifies patterns existing within the data and uses them to update the weights (Tibaldi *et al.*, 2023).

3.2.6 Gradient Descent and Backpropagation

Gradient descent is a weight optimisation algorithm which minimises the error function by iteratively changing its parameters until the parameters resulting in the smallest error function are determined (Tran-Dinh and van Dijk, 2022). The weights are initialised randomly and the activation response of the network is calculated, the weights are then adjusted and updated, and this process is repeated until the error is close to zero (Staudemeyer and Morris, 2019; Tran-Dinh and van Dijk, 2022). The gradient descent algorithm makes use of the gradient of the error function with respect to the weights, these gradients are computed via backpropagation. As the gradient of the error function is iteratively calculated, the error function needs to be continuous and differentiable, motivating the use of activation functions such as the sigmoid function, as opposed to step functions used in single perceptrons (Hayken, 2014).

Let E be half of the Sum of Squares Error,

$$E = \frac{1}{2}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{3.8}$$

$$= \frac{1}{2}\sum_{i}^{n}(y_i - (w_i x_i + b_i)^2). \tag{3.9}$$

Gradient descent seeks to minimise this error by continuously updating the weights. The weights and biases are updated using the following learning rules respectively,

$$w_{i+1} = w_i - \triangle w_i \tag{3.10}$$

$$b_{i+1} = b_i - \triangle b_i, \tag{3.11}$$

where $\triangle w_i$ and $\triangle b_i$ denote the weight step and bias step respectively. The weights and biases are updated using the gradient of the error function and, these gradients are computed via backpropagation.

Backpropagation was introduced to address the limitations of the perceptron, giving rise to a neural network that can handle linearly inseparable data. This allowed multilayer perceptrons to be able to approximate continuous functions instead of only linear functions (Hayken, 2014). The neural network is trained using gradient descent on the Mean Squared Error, making use of the chain rule, and the network propagates forward until an output is produced (Waldo, 2022). The gradient of the loss function with respect to the weights, which is a vector of partial derivatives, is computed via backpropagation. This error is then propagated backwards and used to update the weights via gradient descent, this process is repeated until optimum weights are found, which minimise the error (Waldo, 2022). Figure 3.7 illustrates a neural network trained using backpropagation.

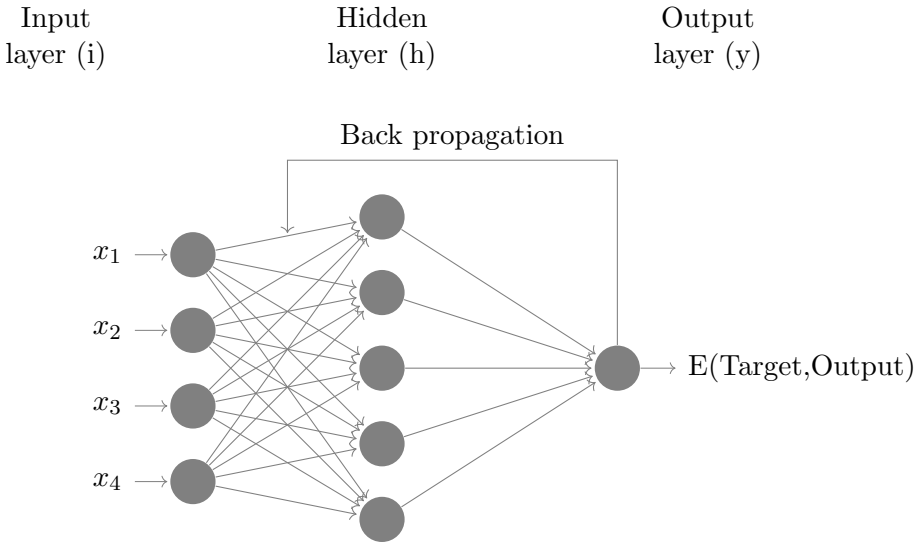


Figure 3.7: Neural network trained using backpropagation (Hayken, 2014).

The derivatives are computed as follows:

Let E be half of the Sum of Squares Error, as defined in Equation 3.8,

$$\triangle w_i \propto \frac{\partial E}{\partial w}. \tag{3.12}$$

Let a_y and a_h be the activation responses at the output and hidden layer respectively. The weight step from the hidden layer (h) to the output layer (y) is given by:

$$\triangle w_{yh} = -\eta \frac{\partial E}{\partial a_y} \frac{\partial a_y}{\partial w_{yh} a_h} \frac{\partial w_{yh} a_h}{\partial w_{yh}}, \tag{3.13}$$

where $w_{yh}a_h$ is the net input at the output layer, and η is the learning rate, a hyperparameter that determines the rate at which a neural network learns.

Let t_y and a_y be the target and activation response at the output layer respectively. The derivative of the error function with respect to the activation at the output layer is given by:

$$\begin{aligned}
\frac{\partial E}{\partial a_y} &= \frac{\partial (\frac{1}{2} \sum_y (t_y - a_y)^2)}{\partial a_y} \\
&= \frac{\partial (\frac{1}{2} \sum_y (t_y - a_y)^2)}{\partial a_y} \\
&= 2\frac{1}{2}(t_y - a_y)(-1) \\
&= -(t_y - a_y).
\end{aligned} \tag{3.14}$$

The derivative of the activation function (sigmoid) with respect to the net input at the output layer is given by:

$$\begin{aligned}
\frac{d}{dx}(\frac{1}{1+\mathrm{e}^{-x}}) &= (\frac{1}{1+\mathrm{e}^{-x}})(1 - (\frac{1}{1+\mathrm{e}^{-x}}) \\
&= a(x)(1 - a(x)) \\
&= \frac{\partial a_y}{\partial w_{yh} a_h} \\
&= a_y(1 - a_y).
\end{aligned} \tag{3.15}$$

The derivative of the net input with respect to the weight at the output layer is:

$$\frac{\partial w_{yh} a_h}{\partial w_{yh}} = a_h. \tag{3.16}$$

Substituting equations 3.14, 3.15, 3.16 into 3.13 yields:

$$\triangle w_{yh} = \eta(t_y - a_y)a_y(1 - a_y)a_h. \tag{3.17}$$

Let $\delta_y = (t_y - a_y)a_y(1 - a_y)$, the weight step from the hidden layer (h) to the output layer (y) is given by:

$$\triangle w_{yh} = \eta \delta a_h. \tag{3.18}$$

Let a_x be the activation response at the input layer. Similarly, for the weight step from the input layer (x) to the hidden layer (h) we have,

$$\begin{aligned}
\triangle w_{hx} &\propto -\sum (\frac{\partial E}{\partial a_y}\frac{\partial a_y}{\partial w_{yh} a_h}\frac{\partial w_{yh} a_h}{\partial a_h})\frac{\partial a_h}{\partial w_{hx} a_h}\frac{\partial w_{hx} a_h}{\partial w_{hx}} \\
&= (\sum_y (t_y - a_y)a_y(1 - a_y)w_{yh})a_h(1 - a_h)a_x \\
&= \sum_y \delta_y w_{yh} a_h(1 - a_h)a_x,
\end{aligned} \tag{3.19}$$

where $w_{hx}a_h$ is the net input at the hidden layer.

Let $\delta_h = \delta_y w_{yh} a_h(1 - a_h)$, Equation 3.19 becomes:

$$\sum_y \delta_y w_{yh} a_h(1 - a_h)a_x = \delta_h a_x. \tag{3.20}$$

The weight step from the input layer (x) to the hidden layer (h) is given by:

$$\triangle w_{hx} = \eta \delta_h a_x. \tag{3.21}$$

Similarly for the bias,

$$\begin{aligned}\frac{\partial E}{\partial b_{kl}} &= \frac{\partial E}{\partial a_l}\frac{\partial a_l}{\partial b_{kl}} \\ &= (t_l - a_l)\frac{\partial a_l}{\partial b_{kl}} \\ &= t_l - a_l, \end{aligned} \tag{3.22}$$

where b_{kl} denotes the bias between layer k and layer l, which can be the input, hidden, or output layer, a_l and t_l denote the activation response and target value in layer l respectively.

The weight step for the bias between layer k and layer l is given by:

$$\triangle b_{kl} = \eta(t_l - a_l), \tag{3.23}$$

where η is the learning rate.

Equations 3.18, 3.21 and 3.23 show that the weight and bias updates in the network are affected by the learning rate. A small learning rate may lead to very slow convergence, requiring the training data to be passed into the network a high number of times in order for a minimum to be found. On the other hand, a high learning rate may prevent convergence, causing the gradient descent algorithm to miss the local minimum completely (Ruder, 2017). The convergence rate for this algorithm is very slow, and is given by $\mathcal{O}(1/k)$, where k is the number of iterations (Mokhtari *et al.*, 2020). The learning rate has to be fine tuned carefully because if it is too fast, the local minimum can easily be missed, and if it is too slow, the algorithm may not converge (Mokhtari *et al.*, 2020). Additionally, Equation 3.14 shows that a crucial step in backpropagation is computing the derivative of the cost function, therefore, cost functions have to be differentiable (Ciampiconi *et al.*, 2023).

After training, an important aspect of developing a model is model evaluation, which allows for the performance of a trained model to be assessed. This is crucial in finding the best model suited to a problem (Ciampiconi *et al.*, 2023). There are various metrics which can be used to assess the performance of a prediction model, this is discussed in the following section.

3.3 METRICS FOR MODEL EVALUATION

These metrics assess the error in the predictions made by a model during the testing phase, evaluating the accuracy of a trained model. While some performance metrics can be used as cost functions, this does not apply to all of them. Performance metrics are different from cost functions as they are not used during training, and therefore do not need to be differentiable (Ciampiconi *et al.*, 2023). Contrary to this, cost functions need to be differentiable as they are used during backpropagation (Ciampiconi *et al.*, 2023). Metrics used in predictive models, where the outputs are continuous, are known as regression metrics (Ciampiconi *et al.*, 2023). These metrics all assess the error in the predictions in different ways. Regression metrics include:

1. Mean Squared Error (MSE)

   As previously shown in Section 3.2.5, this function can be used as a cost function when training models which learn by gradient descent. However, it can also be used as a performance metric to assess the overall performance of a model. In this case, it is the mean of all the squared errors across testing dataset (Ciampiconi *et al.*, 2023). As mentioned in Section 3.2.5, although squaring the errors is advantageous in that it prevents them from cancelling out during the summation when dealing with negative errors, it also magnifies them, inflating the MSE. This is addressed by taking the square root of the Mean Squared Error, which deflates inflated MSE values while still keeping them positive.

2. Root Mean Squared Error (RMSE)

   This is the square root of the Mean Squared Error, but unlike the MSE, it preserves the original units of the target and predicted values (Ciampiconi *et al.*, 2023). It is commonly

used and is considered a good performance metric (Botchkarev, 2019).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}. \tag{3.24}$$

However, this metric can only be used for comparison between models which have the same error units (Ciampiconi *et al.*, 2023). It also does not provide any information as to whether the error is an overestimate or an underestimate.

3. Coefficient of Variation Root Mean Squared Error (CV RMSE)

   This metric is obtained by normalising the Root Mean Squared Error by the average of the target values (Botchkarev, 2019).

$$CVRMSE = \frac{1}{\frac{\sum_{i=1}^{N} y_i}{N}} \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}. \tag{3.25}$$

   Normalising the RMSE score is advantageous in that it allows for comparisons to be made between models using different datasets and different units (Botchkarev, 2019).

4. Mean Absolute Error (MAE)

   This function, like the MSE, can also be used as a cost function. However, in this case, it is the mean of all the absolute errors across the testing data. It is also commonly used as a performance metric and is favourable in that it changes linearly with the error, preventing inflated or disproportionate scores (Botchkarev, 2019; Ciampiconi *et al.*, 2023). The original units of the target and predicted values are also preserved in the score.

5. Mean Absolute Percentage Error (MAPE)

   The mean of all the absolute percentage errors across the testing data, indicating how far off predictions are from target values on average (Botchkarev, 2019; Ciampiconi *et al.*, 2023).

$$MAPE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)/y_t}{n}. \tag{3.26}$$

This metric is commonly used as it is relatively simple to understand Ciampiconi *et al.* (2023). However, the disadvantage of using this metric is that it is biased towards predictions that are smaller than the target values. A prediction that is lower than the target by the same amount that another prediction is higher, will have a lower MAPE score despite the fact that their absolute errors are the same. Because of this, the Root Mean Square Error (RMSE) has also been included as a metric, as it does not have any bias towards predictions that are smaller or higher than the target.

3.4 CONCLUSION

In conclusion, this chapter provided a comprehensive introduction to the fundamental concepts of machine learning and deep learning. It explored the fundamentals of neural networks, closely examining artificial neurons, perceptrons, cost functions, gradient descent, backpropagation and metrics for model evaluation. A link was shown between gradient descent and backpropagation, motivating the use of differentiable functions as cost functions. Additionally, the distinction between cost functions and evaluation metrics was highlighted. While cost functions aim to optimise the model during training, evaluation metrics are used to assess the model's performance once the model is trained. Furthermore, the learning difficulties associated with the gradient descent algorithm, such as slow convergence and problems with locating local minima, were also discussed. Finally, metrics for model evaluation were discussed as means for evaluating and assessing trained models.

These concepts serve as the building blocks for understanding and implementing machine learning algorithms. Moving forward, the next chapter will delve into machine learning techniques associated with Recurrent Neural Networks. This will further expand understanding of how neural networks can be used to process sequential data and make predictions based on temporal dependencies.

Overall, this chapter has laid a solid foundation for understanding the core concepts of machine learning and deep learning, setting the stage for further exploration of advanced techniques in the upcoming chapter.

CHAPTER 4

RECURRENT NEURAL NETWORKS

4.1 INTRODUCTION

This chapter extensively reviews Recurrent Neural Networks (RNNs), providing a comprehensive background study on related deep learning techniques. Section 4.2 establishes the foundations of traditional Recurrent Neural Networks, and also explores variations of gradient based learning algorithms, introducing techniques such as Stochastic Gradient Descent (SGD), and the Adaptive Moment Estimation Optimiser (Adam) in Sections 4.2.1 and 4.2.2 respectively. Section 4.2.3 introduces Backpropagation Through Time (BPTT), and investigates the challenges that can arise when training RNNs with gradient based learning methods, such as the vanishing or exploding gradient problem. A more advanced type of RNN, the Long Short-Term Memory Network (LSTM), is introduced and discussed in depth in Section 4.3. Error Correction Neural Networks, which are a sophisticated type of RNN, are also examined in Section 4.4.1, together with several exponential smoothing techniques for optimisation in Section 4.4.3.

4.2 TRADITIONAL RECURRENT NEURAL NETWORKS

Deep learning has seen a bloom in recent years with neural networks, supervised and unsupervised, being widely used because of their proficiency and speed (Gao and Guan, 2023). However, traditional neural networks make use of the assumption that any two succeeding inputs in the training and test sets are independent of each other, making them inefficient at handling sequential data as the state of the network is discarded after processing each input (Schmidt, 2019). Recurrent Neural Networks were established to address this problem, they are a generalization of feed-forward neural networks (traditional neural networks), with a recursive nature, that are used when dealing with sequential data (Rezk *et al.*, 2020). They differ from traditional neural networks in that their inputs are related to each other, in addition they also have memory, which is used to process sequences of data, allowing them to be useful in sequence prediction problems (Rezk *et al.*, 2020).

An RNN is comprised of an input layer, hidden layers and an output layer. The networks repeating unit is a single neural network layer (Schmidt, 2019). This is shown in Figure 4.1, where $h^{(t-1)}$ and $h^{(t)}$ are the previous and current hidden states respectively, and x_t is the current input.

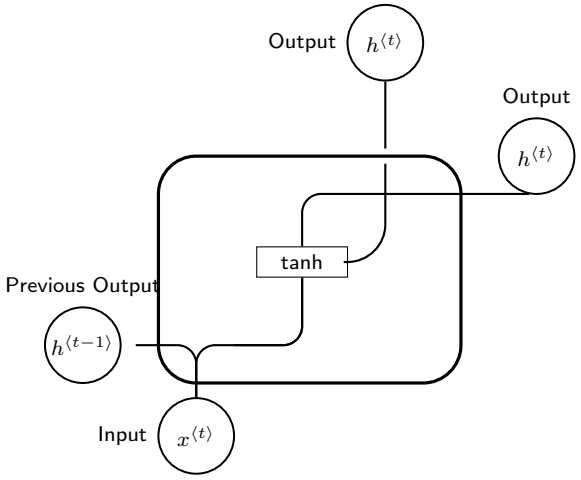


Figure 4.1: Repeating unit of an RNN (Shiri *et al.*, 2023).

There are three weight matrices in an RNN, the matrix containing the weights from the input layer to the hidden layer $\boldsymbol{W_{hx}} \in \mathbb{R}^{h \times x}$, the matrix containing the weights between hidden layers $\boldsymbol{W_{hh}} \in \mathbb{R}^{h \times h}$, and the matrix containing the weights from the last hidden layer to the output layer $\boldsymbol{W_{hy}} \in \mathbb{R}^{h \times y}$. The network takes in two inputs, the current input $\boldsymbol{x_t}$, which is a vector whose length is equal to the number of timesteps, and the previous hidden state h_{t-1}, which is the output from the hidden layers. The previous hidden state being passed to the next step in the network acts as the networks memory, as it makes use of previous data that the network is familiar with. The current hidden state now contains information on both the previous and current input, as it is a function of the previous hidden state and the current input, as shown in Equation 4.1.

$$h_t = f_{\boldsymbol{W}}(h_{t-1}, \boldsymbol{x_t}). \tag{4.1}$$

The hidden state at time t is given by:

$$h_t = f(\boldsymbol{W_{hx}} \boldsymbol{x_t} + \boldsymbol{W_{hh}} \boldsymbol{x_{t-1}}) + \boldsymbol{b_h}, \tag{4.2}$$

where h_t is the current hidden state, h_{t-1} is the previous hidden state, $\boldsymbol{x_t}$ is the input at time t, f is the activation function, $\boldsymbol{W_{hx}}$ is the weight between the input and hidden layer, $\boldsymbol{W_{hh}}$ is the weight between the hidden layers, and $\boldsymbol{b_h}$ is the bias in the hidden layer. Initially, at $t = 0$, there is no previous hidden state, thus h_0 is initially set to 0. The activation function which is used is the hyperbolic tangent function, which regulates the neural network by limiting values to the range of [-1,1] to prevent them from growing large. The resultant output becomes the next hidden state, and the process is repeated.

The final output is given by:

$$y_t = f(\boldsymbol{W_{yh}} h_t) + \boldsymbol{b_y}, \tag{4.3}$$

where $\boldsymbol{W_{yh}}$ is the weight matrix from the previous hidden layer to the output layer, and $\boldsymbol{b_y}$ is the bias vector in the output layer. This network is trained using gradient descent, for which there are numerous variations (Ruder, 2017).

4.2.1 Variations of Gradient Descent

As shown by an article written by Ruder (2017), there are many variations of the traditional gradient descent algorithm. These variations vary in the amount of data used to compute the gradient of the cost function for each model update. They consist of Batch Gradient Descent, which is the traditional gradient descent algorithm, Stochastic Gradient Descent, Mini Batch Gradient Descent and Gradient Descent with Momentum (Ruder, 2017).

In Batch Gradient Descent, the gradient of the cost function is computed across the entire training data, and the model is updated after all of the errors have been calculated for every observation in the training data (Ruder, 2017). This method is computationally efficient as it requires few machine cycles, and it converges easily (Ruder, 2017). However, it can be very slow and tends to use a lot of memory. It can also lead to redundant computations when dealing with large data, as the gradient will be recomputed for similar examples (Ruder, 2017). These drawbacks are addressed by the Stochastic Gradient Descent algorithm.

As opposed to Batch Gradient Descent, in which the model is only updated after gradients have been computed across the entire training data, Stochastic Gradient Descent updates the model after the gradient is computed for each training example, eliminating redundant computations for similar training examples (Ruder, 2017). This method is more computationally efficient than Batch Gradient Descent, due to memory efficiency and speed, as only one training example is used for each update (Ruder, 2017). Mini Batch Gradient Descent makes use of traditional gradient descent and Stochastic Gradient Descent, partitioning the training data into mini batches and computing the gradient over each small batch as opposed to a single training example, subsequently performing updates for each batch (Ruder, 2017). Computing the gradient over batches has a much better computational efficiency than Stochastic Gradient Descent, as processing a single training example at a time requires many computations. However, this method does not compute the actual derivative of the cost function, but rather an estimate on a small batch (Ruder, 2017). Momentum can be used to provide a closer estimate to the actual derivative (Ruder, 2017). Gradient Descent with Momentum typically uses an exponentially decaying moving average of previous gradients to update the model (Ruder, 2017). Gradient descent and its variations are essentially optimisers, algorithms which adjust training parameters in a network to minimise the cost function. Another commonly used optimiser is the Adaptive Moment Estimation (Adam).

4.2.2 Adaptive Moment Estimation Optimiser (Adam)

Adam is an optimiser which uses a combination of gradient descent and momentum based techniques to update the parameters in a network (Ruder, 2017). It works by iteratively adjusting the learning rate (Ruder, 2017). Similar to other gradient based learning methods, such as gradient descent and its variations, it uses backpropagation to compute its gradients, which are used to update the model parameters until they are optimal (Ruder, 2017).

Section 3.2.6 discusses the backpropagation algorithm for training feed-forward neural networks, which is only suitable for tasks which do not have a sequential aspect to them. Recurrent Neural Networks make use of the Backpropagation Through Time algorithm to compute gradients (Schmidt, 2019). Backpropagation Through Time is a variation of the traditional backpropagation algorithm, which enables the network to learn sequential data (Schmidt, 2019).

4.2.3 Backpropagation Through Time

Backpropagation Through Time works by unfolding the RNN over multiple timesteps, treating each timestep as a separate layer, and applying traditional backpropagation to compute the gradients (Schmidt, 2019). It then propagates the error back through the unfolded network to update the weights and biases. During backpropagation, the weights are continuously updated using gradient descent until the error is minimised (Schmidt, 2019). The gradients are computed as shown in Equations 4.4 - 4.7.

The loss is a function of the targets y and predictions $\hat{y}$. The overall loss function can be represented as a sum of the errors at time t, from t to T,

$$E(\hat{y}, y) = \sum_{t=1}^{T} E_t. \tag{4.4}$$

Let $\boldsymbol{W}$ be a weight matrix of the network. The derivative of the loss function with respect to $\boldsymbol{W}$, for T timesteps is given by:

$$\frac{\partial E}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial \boldsymbol{W}}. \tag{4.5}$$

Applying the chain rule to Equation 4.5 yields:

$$\frac{\partial E}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial E}{\partial \boldsymbol{y_t}} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \boldsymbol{W}}, \tag{4.6}$$

where $\frac{\partial h_t}{\partial h_k}$ is the derivative of the hidden state at time t with respect to a hidden state at some other time k. Taking a further look at this term yields the following:

$$\begin{aligned} \frac{\partial h_t}{\partial h_k} &= \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k-1}}{\partial h_k} \\ &= \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}. \end{aligned} \tag{4.7}$$

Equation 4.6 shows that the gradient of the weights is dependent on the derivative of the hidden state at time t with respect to a hidden state at some other time k, which is dependent on the previous hidden state at any time t, as shown in Equation 4.7. This can lead to problems with the gradient during backpropagation, such as vanishing or exploding gradients (Schmidt, 2019). This is a problem in which the gradient tends to shrink, or explode, when training a deep neural network which utilises gradient based learning algorithms, for instance, during backpropagation of a traditional RNN (Schmidt, 2019).

Case 1: if $h_{t-1} < 1$,

$\frac{\partial E}{\partial \boldsymbol{W}}$ will decrease during backpropagation, until it approaches 0, causing the gradient to vanish.

Case 2: if $h_{t-1} > 1$,

$\frac{\partial E}{\partial \boldsymbol{W}}$ will increase during backpropagation, until it approaches infinity, causing the gradient to explode.

Small gradients have an insignificant contribution to learning, making the network inefficient and resulting in underfitting, while large gradients make learning highly unstable, resulting in overfitting (Schmidt, 2019). The activation function used in each hidden layer determines the behaviour of the gradient, the sigmoid function can lead to a vanishing gradient (Hu Zheng and Yun, 2021) while the ReLu function can lead to an exploding gradient (Hu Zheng and Yun, 2021).

In conclusion, RNNs are advantageous in that they factor in previous data in decision making, the size of the input data does not cause the size of the model to increase, and inputs of any length can be used (Schmidt, 2019). However, RNNs are difficult to train and can be slow. They also tend to break down when the desired output from the data being modelled is dependent on input data that is too far back, making them inefficient at handling problems with long term dependencies (Schmidt, 2019). A solution to the long term dependency problem experienced by RNNs was the introduction of Long Short-Term Memory Networks (LSTM) (Schmidt, 2019).

4.3 LONG SHORT-TERM MEMORY NETWORKS

LSTMs are a special type of RNN that were brought about to address the vanishing gradient problem of its predecessors, to enable them to handle long term dependencies (Schmidt, 2019). LSTMs have the same control flow as traditional RNNs, data is passed through the network sequentially as the network propagates forward, however LSTMs have structures in place to keep or forget information depending on relevancy (Schmidt, 2019). They comprise of cell states, which carry information through the network thus minimising the effects of short-term memory. Structures called gates modify the information contained in the cell state (Schmidt, 2019). Gates are neural networks that are trained to learn which information should be kept or discarded, they make use of the sigmoid activation function which compresses outputs between the range of [0,1], with values near 0 meaning discard, and values near 1 meaning keep (Schmidt, 2019). This architecture is shown in Figure 4.2, where $x^{(t)}$, $c^{(t)}$, and $h^{(t)}$ are the current input, cell state and hidden state respectively. Furthermore, $c^{(t-1)}$ and $h^{(t-1)}$ are the previous cell state and hidden state respectively.

4.3.1 Architecture

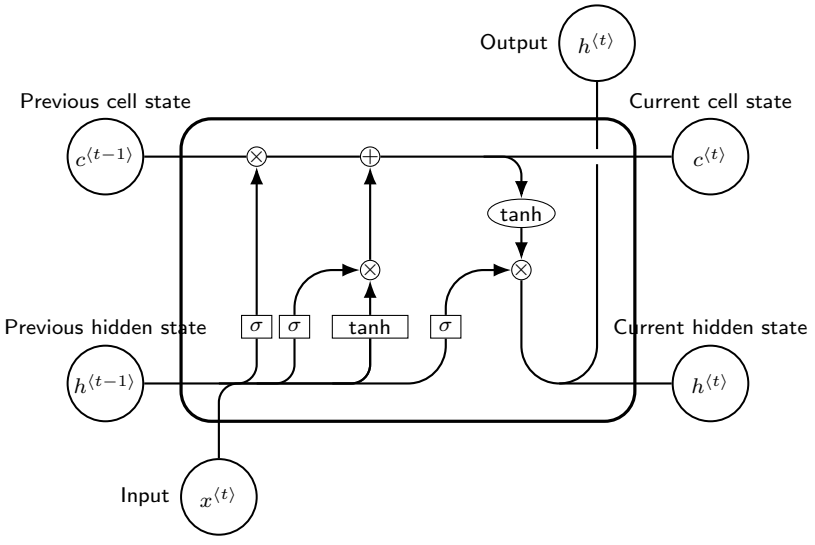


Figure 4.2: Repeating unit of an LSTM (Shiri *et al.*, 2023).

There are three types of gates which are used to regulate the flow of information within each cell, the input gate, forget gate and output gate (Schmidt, 2019; Staudemeyer and Morris, 2019).

1. Input Gate

   This gate decides which information from the input should be permitted to modify memory (Schmidt, 2019; Staudemeyer and Morris, 2019). It updates the cell state by using information from the current input x_t, and the previous hidden state h_{t-1}. This undergoes sigmoid activation which compresses the values between 0 and 1, giving an indication of which values should be updated, with values near 1 being significant and values near 0 being insignificant, essentially deciding which values to let through (Staudemeyer and Morris, 2019). This process is captured by Equation 4.8.

   The input gate is given as:

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i), \tag{4.8}$$

   where i_t is the value of the input gate, w_i is the weight for the input gate, h_{t-1} is the previous hidden state, x_t is the current input, and b_i is the bias for the input gate.

   To regulate the network, the previous hidden state h_{t-1} and current input x_t are passed through the hyperbolic tangent function to transform the values to fall within the range of [-1,1], giving the values weightage, which decides their level of importance (Schmidt, 2019; Staudemeyer and Morris, 2019). This new output is called a candidate cell state, the process outlined is depicted in Equation 4.9.

$$\tilde{c_t} = tanh(w_c[h_{t-1}, x_t] + b_c). \tag{4.9}$$

   The cell state is calculated by performing pointwise multiplication of the previous cell state c_{t-1} and the output from the forget gate f_t (Schmidt, 2019; Staudemeyer and Morris, 2019).

Information from the forget gate is discarded or kept, which is added in a pointwise manner the pointwise product of the current cell state c_t, and the output of the input gate i_t (Schmidt, 2019; Staudemeyer and Morris, 2019). This process is depicted in Equation 4.10

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t. \tag{4.10}$$

2. Forget Gate

   This gate decides which information should be kept or discarded (Schmidt, 2019; Staudemeyer and Morris, 2019). According to (Staudemeyer and Morris, 2019), information from the previous hidden state and the current input is processed by the sigmoid function resulting in an output ranging between 0 and 1, indicating what should be kept or what should be forgotten. This is shown in Equation 4.11.

$$f_t = \sigma(w_i[h_{t-1}, x_t] + b_i). \tag{4.11}$$

3. Output gate

   This gate decides the next hidden state o_t, which contains information about the previous inputs and is able to make predictions (Staudemeyer and Morris, 2019; Schmidt, 2019). Information from the previous hidden state and current input are passed through the sigmoid activation function, this is shown in Equation 4.12.

$$o_t = \sigma(w_0[h_{t-1}, x_t] + b_0), \tag{4.12}$$

   where w_o and and b_o is the weight and bias at the output layer respectively. Equation 4.12 results in the new hidden state – which, along with the new cell state (Equation 4.10, is passed onto the next step in the network, and the process is repeated.

LSTMs come in four main types, each categorised by the shape of their input and output data. Namely, many-many, many-one, one-one, and one-many. This is discussed further in Section 4.3.2. Section 4.3.1 has outlined the traditional architecture of an LSTM model. There are however many variations, each suitable for different tasks, as shown by a study conducted by Murugesan *et al.* (2021). This is discussed in Section 4.3.3.

4.3.2 Types of LSTMs

Let $x = (x_0, x_1, \cdots, x_n)$ be the set of inputs of an LSTM, and $y = (y_0, y_1, \cdots, y_m)$ the set of outputs. A many-many LSTM maps multiple inputs to multiple outputs (Zelios *et al.*, 2022). It takes in a sequence for each input and outputs the corresponding sequence, $(x_0, x_1, \cdots, x_t) \rightarrow (y_0, y_1, \cdots, y_k)$, where t is the timestep, and k is the desired length of the output sequence. This is particularly useful in machine translation when translating a sentence from one language to another (Zelios *et al.*, 2022). According to a paper by Zelios *et al.* (2022), which examines RNNs, a many-one LSTM maps multiple inputs to one output. It takes in a sequence for each input and maps it to a single value $(x_0, x_1, \cdots, x_t) \rightarrow (y_j)$. An example of this is in sentiment classification where text reviews are converted into ratings (Zelios *et al.*, 2022). The paper also highlights another type of LSTM is a one-one LSTM, which takes in a single value and maps it to another single value $(x_i) \rightarrow (y_j)$, (Zelios *et al.*, 2022). The last type is a one-many LSTM, it takes in a single value and maps it to a sequence of values $(x_i) \rightarrow (y_0, y_1, \cdots, y_k)$, (Zelios *et al.*, 2022).

4.3.3 Variations of LSTM Models

There numerous variations of LSTM models which each work well for different types of sequential problems, namely, the classic LSTM (Murugesan *et al.*, 2021), stacked LSTM (Murugesan *et al.*, 2021), and Bi-directional LSTM (Murugesan *et al.*, 2021). The classic LSTM is the traditional LSTM model described in section 2.3.1, otherwise known as the vanilla LSTM. It consists of one LSTM layer with 4 gating layers; two input gates, a forget gate and an output gate (Murugesan *et al.*, 2021). Classic or vanilla LSTMs compromise of hidden states which are dependent on previous hidden states, making them deep in time, a stacked LSTM consists of multiple LSTM layers, making it deep in space (Murugesan *et al.*, 2021). The hidden states of all of the layers are computed recursively starting from the first layer, in the first layer h_t is replaced by x_t since there is no previous layer to obtain a hidden state from. Traditional LSTMs only analyse data in one direction, data from the past, Bi-directional LSTMs analyse data from the past and from the future (Murugesan *et al.*, 2021). The input is fed into two separate neural networks, one going forward and one going backwards, which are both connected to one output layer. Bi-directional LSTMs are useful when dealing with problems where both past and future information in a sequence influence the current prediction (Murugesan *et al.*, 2021).

4.3.4 Advantages and Disadvantages of LSTMs

LSTMs are advantageous in that they are able to handle long term dependencies, remembering relevant information over multiple timesteps (Xing *et al.*, 2022). They are also suitable for a wide variety of sequence prediction tasks (Schmidt, 2019). These factors make them particularly useful within this research, given the sequential nature of the COCOON data. Additionally, smoothing techniques can be employed to optimise the network for even better performance. However, LSTMs can be computationally expensive due to increased model complexity compared to traditional neural networks (Xing *et al.*, 2022). Large datasets further exacerbate this problem as they require more computational resources, such as memory and processing power, resulting in slower training (Xing *et al.*, 2022). Training large data with LSTMs is beneficial since it can reduce overfitting, further improving the model's accuracy. However, it is important to find a balance between the size of the data and the computational efficiency of the model.

4.4 FURTHER BACKGROUND STUDY ON MACHINE LEARNING TECHNIQUES

4.4.1 Error Correction Neural Networks

In a dynamical system, the external variables which give an indication of the underlying relationships in the data are not always known, the Error Correction Neural Network (ECNN) was developed to address this limitation (Mvubu *et al.*, 2020). An Error Correction Neural Network is a type of Recurrent Neural Network, designed to improve forecasting accuracy in various domains, such as economic forecasting (Mvubu *et al.*, 2020). However, it differs from a traditional neural network in that it feeds the difference between the prediction made in the previous timestep and it's corresponding target, back into itself. This error contains information about the missing external variables (Nandutu *et al.*, 2022; Mvubu *et al.*, 2020). The error correction mechanism is incorporated into the training process of the neural network, allowing the network to learn from the errors it makes during forecasting, and adjust its future predictions accordingly (Mvubu *et al.*, 2020).

By continuously updating the model's predictions based on the observed errors, ECNNs aim to converge to more accurate and reliable forecasts. The error correction process can occur at different stages in the network, such as during the training phase or during the prediction phase. This is comparable to the Autoregressive Integrated Moving Average model (ARIMA), however it is applicable to non-linear dynamical systems (Mvubu *et al.*, 2020). Figure 4.3 shows an unfolded ECNN.

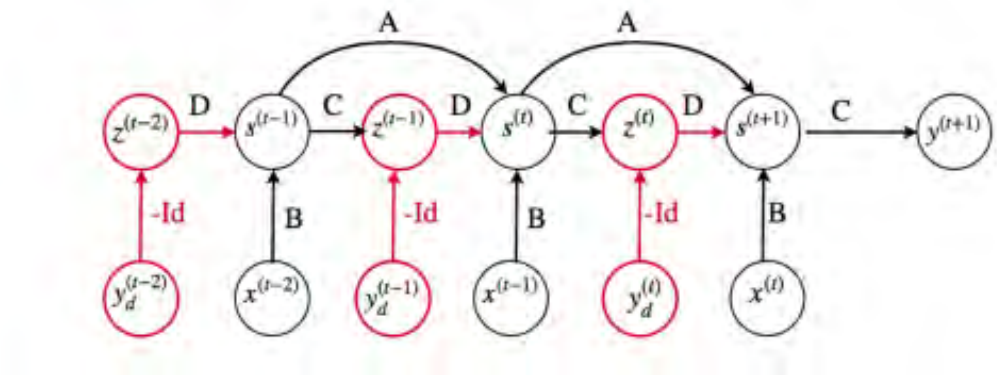


Figure 4.3: Unfolded Error Correction Neural Network (Mvubu *et al.*, 2020).

Figure 4.3, **A B C and D** represent the weight matrices between the different layers. **A** is the weight matrix between the hidden layers, **B** is the weight matrix from the input layer to the hidden layer, and **C** is the weight matrix from the hidden layer to the output layer. z_t is the error at time t, x_t is the input at time t, s_t represents the cell state at time t and y_{t+1} is the output of the network. **-Id** is the identity matrix multiplied by -1, as shown Figure 4.13.

$$-Id = \begin{bmatrix} -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & \cdots & -1 \end{bmatrix}. \tag{4.13}$$

The equations governing an ECNN are as follows:

The error at time t is given as:

$$z^{(t)} = y^{(t)} - y_d^{(t)}, \tag{4.14}$$

where $y^{(t)}$ is the predicted value, and $y_d^{(t)}$ the target value. The cell state at time t is given as:

$$s^{(t)} = tanh(\boldsymbol{A}s^{(t-1)} + \boldsymbol{B}x^{(t-1)} + \boldsymbol{D}tanh(z^{t-1})), \tag{4.15}$$

where x_{t-1} and z_{t-1} are the input and error respectively at time $t-1$. Finally the output at time t is given by:

$$y^{(t)} = \boldsymbol{C}s^{(t)}, \tag{4.16}$$

where $\mathbf{C}$ is the weight matrix from the hidden layer to the output layer, and $s^{(t)}$ is the cell state given by Equation 4.15. The network is trained via Backpropagation Through Time (Mvubu *et al.*, 2020). During backpropagation, the weights are continuously updated using gradient descent until the error is minimised, to improve performance.

4.4.2 Advantages and Disadvantages of ECNNs

Due to their internal error correction mechanism, ECNNs tend to have high accuracy and adaptability (Mvubu *et al.*, 2020). This allows them to adapt relatively quickly to changing data distributions, which is of particular interest within this research as we would like the models to learn the distribution of the Difference data. Techniques such as exponential smoothing can be employed to optimise the network for even better performance. However, ECNNs tend to be more complex and require more data than traditional neural networks (Hu *et al.*, 2021).

4.4.3 Exponential Smoothing

Exponential smoothing is a smoothing technique developed in the late 1950's for forecasting (univariate) time series data (Mvubu *et al.*, 2020). It was initially put forward by Holt (1957), Brown (1959) and Winter (1960).

Simple exponential smoothing is used when the time series data to be forecast has no apparent patterns (Ostertagova and Ostertag, 2012). It is one of the most commonly used smoothing techniques, due to it's efficiency, simplicity and accuracy (Ostertagova and Ostertag, 2012). The original time series data is smoothed using an exponential weighted average, and the smoothed dataset is subsequently used to make predictions. Each observation is weighted according to how recent it is at that timestep, with the current observation obtaining the largest weight. The weightage decays exponentially for historic data (Ostertagova and Ostertag, 2012).

Let Y_t represent a time series, the Simple Exponential Smoothing Equation is given by:

$$\hat{Y}_{i+1} = \alpha Y_i + (1-\alpha)\hat{Y}_i, \tag{4.17}$$

where $\hat{Y}_{i+1}$ is the predicted value at time $i+1$, Y_i the actual value at time i, and $\hat{Y}_i$ the predicted value at time i. The most recent observation Y_i is weighted using the smoothing parameter α, and the most recent prediction $\hat{Y}_i$ is weighted by $(1-\alpha)$, where $0 < \alpha < 1$.

Using Equation 4.17, we get the following:

$$\hat{Y}_{i+1} - \hat{Y}_i = \alpha(Y_i - \hat{Y}_i). \tag{4.18}$$

From this, we can see that the change in successive predictions is directly proportional to the prediction error. Therefore the prediction made from exponential smoothing, is the previous prediction adjusted by the error from the previous timestep.

If we take the Simple Exponential Smoothing Equation (Equation 4.17), and continuously substitute in the previous predictions $\hat{Y}_i$, $\hat{Y}_{i-1}$, $\hat{Y}_{i-2} \cdots$, we get the following:

$$\begin{aligned}
\hat{Y}_{i+1} &= \alpha Y_i + (1-\alpha)[\alpha Y_{i-1} + (1-\alpha)\hat{Y}_{i-1}] \\
&= \alpha Y_i + \alpha(1-\alpha)Y_{i-1} + (1-\alpha)^2\hat{Y}_{i-1} \\
&= \alpha Y_i + \alpha(1-\alpha)Y_{i-1} + (1-\alpha)^2 Y_{i-2} + (1-\alpha)^3\hat{Y}_{i-2} \\
&= \alpha Y_i + \alpha(1-\alpha)Y_{i-1} + (1-\alpha)^2 Y_{i-2} + (1-\alpha)^3 Y_{i-3} + (1-\alpha)^4\hat{Y}_{i-3} \\
&\dots
\end{aligned}$$

From this, we are able to observe a pattern, and thus, we can obtain the general form of the forecast equation.

The general form of the forecast equation is given by:

$$\hat{Y}_{i+1} = \alpha \sum_{k=0}^{i-1} (1-\alpha)^k Y_{i-k}, \tag{4.19}$$

where $\hat{Y}_{i+1}$ is the predicted value at time $i+1$, Y_i the actual value at time $i-k$, and α is a smoothing parameter: $0 < \alpha < 1$.

Simple exponential smoothing does not take into consideration any seasonality or trends in the data, the Holt-Winters exponential smoothing method addresses this limitation.

Let X_t represent a monthly seasonal time series and assume an additive decomposition model in the form:

$$X_t = \widetilde{L_t} + \widetilde{S}_t + \epsilon_t, \tag{4.20}$$

where $\widetilde{L_t}$ is the level of the series at month t, $\widetilde{S}_t$ is the seasonal effect at time t, and ϵ_t is the irregular component. The level of series at time t is the average value of the series at that time

(Pfeffermann and Allon, 1989). Since the data is seasonal it has seasonal fluctuations, the seasonal effect is a measure of those fluctuations (Pfeffermann and Allon, 1989). The irregular component is some random disturbance in the data which we can think of as noise, from the error in the model (Pfeffermann and Allon, 1989). At any given time, the level gives an estimate of the weighted average, the trend gives an estimate of the change between consecutive points, and the seasonality gives an estimate of the deviation from the local mean caused by seasonal effects. There are 3 main updating equations (level, trend, seasonality) each of these have their own smoothing parameters.

The updating equations are:

$$L_t = \alpha(X_{t+1} - S_t^{t+1}) + (1 - \alpha)(L_t + T_t), \tag{4.21}$$

where L_t is the level of the series at month t, α is the smoothing parameter, X_{t+1} is a monthly seasonal time series at month $t + 1$, S_t^{t+1} is the one step ahead seasonal effect at time t, and T_t is the trend of the series at time t. In Equation 4.21, the seasonality factor is subtracted from time series, which has the effect of deseasonalising the new observation, the previous level is updated by adding the trend estimate and combining it with a deseasonalised value of the time series.

The previous trend is updated by considering the latest difference between levels:

$$T_{t+1} = \gamma(L_{t+1} - L_t) + (1 - \gamma)T_t, \tag{4.22}$$

where T_t and T_{t+1} are the trends at time t and $t + 1$ respectively, and L_t and L_{t+1} are the levels at time t and $t + 1$ respectively, and γ is the smoothing parameter.

$$S_{t+1}^{*t+1} = \delta(X_{t+1} - L_{t+1}) + (1 - \delta)S_t^{t+1} \tag{4.23}$$

$$\sum_{i=0}^{11} S_{t+1}^{t+1+i} = 0, \tag{4.24}$$

where the last two equations are a two step procedure to compute the seasonal effects. First, the initial value of S is computed, then it is used along with the 11 other seasonal effects at time t (for a monthly time series) and standardized so that their sum is 0 (Pfeffermann and Allon, 1989).

The forecast at time t is given by X_{t+m}:

$$\hat{X}_t(m) = L_t + mT + S_t^{t+m}. \tag{4.25}$$

The series contains level, trend, seasonality and noise, which are combined in the forecasting equation in an additive model. Addition is used to add the seasonal component, implying that for different seasons forecasts are different by a fixed amount. This process is univariate, it only uses one time series variable to predict future values, whereas multivariate exponential smoothing uses multiple related time series variables to make predictions. A multivariate generalisation is shown below.

Consider a series fluctuating around a constant level, according to Pfeffermann and Allon Pfeffermann and Allon (1989) the one step forecast is:

$$\boldsymbol{x}_t(1) = \boldsymbol{A_0 x}_t + \boldsymbol{A_1 x}_{t-1} + \boldsymbol{A_2 x}_{t-2} + \cdots,$$

where $\boldsymbol{A_i}$ are k x k matrices of decaying weights, and $\boldsymbol{x}_{t-k}$ is a time series at time $t-k$, where $k = 0, 1, 2, 3 \cdots$.

Considering a series with trends and seasonal effects:

$$\boldsymbol{x}_t = \boldsymbol{l}_t + \boldsymbol{s}_t + \boldsymbol{\epsilon}_t, \tag{4.26}$$

where $\boldsymbol{x}_t$ is a vector containing time series' at time t, $\boldsymbol{l}$ is the vector containing levels, $\boldsymbol{s}_t$ is the vector containing seasonal effects, and $\boldsymbol{\epsilon}_t$ is the vector containing irregular components.

The updating equations are:

$$\boldsymbol{l}_{t+1} = \boldsymbol{A}(\boldsymbol{x}_{t+1} - \boldsymbol{s}_t^{t+1}) + (1 - \boldsymbol{A})(\boldsymbol{l_t} + \boldsymbol{t}_t), \tag{4.27}$$

where l_{t+1} is the trend vector at time $t+1$, $\boldsymbol{A}$ is a square matrix of decaying weights, $\boldsymbol{s}_t^{t+1}$ is the seasonality vector at time t for a one step ahead forecast, $\boldsymbol{l_t}$ and $\boldsymbol{t}_t$ are the level and trend vectors at time t respectively.

$$\boldsymbol{t}_{t+1} = \boldsymbol{\Gamma}(\boldsymbol{l}_{t+1} - \boldsymbol{l}_t) + (I - \Gamma)\boldsymbol{t}_t \tag{4.28}$$

$$\boldsymbol{s}_{t+1}^{*t+1} = \Delta(\boldsymbol{x}_{t+1} - \boldsymbol{l}_{t+1}) + (I - A)\boldsymbol{s}_t^{t+1} \tag{4.29}$$

$$\sum_{i=0}^{11} \boldsymbol{s}_{t+1}^{t+1+i} = \boldsymbol{0}. \tag{4.30}$$

The m step ahead forecast at time t is now:

$$\boldsymbol{x}_t(m) = \boldsymbol{l}_t + m\boldsymbol{t}_t + \boldsymbol{s}_t^{t+m}, \tag{4.31}$$

where $\boldsymbol{\Gamma}$ and Δ are smoothing matrices, $\boldsymbol{x}_t(m)$ is the m step ahead forecast of the time series at time t, $\boldsymbol{l}_t$ and $\boldsymbol{t}_t$ are the level and trend vectors at time t respectively, and $\boldsymbol{s}_t^{t+m}$ is the m step ahead seasonality vector forecast at time t.

4.5 CONCLUSION

In conclusion, this chapter provided an extensive review of Recurrent Neural Networks and related machine learning techniques. The foundations of traditional RNNs were established in Section 4.2, revealing that RNNs are a type of neural network specifically designed to handle sequential data. An

exploration of variations of gradient-based learning algorithms such as Stochastic Gradient Descent and the Adaptive Moment estimation optimizer in Section 4.2.1 - 4.2.2 respectively. Revealing that Mini-Batch and Stochastic Gradient Descent are more computationally efficient but may converge slower or have less accurate updates, whereas Gradient Descent with Momentum accelerates convergence but at a higher computational cost. The challenges of training RNNs with gradient-based learning methods, including the vanishing or exploding gradient problem, were discussed in Section 4.2.3 through the introduction of Backpropagation Through Time. To address this limitation, the chapter delved into the Long Short-Term Memory Network, a more advanced type of RNN, in Section 4.3, with built in 'memory'. Additionally, Error Correction Neural Networks, a sophisticated type of RNN capable of self correcting, were examined in Section 4.4.1. This was coupled with an exploration of several exponential smoothing techniques for optimization in Section 4.4.3, which have the ability to improve the forecasting ability of a model. By covering these topics, this chapter offers a comprehensive understanding of RNNs, their variations, and their applications in machine learning. Setting a solid foundation for effectively choosing and implementing machine learning algorithms to make predictions on the COCOON data.

CHAPTER 5

RESEARCH METHODOLOGY

5.1 INTRODUCTION

This chapter describes the approaches and techniques that were used to build machine learning models to make predictions on COCOON data. We recap that the research thrust as introduced in Chapter 2 consists of the following: exploring various implementations of Recurrent Neural Networks for predicting the $(n+1)^th$ COCOON, given any list of n COCOONs. Additionally, comparing and refining the accuracy and computational efficiency of the two main algorithms of interest. Section 5.2 outlines the prediction model framework which was followed, with Section 5.3 showing the different datasets which were used, clarifying the preprocessing of the data into the model input. The experimental design is discussed in Section 5.4, including the hyperparameter tuning technique which was used. Finally, Section 5.5 discusses how the models' accuracy and computational efficiency was evaluated.

5.2 PREDICTION MODEL FRAMEWORK

Figure 5.1 illustrates the framework followed for developing the prediction models. The initial step involved data collection, where a COCOON dataset was obtained. Subsequently, the difference dataset was created by iteratively calculating the sequential difference between COCOON pairs. To simplify the data, each entry in the difference dataset was divided by a factor of 2. Both the difference and COCOON datasets were then transformed into input sequences with corresponding outputs.

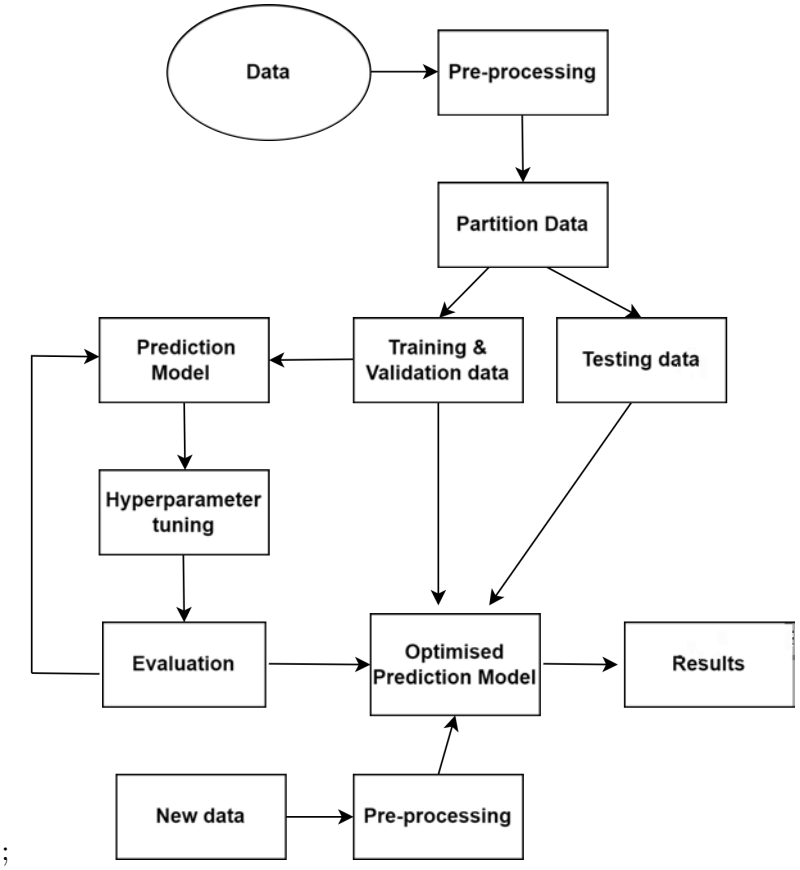
Two distinct pre-processing methods were used to pre-process the data, based on the model. In the case of models employing exponential smoothing, the data underwent exponential smoothing prior to being fed to the model. For the remaining models, the data was pre-processed using the min-max scaler technique. This normalisation technique scales the data between 0 and 1, preventing any single feature from overpowering the model.

Next, the data was partitioned into training, validation, and testing sets, using a split ratio of 70 : 15 : 15 respectively. Model selection was based on a literature review conducted in Chapter

3 and 4, which motivated the use of Long Short-Term Memory Networks and Error Correction Neural Networks due to the sequential nature of the data. Variations of these two main algorithms, specifically stacked LSTM and ECNN with exponential smoothing, were also tested.

Hyperparameter tuning was performed through a grid search to optimise parameters such as batch size, number of epochs, number of neurons, timestep, and dropout rate. The learning rate was automatically adjusted using the Adam optimiser. Each combination of hyperparameters was evaluated using the Root Mean Square Error. The hyperparameter combination that yielded the lowest RMSE was selected as the optimised model.

The optimised model was then tested on new data, and the results were evaluated using appropriate metrics such as RMSE, MAPE, Theil U coefficient, and directional accuracy. To obtain the final results from the models, the predictions were scaled back to their original representation, to account for the initial scaling which was done during pre-processing.



;

Figure 5.1: Machine Learning Prediction Model Framework.

5.3 DATASET

The dataset required for the research is the list of COCOONs, which was generated from Equations 2.16- 2.26 using a threshold of 404 364. An example of a small dataset of COCOONs given a threshold of 1000 is shown in Table 2.1. Another dataset can also be obtained from the COCOON dataset, a dataset of differences, consisting of the normalised differences of COCOON pairs.

Difference data= $\{d_i : i \in \mathbb{N}\}$, let N_{i+1}, N_i be COCOONs, then:

$$d_i = \frac{N_{i+1} - N_i}{2}. \tag{5.1}$$

An example of such a dataset is shown in Figure 5.1,

3	3	2	1	3	1	2	3	2	1	2	1	3	1	2	3	1	2	2	1	2	1	1	2	3	3	2	1	1	1	1	1	2	2	1	3	1	1	1	3	1	2	1	2	2	1	2	1
3	1	1	1	3	3	1	1	1	1	2	1	1	1	1	2	3	2	1	3	1	1	1	2	1	2	1	2	1	3	1	2	3	1	1	1	2	1	1	1	1	1	2	3	2	1	1	1
1	1	2	1	2	1	1	1	3	2	1	2	1	1	2	1	2	1	2	2	1	2	1	1	2	2	1	1	2	1	1	1	3	1	1	1	3	1	2	2	1	2	1	1	2	3	2	1
1	1	1	2	1	1	2	2	1	1	2	2	1	2	1	1	1	1	3	1	1	1	1	1	1	1	2	1	2	1	1	1	2	1	2	1	3	1	2	1	1	1	2	1	2	1	3	1
2	1	2	3	1	1	1	1	2	1	1	1	3	2	1	2	1	3	1	1	1	1	2	2	1	2	1	1	2	1	1	1	2	1	1	2	1	1	1	2	1	1	2	1	2	1	2	2
1	1	2	1	2	2	1	1	2	2	1	1	1	1	1	2	1	1	1	2	1	1	1	1	3	1	1	1	3	3	1	1	1	2	1	1	1	1	1	2	3	2	1	1	1	1	1	2
3	2	1	1	1	1	1	1	1	1	2	2	1	1	2	1	1	1	2	1	1	2	2	1	2	1	2	1	1	1	1	1	2	2	1	2	1	2	1	1	2	1	2	1	1	1	1	2
2	1	3	1	1	1	3	1	2	1	1	1	3	1	1	1	3	1	2	1	1	1	1	1	1	1	2	3	2	1	2	1	2	1	1	2	1	2	1	2	1	1	1	1	1	1	1	1
1	3	1	1	1	2	1	1	2	1	1	1	2	1	2	1	2	1	1	2	1	1	1	1	2	2	1	2	1	3	1	2	1	1	1	1	2	1	1	1	2	1	1	1	1	1	1	1
3	2	1	3	1	2	3	2	1	1	1	1	3	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	2	1	2	1	1	2	1	1	1	1	1	1	1	2	1	1
1	2	1	1	1	1	1	2	3	2	1	2	1	1	2	3	1	2	1	1	1	1	2	1	1	1	3	3	2	1	2	1	1	1	1	2	1	1	1	1	2	1	1	2	1	1	1	1
2	1	2	2	1	2	1	1	2	2	1	1	2	2	1	1	1	1	1	2	2	1	3	2	1	3	1	2	1	1	1	2	1	1	1	1	1	1	1	1	2	1	2	3	1	1	1	1
1	1	1	1	1	1	2	3	1	1	1	2	1	1	1	1	3	1	1	1	2	1	1	2	1	2	1	2	1	2	1	1	1	1	1	1	1	2	1	2	3	1	1	1	1	2	1	1
1	2	1	1	1	1	2	1	1	2	1	1	1	1	1	1	2	1	1	1	1	1	2	1	2	3	3	1	1	1	2	1	1	1	1	2	1	2	1	2	1	1	1	1	1	1	1	3
1	1	1	1	1	1	3	1	1	1	1	1	1	1	1	1	2	1	2	1	1	2	1	2	3	2	1	1	2	1	2	1	1	1	3	1	1	1	2	1	1	1	1	1	2	1	1	1
2	1	2	1	1	1	1	3	3	1	1	1	2	1	1	1	1	3	1	1	1	1	1	1	3	1	2	3	1	1	1	2	1	1	2	1	1	1	1	1	1	1	2	1	1	1	1	1
1	1	1	1	1	2	2	1	2	1	1	2	1	1	1	1	1	1	3	2	1	1	2	1	1	1	1	1	1	3	2	1	1	2	1	2	1	2	2	1	1	1	1	3	1	1	1	1
1	1	1	1	1	2	1	3	1	2	2	1	2	1	1	1	1	1	2	1	2	3	1	2	2	1	2	1	1	1	1	2	1	2	1	2	1	1	1	1	1	2	2	1	2	1	1	1
1	2	1	1	2	1	2	2	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	2	1	1	1	2	1	1	2	2	1	3	1	2	1	1	1	1	1	1	1	1
1	2	1	1	1	1	3	1	1	1	1	1	1	2	1	1	2	2	1	1	1	1	2	1	1	1	1	1	2	1	1	1	3	2	1	1	2	1	2	1	2	3	2	1	2	1	1	2
3	1	2	1	1	1	3	1	1	1	2	1	1	2	2	1	1	1	1	1	2	1	1	1	2	1	1	1	1	3	1	2	3	1	1	1	1	1	1									

Table 5.1: An example of a Difference dataset for COCOONs below 1000.

The initial data consists of a single array of COCOONs, which is used to create separate multi-dimensional arrays for the input and output data. In order to form the input data, the data was partitioned into sequences of equal length, this length is known as the timestep, and was determined through experimentation for each model. Each input is a small sequence of COCOONs, which is then stored in an array containing all of the other small sequences.

As a result, the input data becomes a two-dimensional array. To create the output data, the next COCOON in the series, following each small sequence, is stored in its own array. As a result, each small sequence is associated with a single value representing the subsequent COCOON in the sequence.

Sequences of COCOONs are used as inputs to enable the model to learn the internal structure of the data, each sequence is then used to predict a single output value, this is a many-to-one model.

The first 3 inputs are shown below:

1. $(x_0, x_1, \cdots, x_t) \rightarrow (x_{t+1})$
2. $(x_{t+1}, x_{t+2}, \cdots, x_{2t}) \rightarrow (x_{2t+1})$
3. $(x_{2t+1}, x_{2t+2}, \cdots, x_{3t}) \rightarrow (x_{3t+1})$, where t is the timestep.

For example, choosing a time step of 3 in the dataset shown in Table 2.1 yields:

$$\text{Input Sequence} \rightarrow \text{Single Output Value}$$

$$(9, 15, 21) \rightarrow (25)$$

$$(27, 33, 35) \rightarrow (39)$$

$$(39, 45, 49) \rightarrow (51)$$

$$\vdots$$

$$(x_{n-3}, x_{n-2}, x_{n-1}) \rightarrow (x_n),$$

$$\text{where } n = \text{ length of the dataset } - 1.$$

5.3.1 Model Selection

As previously discussed, the choice of models was made after reviewing previous research in Chapter 3 and 4. The nature of the data being sequential led to the selection of the LSTM and ECNN. Additionally, variations of these main algorithms, involving stacking hidden layers and incorporating exponential smoothing, were also experimented with and evaluated. The architecture of the models was chosen by experimentation, during which the hyperparameters were tuned.

5.3.2 Hyperparameter Tuning

Hyperparameter tuning is the process in which the parameters involved in training a learning algorithm are optimized, one such method of hyperparameter tuning is the Grid Search (Ensor and Glynn, 1997). In this method, the hyperparameter space is defined by assigning a finite range of values for each hyperparameter (Ensor and Glynn, 1997), forming a grid with all of the hyperparameters $A \subseteq \mathbb{R}$ where $A_m = \{\theta_1, \theta_2, \cdots, \theta_m\}$. A model is then built for every single possible combination of the hyperparameter values, each model is then evaluated using an appropriate metric $\alpha : A \to \mathbb{R}$ for each $\theta \in A$, and the hyperparameter combination which maximises α is chosen (Ensor and Glynn, 1997). This method has its advantages in that it is thorough, however because it is exhaustive it is very time consuming and can require a large amount of computational resources, there is also risk of overfitting. Overfitting can be tackled by introducing a drop out layer. Dropout is a regularization method often used in feed-forward neural networks, a dropout rate is specified which gives the percentage of connections to randomly drop during training, this is done so that each hidden unit learns using a randomly chosen sample of other units, helping the model learn to generalise. The research done on the use of a dropout layer in RNNs is very limited, hence the effect of dropout on the performance of RNNs is not well known, because of this — an LSTM with no dropout was also constructed during hyperparameter tuning and compared to iterations with varying dropout. The hyperparameters we set out to optimise for the LSTM are: number of epochs, number of neurons, batch size, drop out rate, and time step.

5.4 EXPERIMENTAL DESIGN

This section presents the experimental design used to develop the machine learning algorithms which were selected for the research. Additionally, it outlines the performance metrics which were used to measure the effectiveness of the models.

5.4.1 Long Short-Term Memory Network

The initial LSTM model was constructed consisting of a single LSTM layer (vanilla LSTM), a dropout layer which randomly drops a specified percentage of the connections to prevent overfitting, and a fully connected layer which connects each neuron in layer N to all of the neurons in the next layer, $N + 1$. Grid search was used to obtain the optimum hyperparameters for the model.

The input data for this model was pre-processed using the min-max scaler function, after this data was passed into the model, the predictions were subsequently scaled back to their original representation using the scaler inverse transform. The architecture of the LSTM model is shown in Table 5.2.

Table 5.2: Architecture of the LSTM model.

timestep	**batch size**	**neurons**	**epochs**
17	2	50	200

5.4.2 Long Short-Term Memory Network with Exponential Smoothing

This model has the same architecture as the model in Section 5.4.1, however, the input data was exponentially smoothed before being passed into the model. This was done in order to reduce fluctuations in the data, in order to make it easier for the model to learn underlying patterns existing within the data. This process in which this was done is detailed in Section 5.4.5.

5.4.3 Stacked Long-Short Term Memory Network

A stacked LSTM was also constructed using multiple LSTM layers, the architecture is shown in Figure 5.2. The number of layers, neurons in each layer, batch size, epochs and timesteps were obtained experimentally.

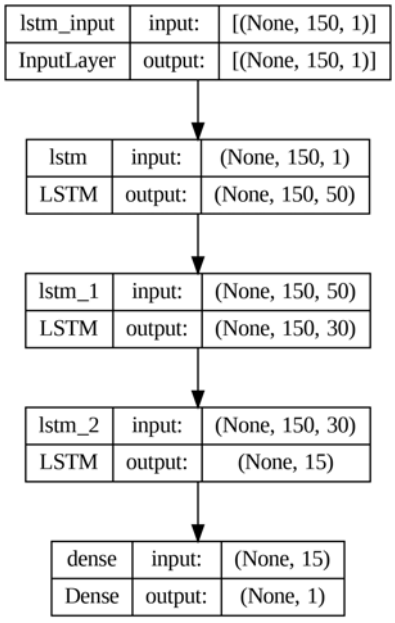


Figure 5.2: Stacked LSTM Model Architecture.

5.4.4 Error Correction Neural Networks

Tables 5.3, and 5.4 show the architecture of the ECNN and ECNN with exponential smoothing respectively. The timestep, batch size, number of neurons and epochs were obtained experimentally.

Table 5.3: Architecture of the ECNN model.

timestep	**batch size**	**neurons**	**epochs**
5	64	34	1000

Table 5.4: Architecture of the ECNN with Exponential Smoothing.

timestep	**batch size**	**neurons**	**epochs**
7	512	32	100

5.4.5 Exponential Smoothing Implementation

This section provides a detailed explanation of how exponential smoothing was performed on the input data for the ECNN and LSTM models that employed exponential smoothing. The experimental setup for the LSTM and ECNN models used with exponential smoothing is shown in Tables 5.4

Exponential smoothing is a popular time series forecasting technique that efficiently captures and represents underlying patterns and trends in the data. By applying this technique to the input variables, our models are able to leverage the historical information contained within the data to make accurate predictions.

The original time series data was smoothed using an exponential weighted average, the smoothed dataset was subsequently used to make predictions. Each observation was weighted according to how recent it was, at that timestep, in the time series.

The level of the series at time t, l_t, is computed according to Equation 5.2,

$$l_t = \alpha y_t + (1 - \alpha) l_{t-1}, \tag{5.2}$$

where α is a smoothing parameter ranging between 0 and 1, and y_t is the value of the time series at time t. The input of the machine learning model is subsequently computed by,

$$x_t = \ln\left(\frac{input_t}{lt}\right). \tag{5.3}$$

x_t is then used as the input of the model at time t. The output of the model at time t is give by:

$$o_t = model(x_t). \tag{5.4}$$

The predictions are scaled back to their original representation using Equation 5.5,

$$\hat{y}_t = l_t \exp(o_t). \tag{5.5}$$

5.5 MODEL EVALUATION

This evaluation aimed to measure the efficiency, accuracy, and generalization capability of a model. The models were evaluated using the mean absolute percentage error and the root mean square error, which are described in Section 3.3. In addition to these metrics, the Theil U coefficient and directional accuracy metrics were also used, this is shown in Equation 5.6 and 5.7. To measure the computational efficiency, the time taken to run the model was measured.

Directional Accuracy

This metric measures the proportion of the times the model forecasts the correct direction of the prediction (whether it's increasing or decreasing).

$$DA = \frac{1}{n-1}\sum_{t+1}^{n-1} pos((\hat{y}^{(t+1)} - \hat{y}^{(t)})(y^{(t+1)} - y^{(t)}), \tag{5.6}$$

where:

$$pos(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases}$$

Here, y_t is the target at time t, $\hat{y}_t$ is the prediction at time t and n is the length of the prediction and target data.

Theil U

This metric measures how well a time series of predictions compares to the time series of corresponding targets.

$$TheilU = \frac{\sqrt{\frac{1}{n}\sum_i (y_t - \hat{y}_t)^2}}{\sqrt{\frac{1}{n}\sum_t y_t^2} + \sqrt{\frac{1}{n}\sum_t \hat{y}_t{}^2}}. \tag{5.7}$$

5.6 CONCLUSION

In conclusion, this chapter presented the methodologies used for constructing machine learning models to predict COCOON and difference data. The prediction model framework outlined in Section 5.2 provided a structured approach for model development, ensuring consistency and reproducibility. Section 5.3 clarified the two datasets of interest, and explained the preprocessing techniques employed, such as the implementation of the min-max scaler and exponential smoothing of the input data. Model selection and hyperparameter tuning were discussed in Sections 5.3.1 and 5.3.2. The experimental design discussed in Section 5.4 encompassed the hyperparameter tuning technique which was used, ensuring optimal model performance. Furthermore, model evaluation was covered, enabling a comprehensive assessment of the predictive capabilities of the developed models. By following these methodologies, a robust prediction system for COCOON and difference data was established, forming a solid foundation for subsequent analysis and insights.

CHAPTER 6

RESULTS

6.1 INTRODUCTION

The primary objective of this chapter is to present the key findings derived from the machine learning models that were implemented to make COCOON predictions. Section 6.2.1 discusses the outcomes derived from the implementation of the LSTM and its variations, including the vanilla LSTM, stacked LSTM, and LSTM with exponential smoothing (LSTM es). These models were trained individually on the difference and COCOON datasets. In Section 6.3, we showcase the results obtained from the implementation of ECNN, as well as its variant, the ECNN with exponential smoothing (ECNN es). Similarly, both of these models were trained separately on the COCOON and difference datasets. In Section 6.4, we thoroughly evaluate the performance of these models using suitable metrics. Finally, we extend our evaluation to the top three models by analysing the residuals through statistical analysis, while also examining their computational efficiency by measuring the time taken for each model to execute.

6.2 LONG SHORT-TERM MEMORY NETWORK

The general structure of the LSTM is shown in Figure 4.2. The architecture can be modified by altering the number of hidden (LSTM) layers, and the number of neurons in the hidden layer.

6.2.1 Vanilla LSTM

This section presents the initial LSTM model, in which one hidden layer was used. Hyperparameters such as batch size, the number of epochs, and the dropout rate were experimented with order to find the best architecture for the model. The choice of these hyperparameters was made using the grid search algorithm, where various hyperparameter combinations are evaluated to find the best model, Table 6.1 shows the hyperparameter space used. COCOON data was used for the initial model during hyperparameter tuning.

Table 6.1: Values for LSTM hyperparameters in the Grid Search.

Hyperparameter	**Values**
batch size	2,4,8,16,32,64,128,256
neurons	5,10,15,19,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100
epochs	50,100,200,300,400,500
dropout rate	0,0.01,0.03,0.05,0.1,0.15,-.2,0.25,0.3,0.35,0.4

Table 6.2 shows the top 5 models resulting in the lowest RMSE, with their respective hyperparameter combination, using an initial timestep of 4. A regularisation technique (dropout) was initially employed to prevent the LSTM from overfitting. This was tested during hyperparameter tuning, but the grid search results showed that the LSTM performed better with little to no dropout. With the best 5 models from the grid search all having a dropout rate of 0, as shown in Table 6.2.

Table 6.2: A summary of the top 5 LSTM models showing the model hyperparameters and test RMSE.

Model	**batch size**	**neurons**	**epochs**	**dropout rate**	**RMSE**
1	32	19	1000	0	27.13
2	2	30	200	0	38.33
3	2	50	200	0	51.42
4	2	30	1000	0	51.71
5	2	15	200	0	82.37

Furthermore, the timestep of the input data was also altered in order to optimise the model. The top 5 best models from the grid search were tested with timesteps varying from 4 to 20, this is shown in Figure 6.1. Model 1 exhibits its lowest RMSE of 27.13 when a timestep of 4 is used, which is the highest RMSE among the 5 models, making it the least accurate model. This could be attributed to its larger batch size compared to the other models, which results in less frequent updates of the model parameters. This affects convergence and can lead to suboptimal solutions, due to the less frequent updates reducing noise in the gradient during backpropagation, resulting in overfitting. Model 2 achieves its lowest RMSE of 11.57 when a timestep of 14 is used. Model 3 obtains its lowest RMSE of 10.61 using a timestep of 17, making it the best performing model.

This is due to a combination of two main factors, small batch size and an optimal number of neurons. The batch size is small, leading to frequent updates of the model parameters, which introduces noise to the gradient during backpropagation, preventing overfitting and leading to better generalisation. Additionally, the number of neurons is not too low or too high, allowing the model to strike a balance between accurately capturing the underlying patterns within the data and avoiding excessive complexity. Model 4 obtains its lowest RMSE of 12.68 using a timestep of 15. Finally, Model 5 obtains its lowest RMSE of 21.8 using a timestep of 17, making it the second least accurate model out of the 5. This could be due to the small number of neurons in the model, resulting in a model of lower complexity than its counterparts. Models of low complexity tend to struggle to learn non-linear relationships in data.

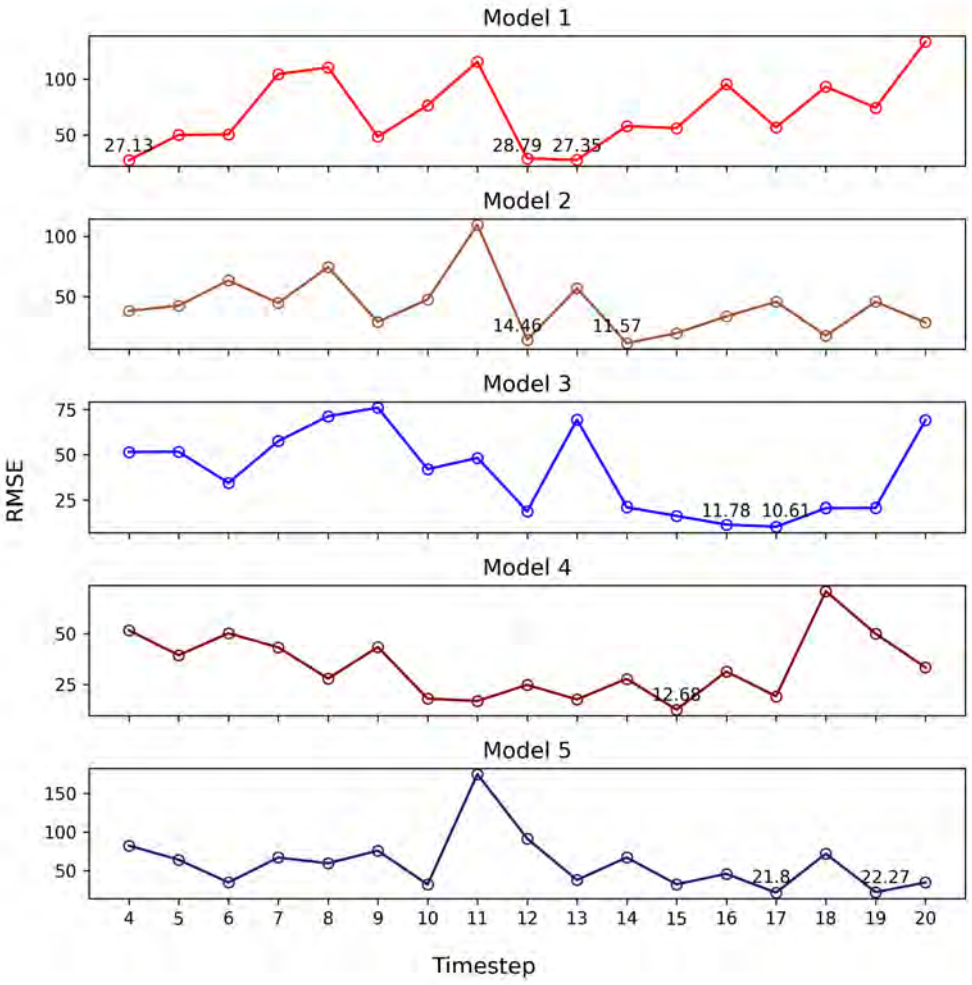


Figure 6.1: LSTM: RMSE for varying timesteps for the top 5 models.

The architecture of the best model is shown in Table 6.3.

Table 6.3: Summary of the best LSTM model.

Model	**timestep**	**batch size**	**neurons**	**epochs**	**dropout rate**	**RMSE**
3	17	2	50	200	0	10.61

Figure 6.2 shows the corresponding predicted vs actual values for this model when trained and tested on COCOON data, at first glance it seems that the predictions perfectly align with the targets.

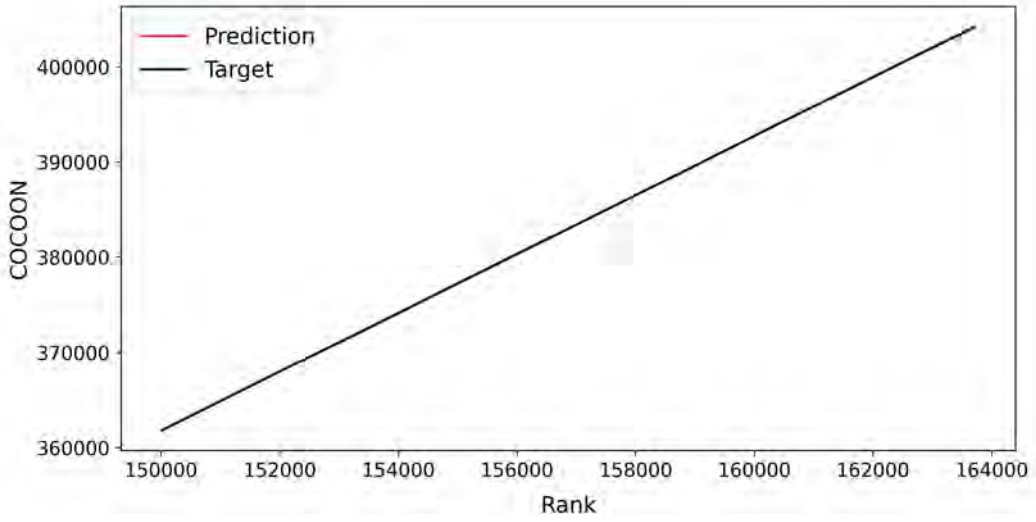


Figure 6.2: Predicted vs Actual for an LSTM trained on COCOON data.

However, when taking a much closer look, the predictions move further away from the target values as the rank of the COCOONS increases. This is shown in Figure 6.3, where each subfigure looks more closely at the behaviour of the predictions versus the target values, by examining them at a much smaller range. The rank at which the results are observed is increased progressively in each subsequent subfigure.

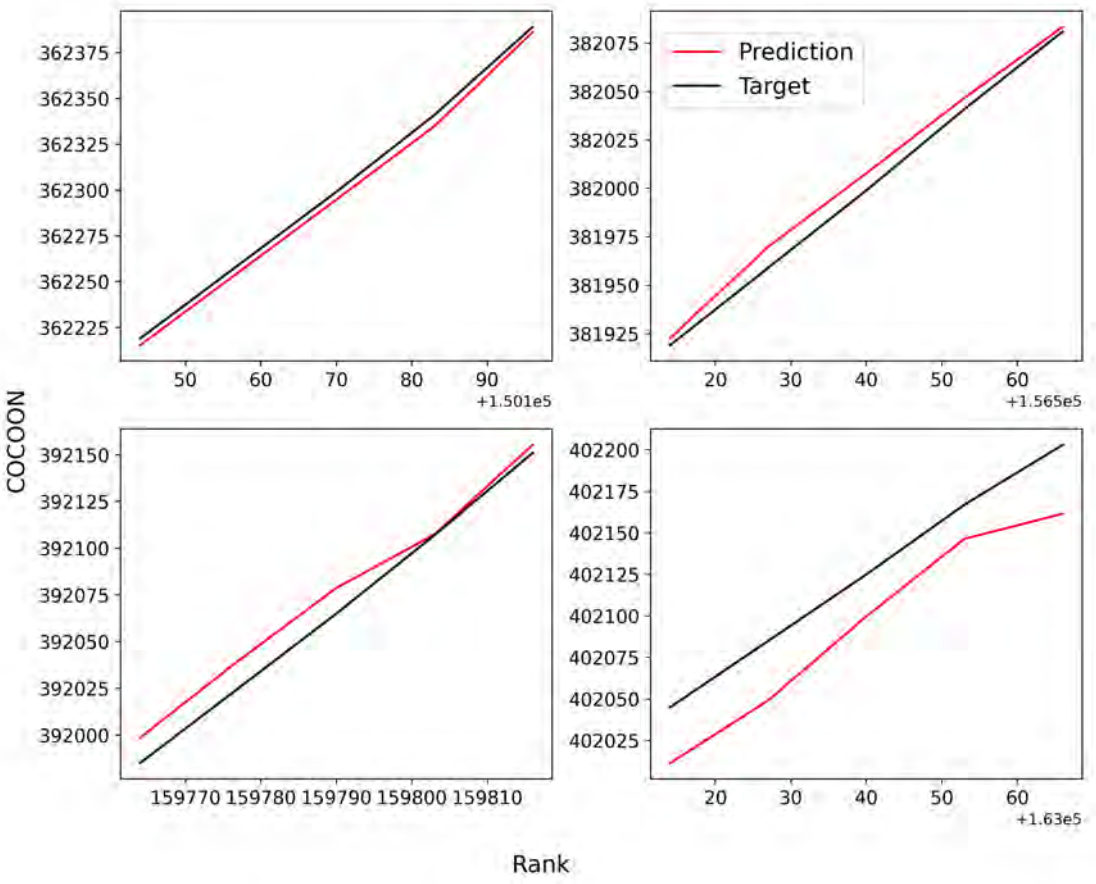


Figure 6.3: Predicted vs Actual for an LSTM trained on COCOON data.

In order to improve this performance, the COCOON data was transformed into a difference dataset, which has a much simpler structure as the values within the dataset belong to a small finite set of values $\{1, 2, 3\}$.

Figure 6.4 shows the predicted versus actual values of the model when trained on difference data. The figure shows that the LSTM trained on difference data is able to capture the distribution pattern of the differences, with most of the predictions being centred around 1 (corresponding to a difference of 2), fewer being around 2 (corresponding to a difference of 3), and even fewer being around 3 (corresponding to a difference of 6). This captures what was established about the distribution of these differences in Section 3.3, shown in Figures 2.4- 2.6.

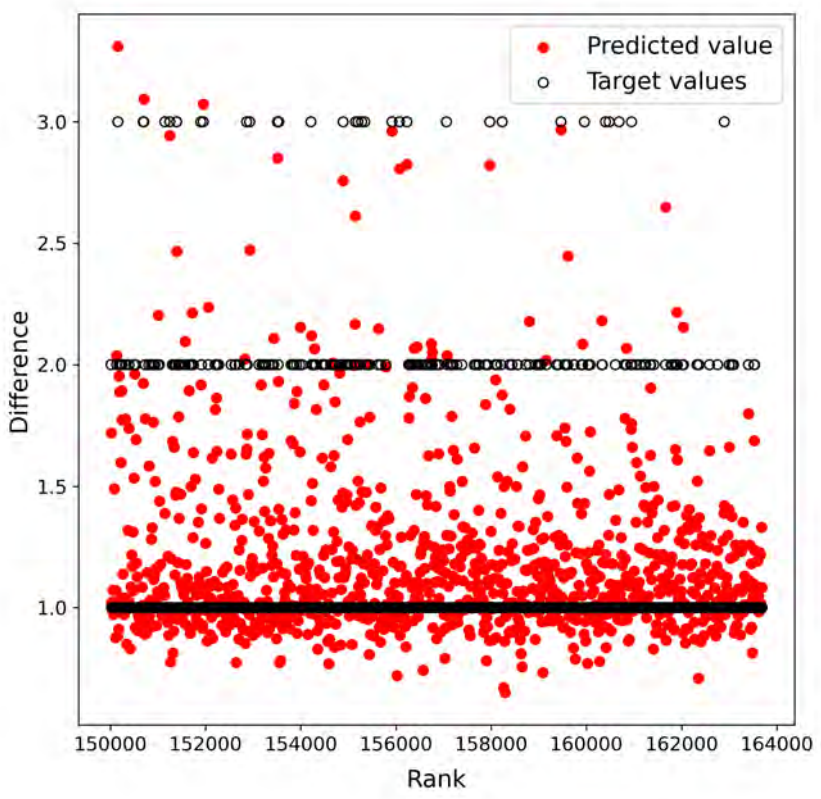


Figure 6.4: Predicted vs Actual for an LSTM trained on Difference data.

The difference values are subsequently scaled back to their COCOON representation, which is shown in Figure 6.5. This is done by using the following procedure:

Recall that the difference data= $\{d_i : i \in \mathbb{N}\}$, where $d_i = \frac{N_{i+1}-N_i}{2}$, and N_{i+1}, N_i are COCOONs, let d_i be the i^{th} difference in the test dataset, and p_n the n^{th} prediction,

Equation 6.1 scales the difference predictions to their COCOON representation,

$$c_n = U_0 + (\sum_{i=0}^{n-2} 2d_i) + 2p_n, \tag{6.1}$$

where U_0 is first COCOON in the test dataset. Figure 6.5 shows that the predictions made with the LSTM trained on difference data, are very close to the target values when scaled back to their respective COCOON representation. This figure takes a close look at the predictions versus targets in their COCOON representation, by examining their behaviour within small ranges. In order to comprehensively visualise the model's behaviour, the rank is progressively increased in each subsequent subfigure to observe the results across low, medium, high and higher ranks.

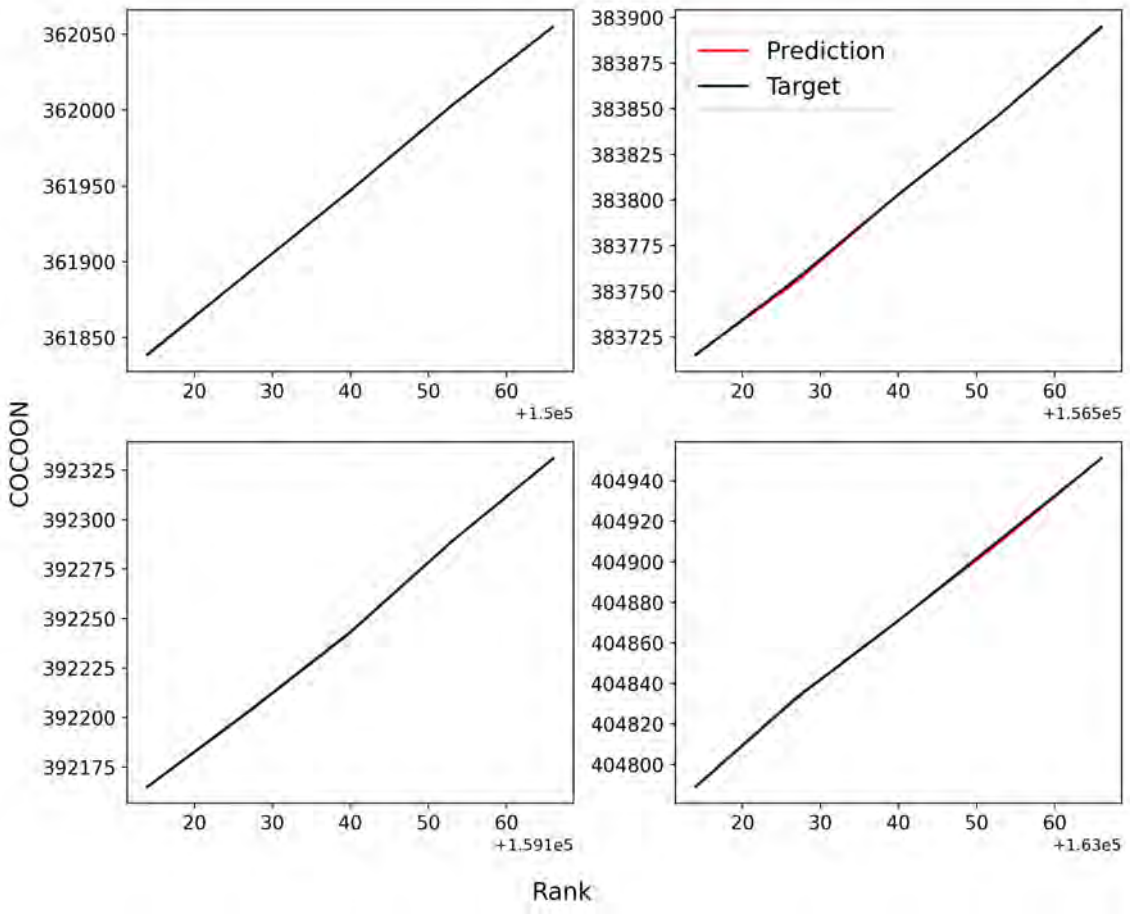


Figure 6.5: Predicted vs Actual (scaled back to COCOON representation) for an LSTM trained on Difference data.

6.2.2 Stacked LSTM

Research conducted by Sutskever *et al.* (2014) demonstrates that stacked LSTMs have a higher capacity to model complex sequences compared to vanilla LSTMs. Therefore, two additional LSTM layers were added to the vanilla LSTM, forming a more complex network. This resulted in a stacked LSTM with 50, 30 and 15 neurons in each layer respectively. A timestep of 150 was chosen experimentally, the architecture of this model is shown in Figure 5.2. The stacked LSTM was trained on COCOON and difference data, but performed worse than its vanilla counterpart.

Recall that upon closer examination, in Figure 6.3, it was evident that the predictions made by the vanilla LSTM deviate significantly from the target values. This deviation was not initially apparent when looking at an overview, as depicted in Figure 6.2. In the case of the stacked LSTM, Figure 6.6 shows that even without closer examination, the predictions diverge further away from the target values as the rank increases.

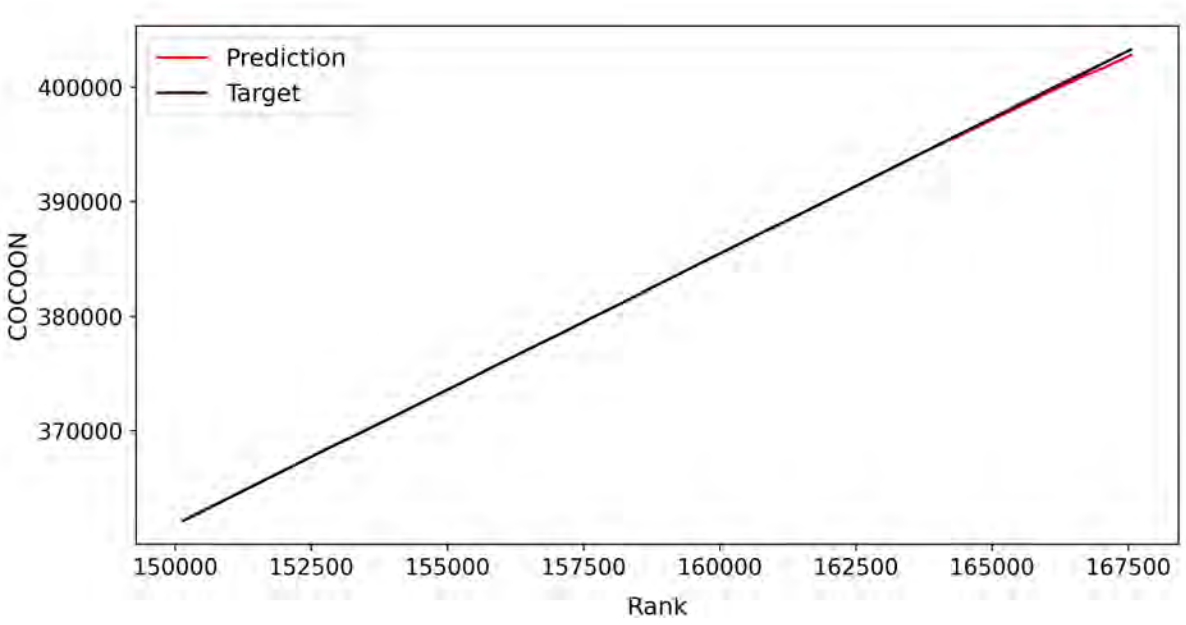


Figure 6.6: Predicted vs Actual for a Stacked LSTM trained on COCOON data.

Furthermore, when trained on difference data, the stacked LSTM performed poorly. With most predictions being centered around 1, indicating a failure to accurately capture the distribution pattern of the differences. This is shown in Figure 6.7.

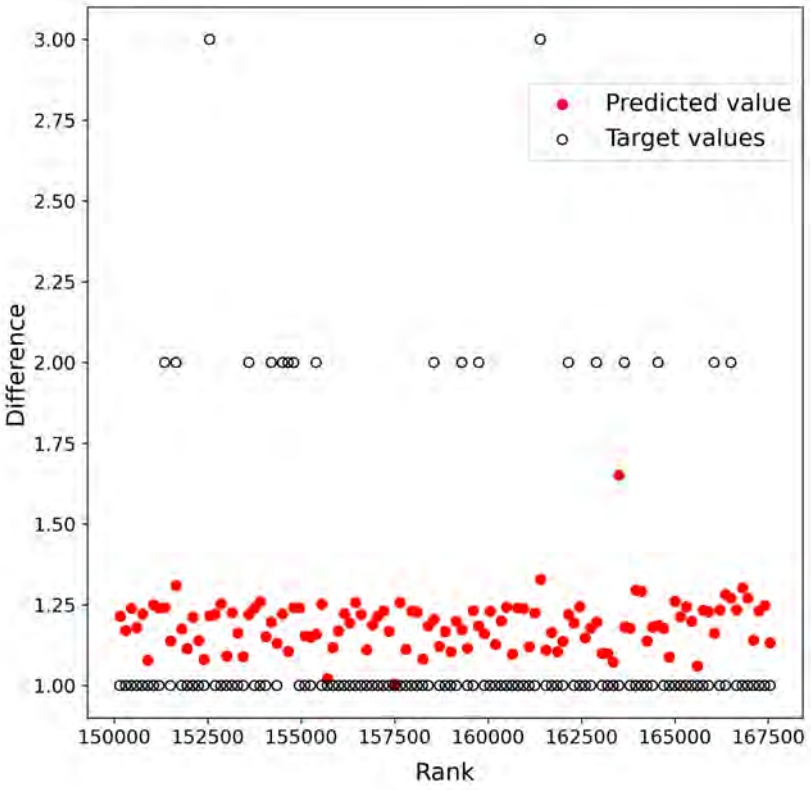


Figure 6.7: Predicted vs Actual for a Stacked LSTM trained on Difference data.

The testing data consisted of COCOONs and differences of much higher rank than the training data, resulting in significantly fewer differences of 3. This was further amplified by the use of a large timestep. To address this issue, the dataset size was increased, but unfortunately, it did not yield any improvements. On the contrary, the model's performance deteriorated. Training the stacked LSTM proved to be computationally exhaustive and time-consuming, often causing resource depletion before completion, especially when the size of the dataset was made larger. The stacked LSTM also performed poorly on unseen data, this was due to the increased complexity and number of parameters in the model, which made it prone to memorising the training data instead of generalising. Consequently, the stacked LSTM was deemed suboptimal, and the vanilla LSTM was preferred due to its computational efficiency and ability to generalise.

6.2.3 LSTM with Exponential Smoothing

In order to optimise the vanilla LSTM even further, exponential smoothing was performed on a vanilla LSTM trained using COCOON data, and on a vanilla LSTM trained using difference data. This method was outlined in Section 5.4.5. Figure 6.8 shows the predicted versus actual values for the LSTM with exponential smoothing (LSTM es) trained using COCOON data, with the rank increasing progressively in each subsequent subfigure. This figure clearly demonstrates a significant discrepancy between the predicted values and the desired targets, indicating the model's inefficiency.

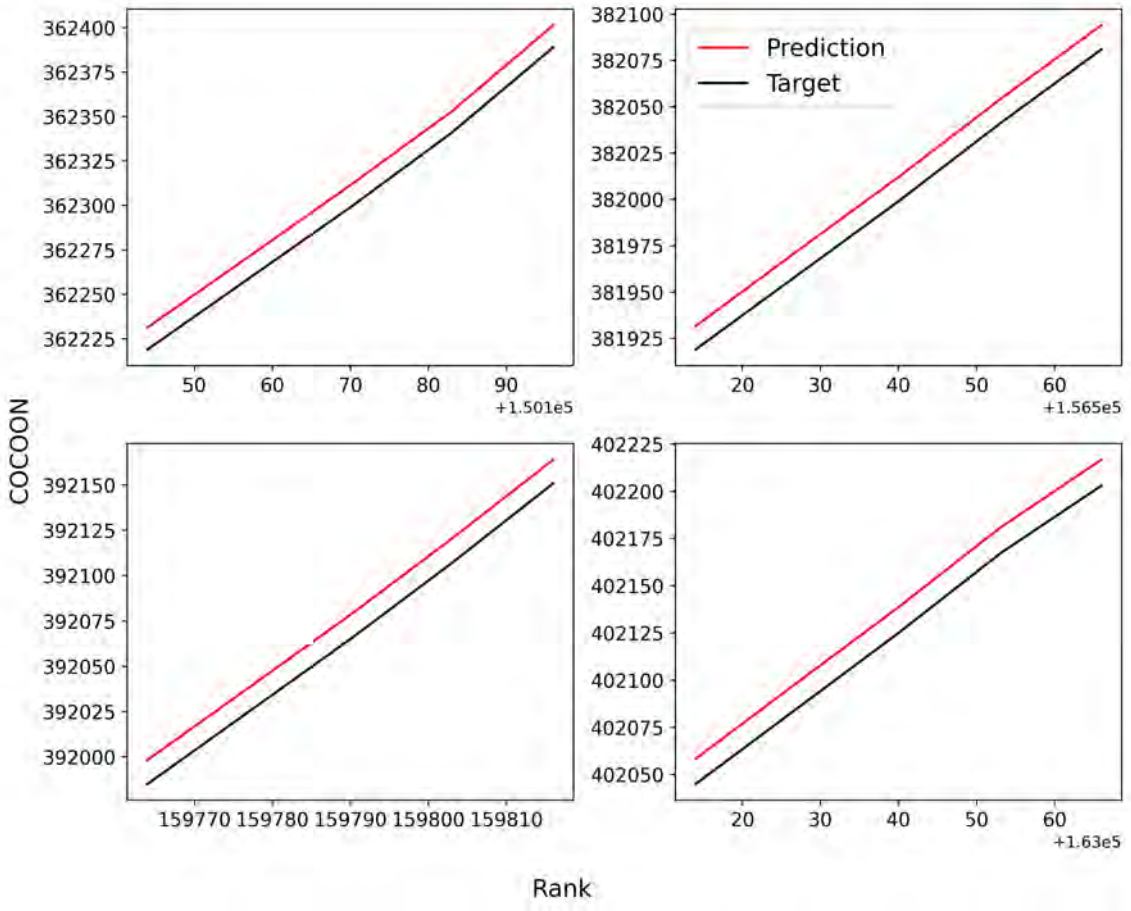


Figure 6.8: Predicted vs Actual for LSTM with exponential smoothing, trained using COCOON data.

Additionally, the model performed significantly worse than its counterpart without exponential smoothing (Section 6.2.1, Figure 6.2 and 6.3). Resulting in a RMSE of 164.5, compared to the model without exponential smoothing's RMSE of 10.61. The under performance can be attributed

to exponential smoothing's tendency to prioritise recent observations and disregard older ones. In the case of the COCOON data, which is strictly increasing, this led to excessive smoothing, causing the model to fail to adjust to any sudden changes in the underlying pattern of the data. As a consequence, the forecasts were suppressed. This explanation is further supported by the negative mean error of the model, which was recorded at -7.9.

The LSTM with exponential smoothing was also trained using the difference dataset, which was shown to lead to a better performing model in Section 6.2.1. Figure 6.9 shows that model is able to capture the distribution pattern of the differences, even better than its counterpart without exponential smoothing (Section 6.2.1, Figure 6.4).

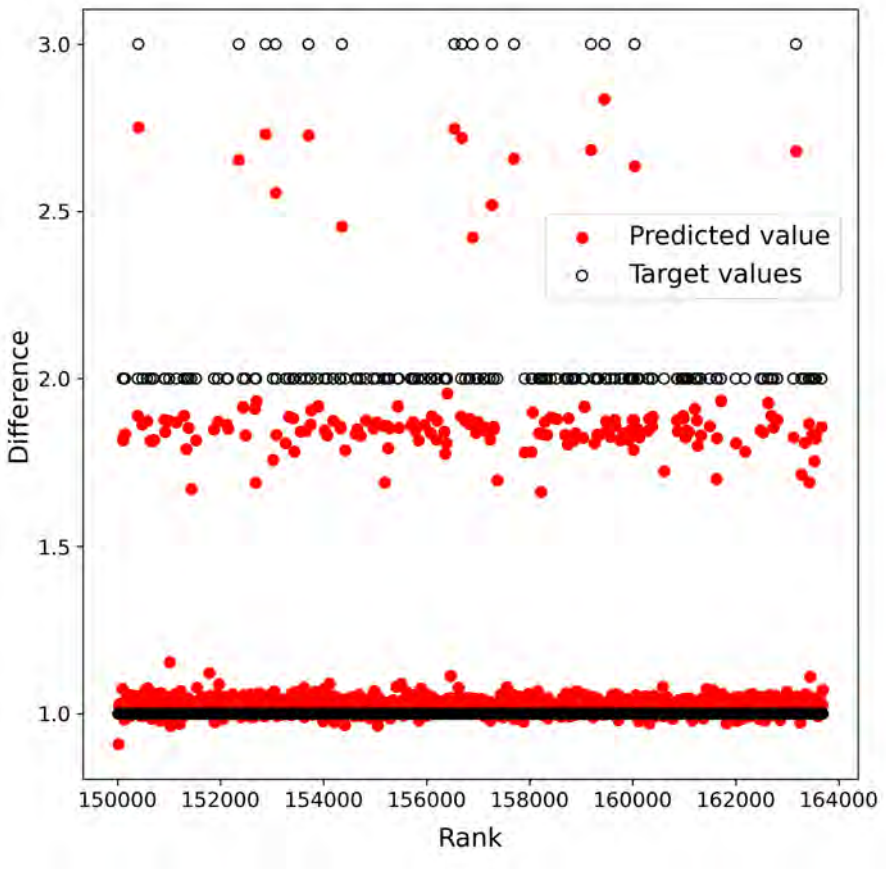


Figure 6.9: Predicted vs Actual for LSTM with exponential smoothing, trained using Difference data.

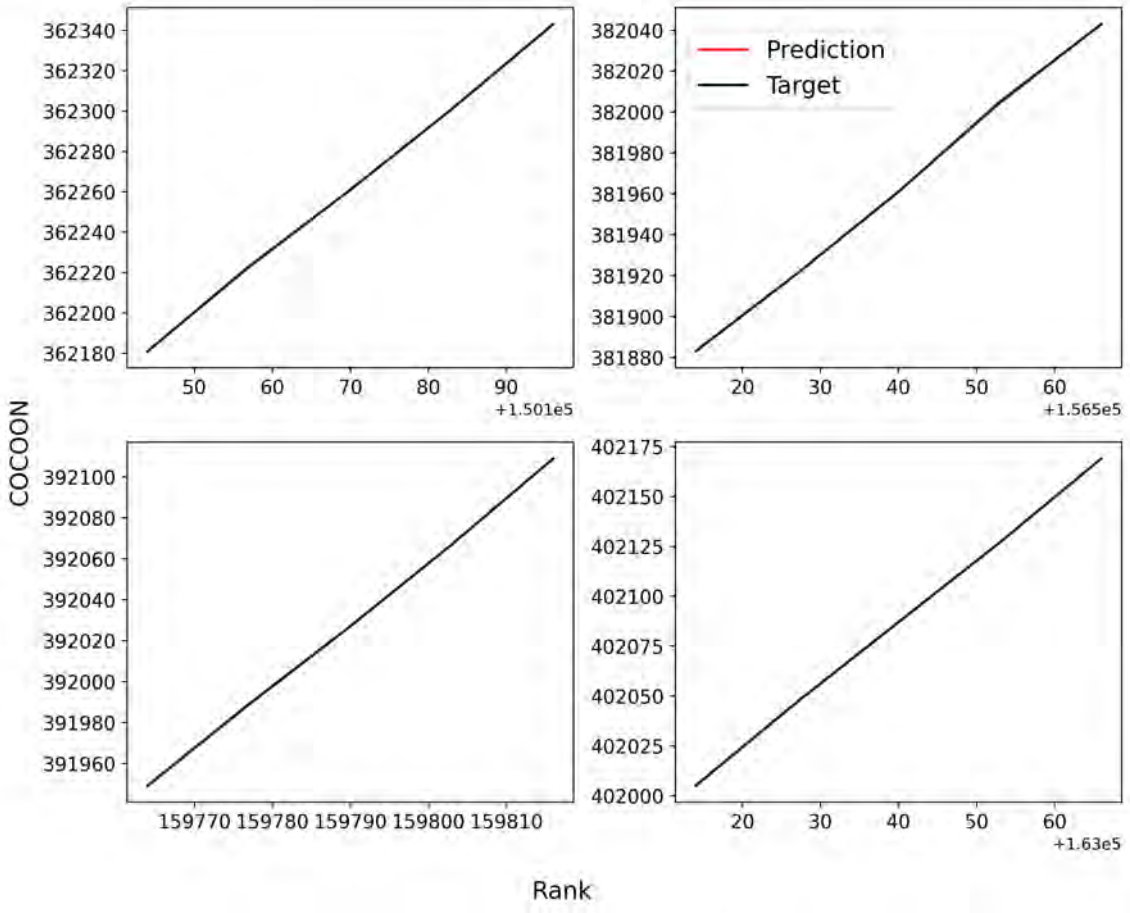


Figure 6.10: Predicted vs Actual (scaled back to COCOON representation) for LSTM with exponential smoothing, trained using Difference data.

Figure 6.10 shows the predicted versus actual values once they are scaled back to their COCOON representation, with the rank increasing progressively in each subsequent subfigure. This figure illustrates that the predictions and target values are nearly indistinguishable as the lines overlap each other seamlessly. Figure 6.9 and 6.10 demonstrate the exceptional performance of the LSTM with exponential smoothing when trained on difference data. Exponential smoothing is inherently designed to rapidly adapt to recent changes in data, which explains the models ability to effectively capture short-term fluctuations and adjust the forecast accordingly. In Section 2.3, it was shown that as the rank increases, the occurrence of differences of 6 (3 when normalised) becomes increasingly rare, while differences of 2 (1 when normalised) become more prevalent throughout the dataset. As a result, the distribution of these differences undergoes significant changes as the rank increases. The advantage of this model, with exponential smoothing, lies in its capacity to quickly adapt to these changes. This especially beneficial in when using the difference data, in which each element in the input data only differs by another by a very small amount, as it lessens the influence of outliers or irregular patterns that might otherwise hinder the accuracy of the forecasts.

6.3 ERROR CORRECTION NEURAL NETWORK

Figure 6.11 shows the learning curve associated with an ECNN trained using COCOON data. It shows that the training loss, remains relatively flat regardless of training, indicating that the model is not learning. This is further supported by Figure 6.12 which shows the large gap between predicted and target values for this model.

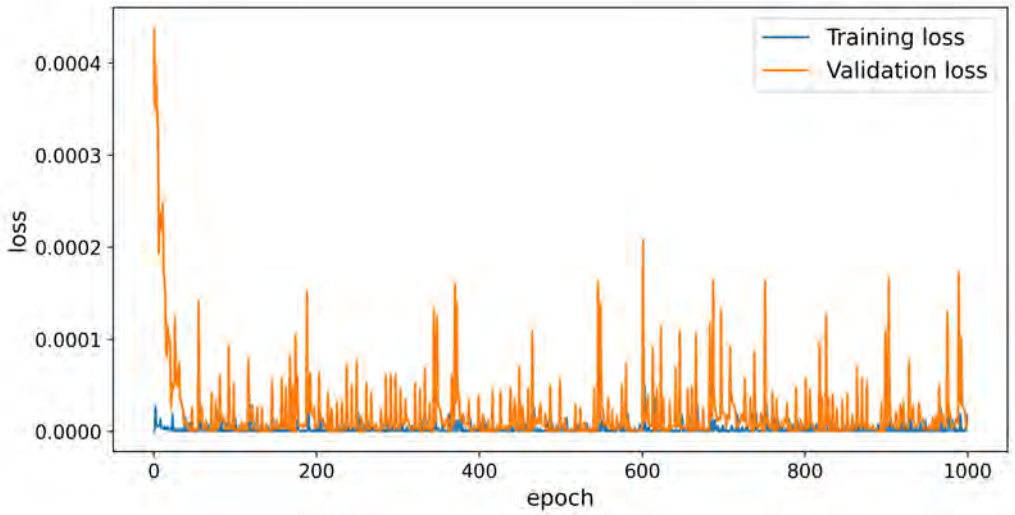


Figure 6.11: Training vs Validation loss for ECNN trained using COCOON data.

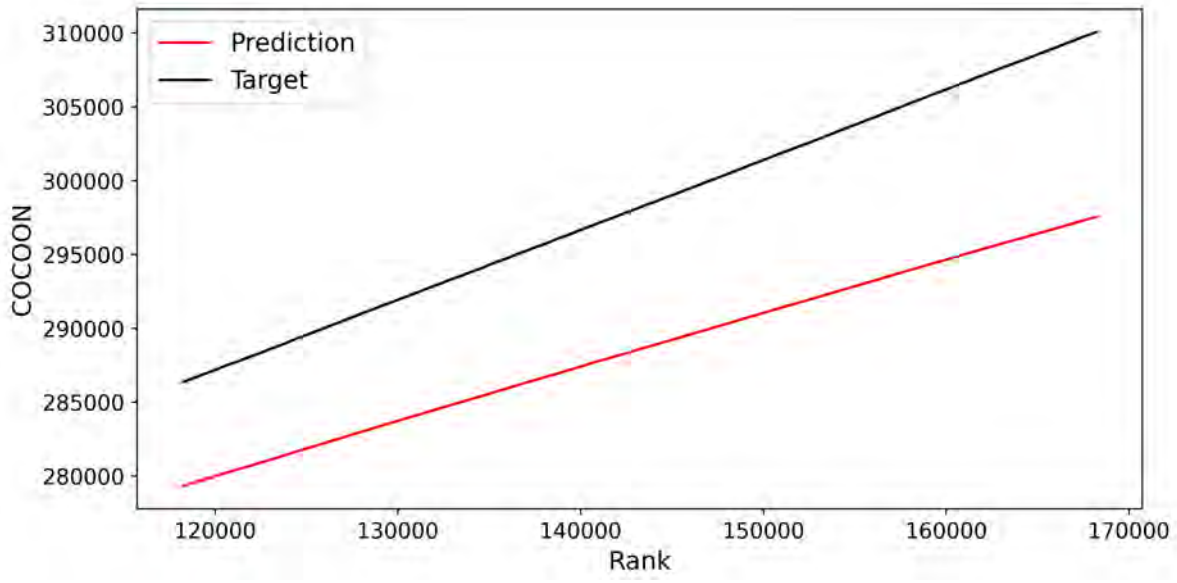


Figure 6.12: Predicted vs Actual for ECNN trained using COCOON data.

In order to improve this model, the COCOON dataset was simplified to the difference dataset, and an ECNN was trained using the difference data. Figure 6.13 shows the learning curve of the ECNN trained using difference data, which shows that there is constantly a big gap between the training and validation error, with the validation error being significantly lower, indicating that the validation data is much easier for the model to predict. This may be due to complex model architecture, causing the model memorise the training data rather than learning general patterns. The training error decreases until it reaches stability.

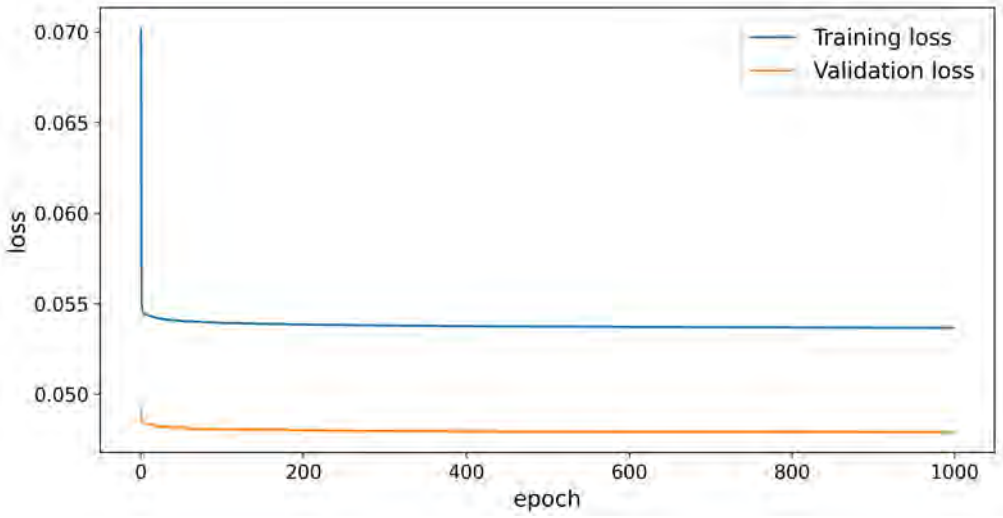


Figure 6.13: Training vs Validation loss for ECNN trained using Difference data.

Figures 6.14 and 6.15 show the predicted vs actual plots of an ECNN trained using difference data, with Figure 6.14 showing that the ECNN struggles to predict the distribution pattern of the differences, with most predictions being close to 1. However, Figure 6.15 shows that once the predictions are scaled back to their COCOON representation, they are very close to the target COCOONs. In this figure, the rank progressively increases in each subsequent subfigure, demonstrating that the closeness between the predictions and targets is consistent when moving from low to medium and high ranks.

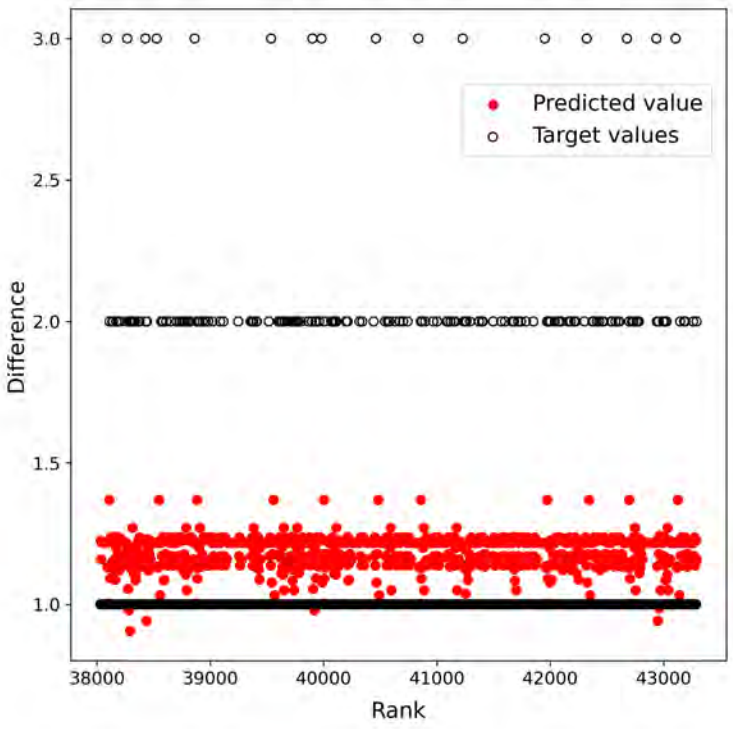


Figure 6.14: Predicted vs Actual for an ECNN trained on Difference data.

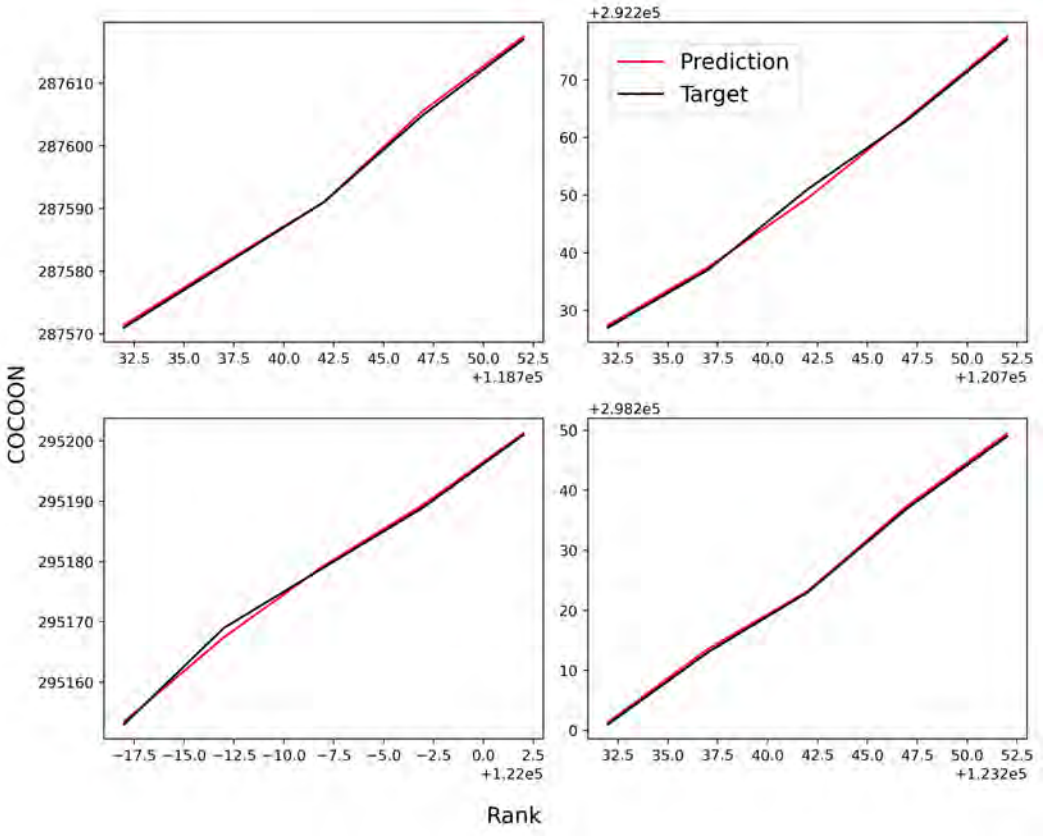


Figure 6.15: Predicted vs Actual (scaled back to COCOON representation) for ECNN trained on Difference data.

6.3.1 ECNN with Exponential Smoothing

Exponential smoothing was subsequently applied to the ECNN in order to improve its performance. Figure 6.16 shows that the training error of the ECNN model with exponential smoothing (ECNN es) deployed on COCOON data decreases rapidly to a point of stability, while the validation error remains relatively flat around 0, indicating that the model is overfitting.

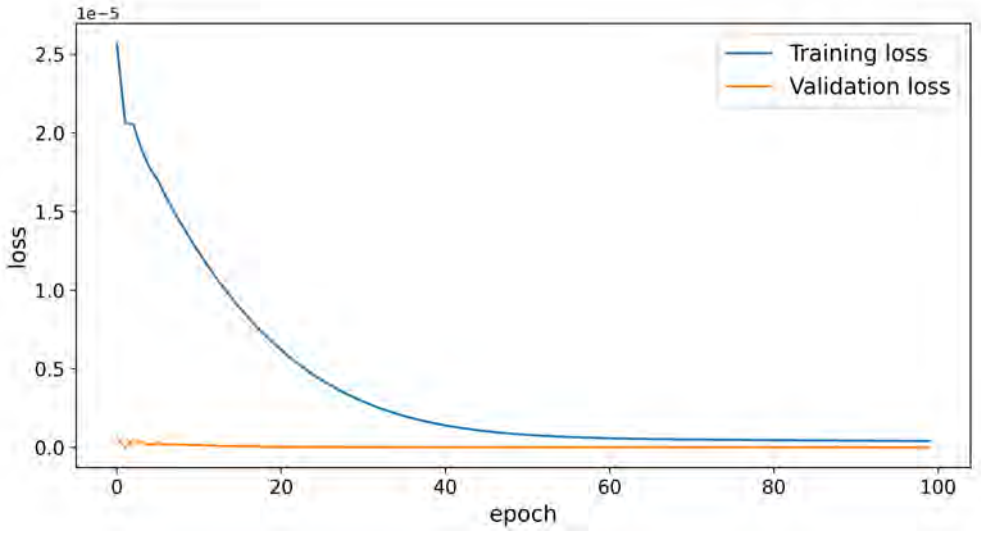


Figure 6.16: Training vs Validation loss ECNN with exponential smoothing, trained using COCOON data.

Figure 6.17 shows the predictions vs the target values for this model, showing that the predictions move further away from the target values as the rank of the COCOONS increases. As previously established, the LSTM trained using COCOON data exhibits similar behaviour (Figure 6.2). However in the case of the LSTM, this is not apparent until one takes a closer look (Figure 6.3), the ECNN with exponential smoothing very clearly exhibits the divergence of the predictions from the target values without having to look too closely, indicating that this model fit performs poorer than its LSTM counterpart.

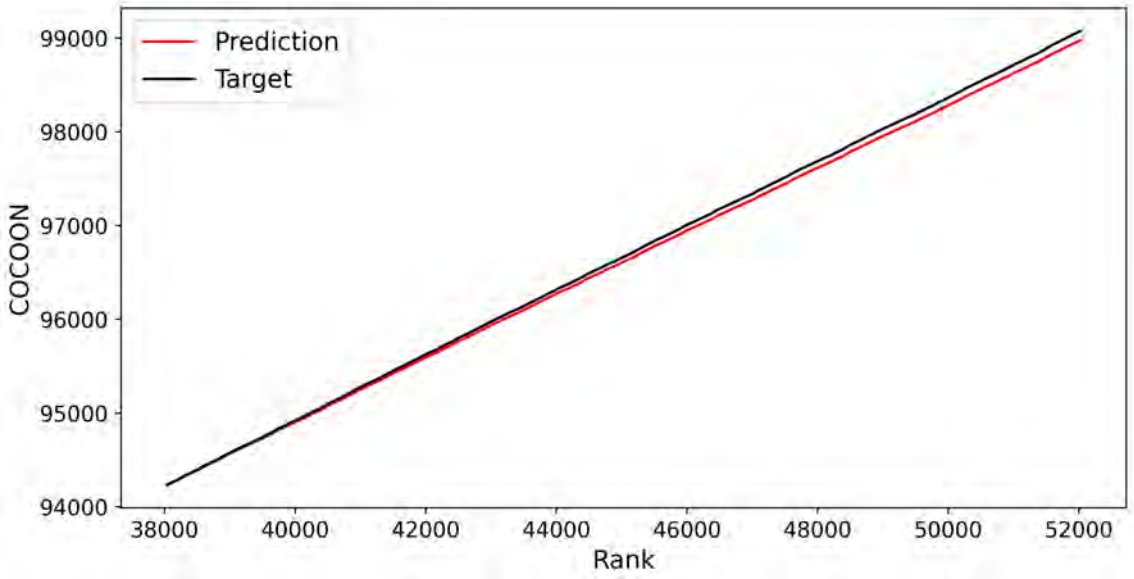


Figure 6.17: Predicted vs Actual for ECNN with exponential smoothing trained with COCOON data.

Figure 6.18 shows that the training and validation error associated with the ECNN with exponential smoothing, trained using difference data, decreases to a point of stability, until there is very little gap between the two curves, indicating a good model fit.

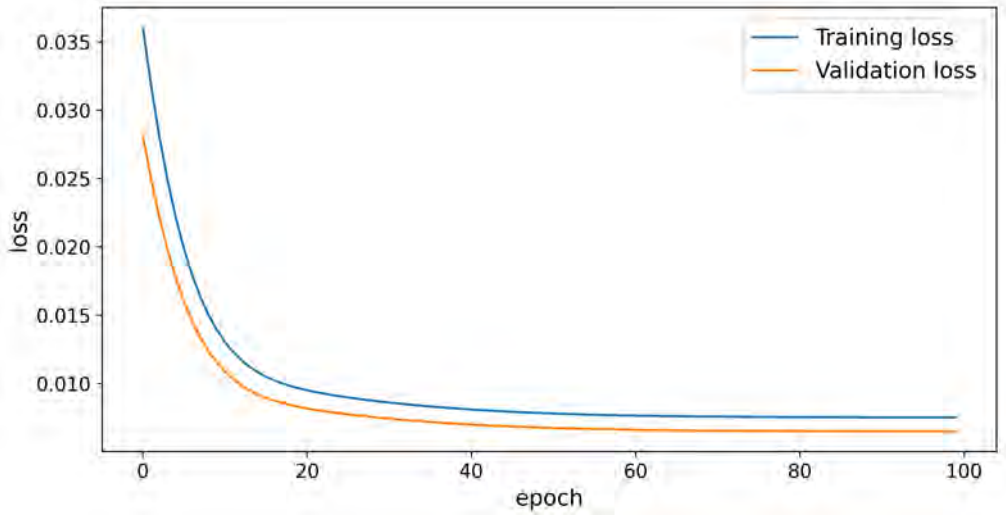


Figure 6.18: Training vs Validation loss for ECNN with exponential smoothing, trained using Difference data.

The model captures the distribution pattern of the differences, with most predictions being centered around 1, fewer around 2, and even fewer around 3, following the distribution pattern established in Section 2.3. This is shown in Figure 6.19.

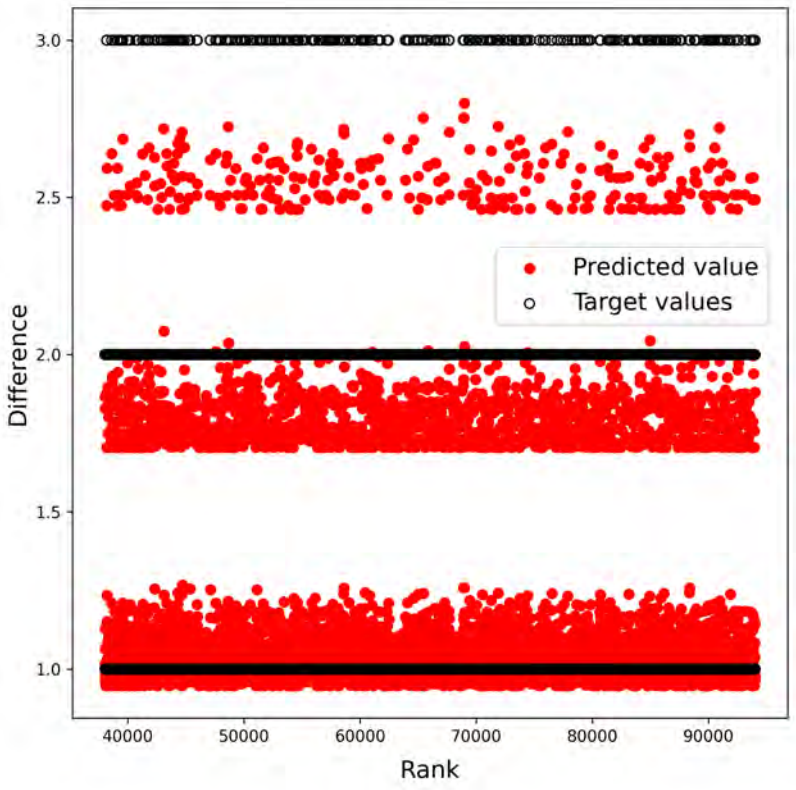


Figure 6.19: Predicted vs Actual for ECNN with exponential smoothing , trained using Difference data.

The model subsequently provides a good fit once the difference predictions are scaled back to their COCOON representation. This is shown in Figure 6.20, where the rank is progressively increased in each subsequent subfigure, in order to observe results across low, medium and high ranks.

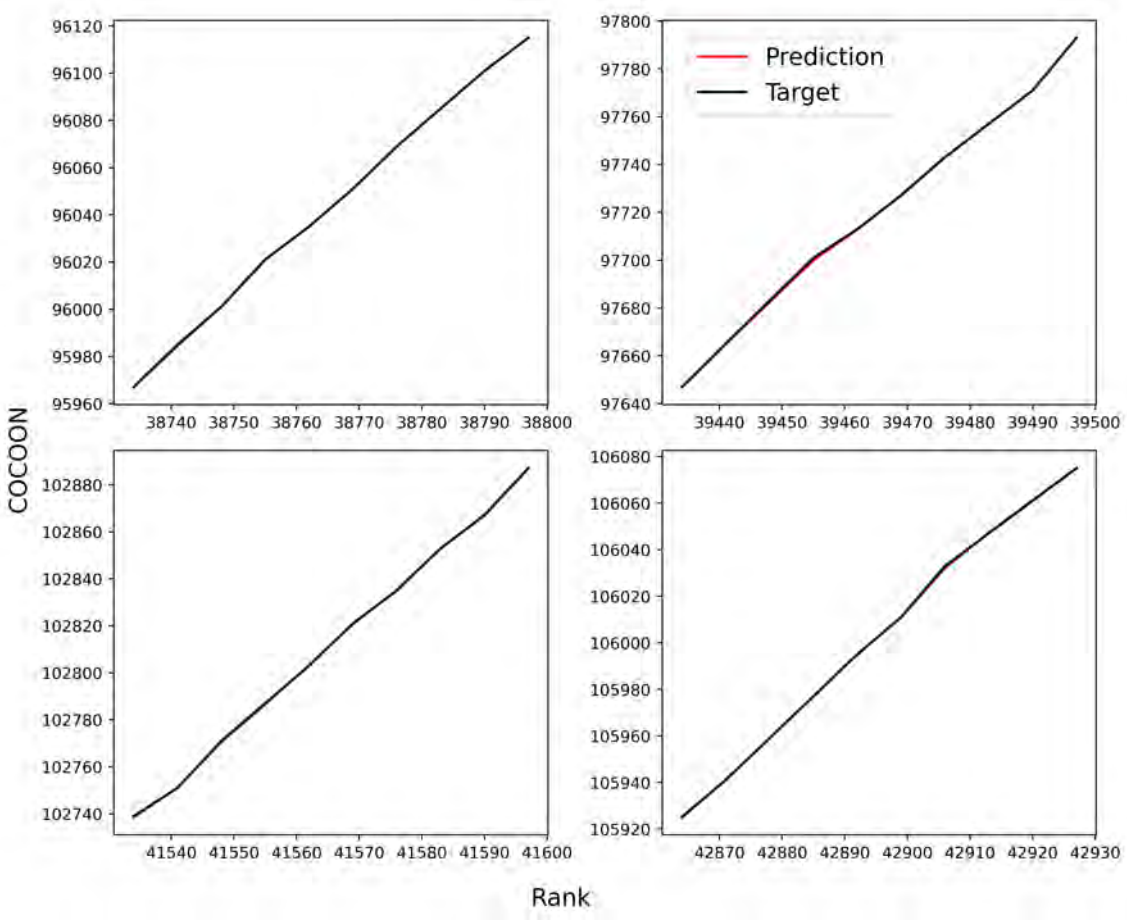


Figure 6.20: Predicted vs Actual (scaled back to COCOON representation) for ECNN with exponential smoothing, trained using Difference data.

6.4 MODEL EVALUATION

In this section, we present a comprehensive evaluation through various metrics measuring the performance of the various models, deployed on COCOON and difference datasets.

The accuracy of various models, using different datasets and metrics, can be observed in Tables 6.4 - 6.7. In the case of models trained using difference data, the predictions and targets were first converted back to their COCOON representation, before the metrics were calculated. Therefore, the metrics have the same scale across datasets.

Table 6.4: Predictive accuracy of different models using MAPE.

Model	**Cocoons**	**Differences**
ECNN	3.8415	0.0002
ECNN es	0.0194	0.0008
LSTM	0.0021	0.0001
LSTM Stacked	0.01	0.0001
LSTM es	0.0048	0.0000

Table 6.5: Predictive accuracy of different models using RMSE.

Model	**Cocoons**	**Differences**
ECNN	15458.17	0.8609
ECNN es	3.1742	0.2837
LSTM	10.61	0.7963
LSTM Stacked	51.35	0.8799
LSTM es	164.53	0.1698

Table 6.6: Predictive accuracy of different models using DA.

Model	**Cocoons**	**Differences**
ECNN	99.98	99.9949
ECNN es	99.9986	99.8093
LSTM	99.91	99.90
LSTM Stacked	99.15	98.8108
LSTM es	99.9051	99.8093

Table 6.7: Predictive accuracy of different models using Thiel U.

Model	**Cocoons**	**Differences**
ECNN	0.15055	0.00894
ECNN es	0.0078	0.0023
LSTM	0.0068	0.0010
LSTM Stacked	0.0081	0.0011
LSTM es	0.1466	0.0005

Figures 6.21- 6.23 depict various forecast metrics comparing models trained with COCOON data to models trained with difference data.

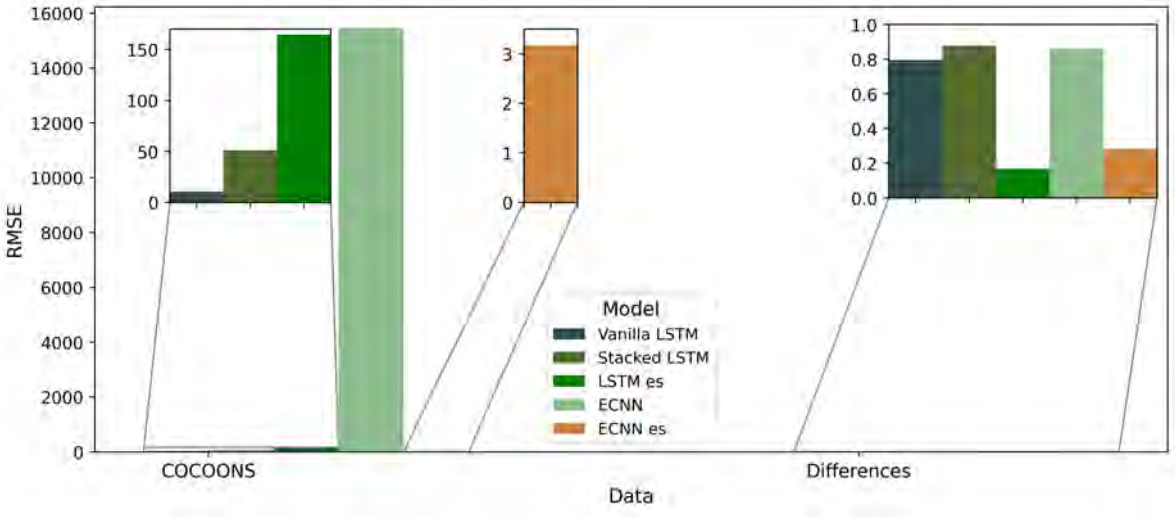


Figure 6.21: Root Mean Square Error of different models deployed on various datasets.

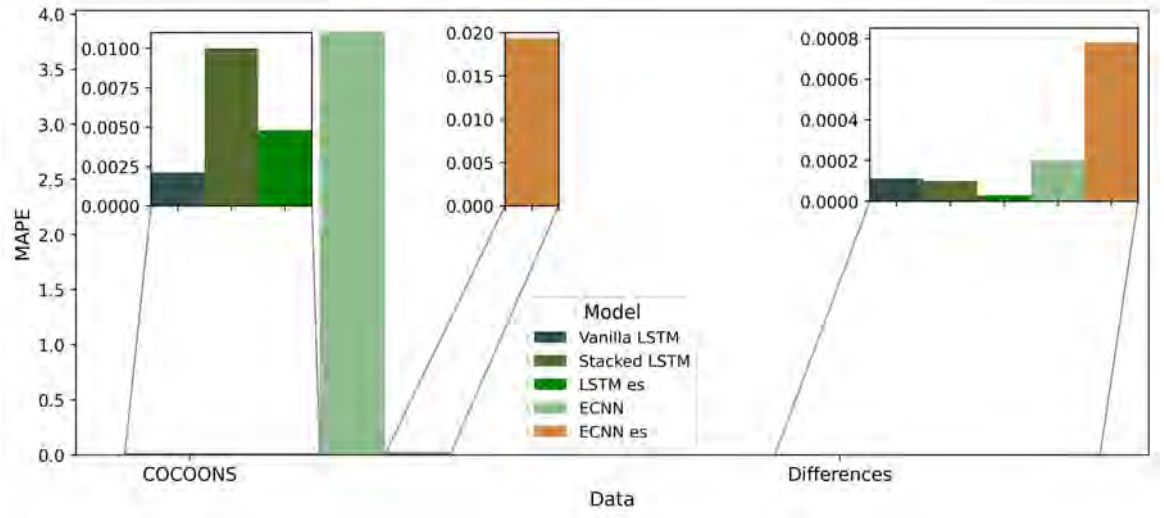


Figure 6.22: Mean Absolute Percentage Error of different models deployed on various datasets.

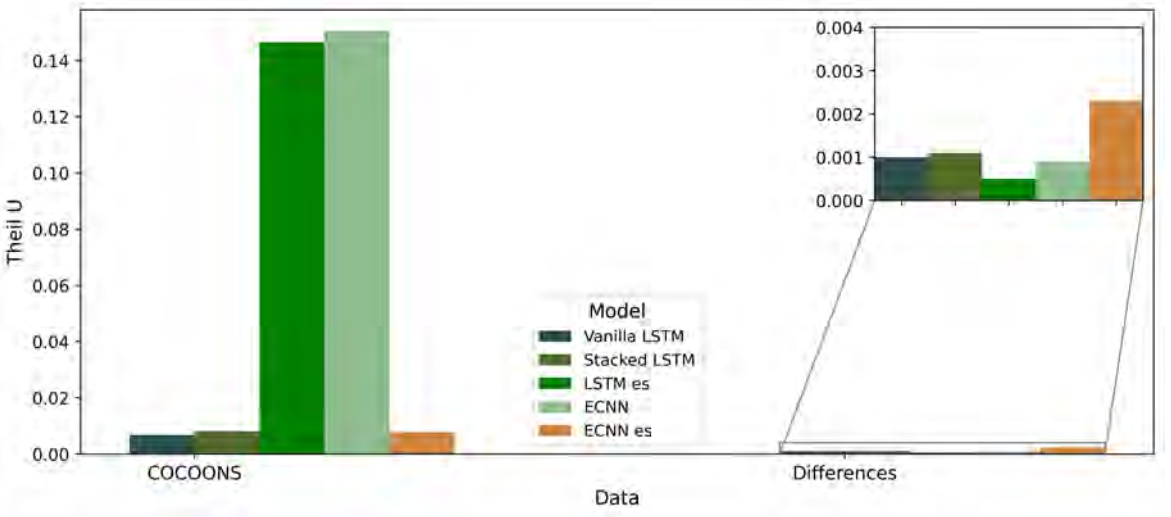


Figure 6.23: Theil U Coefficient of different models deployed on various datasets.

Notably, Figure 6.21 shows that the ECNN model trained on COCOON data performed significantly worse in terms of RMSE compared to other models. Its RMSE was nearly 100 times higher than the second worst performing model, the LSTM with exponential smoothing. Among all models, the LSTM with exponential smoothing achieved the lowest RMSE of 0.17, closely followed by the ECNN with exponential smoothing at 0.28.

Furthermore, Figure 6.22 demonstrates the MAPE for the models trained using different datasets, where it is evident that the ECNN trained on COCOON data exhibited the highest MAPE, surpassing all of the other models by a significant margin. On the other hand, the LSTM with exponential smoothing, trained using difference data, demonstrated the lowest MAPE among all models, with a MAPE score of 2.62×10^{-5}.

Moreover, the LSTM with exponential smoothing also resulted in the lowest Theil U coefficient, indicating its superior forecasting ability compared to naive guessing.

Applying exponential smoothing to the ECNN trained with COCOON data significantly improved its performance, leading to a drastic reduction of almost 5000 fold in the RMSE and nearly 2000 fold in the MAPE. Conversely, the ECNN trained with difference data experienced a rise in the MAPE, while seeing a decrease in the RMSE after exponential smoothing. This may be attributed to the fact that the RMSE is affected by the scale of the target variable, whereas the MAPE remains unaffected. The decrease in the RMSE indicates that the model's accuracy in predicting smaller values has improved, whereas the increase in the MAPE suggests a decline in accuracy for larger values. In the case of the LSTM model, implementing exponential smoothing resulted in higher accuracy when trained with difference data, resulting in a steep 5-fold decrease in RMSE. However, when trained with COCOON data, the RMSE increased by over 15 fold. Additionally, exponential smoothing led to decreased MAPE and Theil U coefficient for the LSTM with difference data, but applying it to the LSTM trained with COCOON data caused a significant increase in MAPE, RMSE, and Theil U coefficient.

6.4.1 Top Performing Models

The top 3 performing models were all trained using difference data. These models include the LSTM with exponential smoothing, the ECNN model with exponential smoothing, and the ECNN respectively. The models can be evaluated further by investigating the residuals. Since all of the top performing models were trained using difference data, they all have the same scale. Therefore the statistical analysis was conducted on the actual differences, without scaling them back to their COCOON representation. Table 6.8 shows a statistical summary of the residuals of these models.

Table 6.8: Statistical Summary of the Residuals of the Top 3 Models.

Model	**mean**	**standard deviation**
LSTM es	0.0251	0.1680
ECNN es	0.0738	0.22435
ECNN	-0.0038	0.8658

The residuals of the LSTM with exponential smoothing have a mean of 0.0251, indicating that on average, the model's predictions are close to the target values. Additionally, the residuals' standard deviation of 0.1680 suggests that the predictions from this model have a moderate amount of dispersion around the mean.

In the ECNN with exponential smoothing, the residuals have a higher mean compared to its LSTM counterpart, measuring at 0.0738. This indicates that, on average, the predictions from the ECNN with exponential smoothing may have a bias towards overestimating the target values. The higher standard deviation of 0.22435 also suggests that the predictions in this model have a wider dispersion around the mean value , compared to the LSTM. Furthermore, the ECNN has a negative mean residual value, indicating that, on average, the predictions in this model are an underestimate of the target values. Moreover, the standard deviation of 0.8658 indicates a large amount of dispersion in the predictions, suggesting that the predictions from the ECNN can have significant variability around the mean.

In terms of computational efficiency, the ECNN with exponential smoothing outperformed the other two models by a significant margin. While the LSTM with exponential smoothing was notably slower and more computationally demanding compared to the ECNN es, it still proved to be over four times faster than the ECNN without exponential smoothing. This is shown in Table 6.9.

Table 6.9: Model Runtime on Google Colab.

Model	**Time (s)**
LSTM es	941.77
ECNN es	29.99
ECNN	3986.75

CHAPTER 7

DISCUSSION AND CONCLUSION

7.1 INTRODUCTION

In this concluding chapter, a comprehensive analysis of the research is presented. The chapter begins with a brief overview of the study, followed by a discussion of the obtained results. Attention is given to the poor performance exhibited by the stacked LSTM and ECNN models trained using COCOON data. Furthermore, a comparison is made between the models trained using COCOON data and their difference data counterparts, highlighting the significant discrepancy in performance.

Additionally, the best machine learning model, developed through the application of exponential smoothing to a vanilla LSTM, is evaluated against a simple estimator. This comparison sheds light on the effectiveness of the proposed model and its potential benefits.

Furthermore, the implications of the research findings are discussed, taking into account any limitations encountered during the study. Recommendations for future research directions are then provided, aiming to address these limitations and further enhance the field of machine learning in number theory.

7.2 STUDY OVERVIEW

This research set out to develop two main machine learning algorithms to make COCOON predictions. The first phase of the research focused on extensively exploring the mathematical foundations and implications of a determinsitic scheme centred around COCOONs. The main results from this phase were used to obtain two datasets, the COCOON and difference data, which were used in the implementation phase. An extensive literature review was done on machine learning and deep learning techniques, with a particular focus on RNNs given the sequential nature of the data. This motivated the choice of the ECNN and LSTM. The implementation phase of the research involved using these two main deep learning algorithms, to develop models capable of predicting COCOONs. In addition to the COCOON data, these models were also trained and tested on the difference data. Different variations of the LSTM and ECNN models were also developed and tested. This included, stacked LSTM, LSTM with exponential smoothing, and ECNN with exponential smoothing.

7.3 PERFORMANCE INVESTIGATION

The addition of more layers to the vanilla LSTM model (stacked LSTM) forming a more complex network, did not prove beneficial and resulted in the second worst performing model, with a significantly high root mean squared error. The ECNN model trained on COCOON data performed even worse, and had a tendency of overfitting. When a model has high complexity, it is more prone to overfitting. Instead of learning the underlying patterns in the data and generalizing well, the network tends to memorise the training data. As a result, the ECNN performed poorly when encountering new, unseen data. This was tackled by simplifying the network structure of the model. While these modifications resulted in some improvement, the model still noticeably lagged behind the other models in terms of performance. However, when the ECNN was trained on difference data, the RMSE decreased significantly by over 99.9%. It is important to note that the difference predictions were scaled back to their COCOON representation before the RMSE was calculated, and therefore comparisons can be made between the two models as the errors have the same scale.

When models were trained with COCOON data, they often exhibited high inaccuracies, with the exception of directional accuracy, due to the strictly increasing nature of the data. On the other hand, the difference data was neither increasing nor decreasing, making it more challenging to correctly forecast the direction of each prediction. Therefore, models trained with difference data often had a lower directional accuracy than their COCOON counterparts. However, they had lower RMSE, MAPE, and Theil U coefficients than the models trained with COCOON data, indicating better accuracy overall. A possible reason for this lies in the simple structure of the difference data. All of the elements in the difference data, regardless of order, only differ from each other by a small amount. This helps to reduce the influence of outliers or irregular patterns that could otherwise affect the accuracy of the forecasts

Additionally, the effects of exponential smoothing varied depending on the model and the training data used. Exponential smoothing generally led to improvements in accuracy for the ECNN model with COCOON data and the LSTM model with difference data. However, the LSTM model trained with COCOON data showed a significant decrease in performance after applying expo-

nential smoothing. Specifically, the model exhibited an increased MAPE but decreased RMSE, highlighting the impact of scaling on these error metrics. Consequently, the decrease in RMSE implies increased accuracy for smaller values in the LSTM with exponential smoothing, while the increase in MAPE suggests a decline in accuracy for larger values in the model.

Finally, the results show that the most efficient model for making accurate COCOON predictions is the LSTM with exponential smoothing trained on difference data. This was due to exponential smoothing's ability to rapidly adapt to the changing distributions of each of the differences as the rank increased.

7.4 FINAL MODEL EVALUATION

The final, or best model, is evaluated by comparing it to a simpler approach, the standard deviation. Since the standard deviation can be thought of as the amount of variability naturally occurring in the predictions of the target values, it is the benchmark that any prediction model needs to beat. The standard deviation in the difference dataset is 0.42. This is equivalent to the error that would have been observed if the average difference was used as the prediction for all sequences. The top model's test RMSE was 0.08, significantly lower than the standard deviation of the dataset. Therefore, the model surpasses this benchmark, demonstrating the advantage of simplifying the COCOON data into a difference dataset when making COCOON predictions.

7.5 IMPLICATIONS OF RESEARCH

In addition to the models making COCOON predictions, they can also be used to enhance our understanding of the relationship between the input and output variables.

The results show that when predicting COCOONs, models trained using datasets consisting of COCOON differences outperform those utilizing COCOON data alone. Therefore, the use of difference data greatly aids in achieving more reliable predictions. Furthermore, the results show that machine learning models have the capability to surpass basic estimator models that rely solely on average calculations.

7.6 CONCLUDING REMARKS: LIMITATIONS AND FUTURE WORK

The LSTM with exponential smoothing trained on difference data serves as a fundamental framework for making COCOON predictions. By providing a sequence of n COCOONs, the model can accurately predict the $(n+1)^{th}$ COCOON. This capability could be particularly valuable in exploring prime numbers further, as COCOONs serve as a gateway towards primes. Furthermore, the model serves as a preliminary benchmark for future endeavours and improvements.

One of the main limitations of the best performing model, the LSTM with exponential smoothing, is computational time and scalability, due to the considerable amount of time required for its execution. The model already exhibits a noticeable delay during training and prediction phases. This time requirement increases even further when larger datasets are used. Consequently, this poses a significant challenge for scaling up the model to handle more extensive datasets.
On the contrary, the second best-performing model, the ECNN with exponential smoothing, exhibits exceptional computational efficiency, with a runtime that is 32 times faster than its LSTM counterpart. However, this enhanced efficiency comes at the expense of reduced accuracy, and a lower proportion of explained variation. The ECNN model with exponential smoothing explains 92.7% of the variation in the predictions, while its LSTM counterpart explains 95.8% of the variation. Additionally, the ECNN with exponential smoothing has a RMSE and MAPE almost double that of its LSTM counterpart.

In conclusion, the LSTM model's time constraints pose a hurdle in terms of scalability, while the ECNN model provides faster execution at the cost of slightly reduced accuracy and explanatory power. These findings suggest potential avenues for future research to optimize the trade-off between computational efficiency and model performance.

REFERENCES

Abdullah, D., Rahim, R., Apdilah, D., Efendi, S., Tulus, T. and Suwilo, S. (2018 03). Prime numbers comparison using sieve of eratosthenes and sieve of sundaram algorithm. *Journal of Physics: Conference Series*, vol. 978, p. 012123.

Aggarwal, K., Mijwil, M., Garg, S., Al-Mistarehi, A.-H., Alomari, S., Gök, M., Zein Alaabdin, A. and Abdul Rahman, S. (2022 01). Has the future started? the current growth of artificial intelligence, machine learning, and deep learning. *Iraqi Journal for Computer Science and Mathematics*, vol. 3, pp. 115–123.

Ahmadi, S., Beyhaghi, H., Blum, A. and Naggita, K. (2020). The strategic perceptron. *CoRR*, vol. abs/2008.01710. 2008.01710.
Available at: https://arxiv.org/abs/2008.01710

Amir, M., He, Y.-H., Lee, K.-H., Oliver, T. and Sultanow, E. (2022). Machine learning class numbers of real quadratic fields. 2209.09283.

Botchkarev, A. (2019). A new typology design of performance metrics to measure errors in machine learning regression algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management*, vol. 14, p. 045–076. ISSN 1555-1237.
Available at: http://dx.doi.org/10.28945/4184

Bufalo, D., Bufalo, M., Orlando, G. and Tetta, R. (2023 08). © a new algorithm to find prime numbers with less memory requirements. *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 26, pp. 1213–1236.

Ciampiconi, L., Elwood, A., Leonardi, M., Mohamed, A. and Rozza, A. (2023). A survey and taxonomy of loss functions in machine learning. 2301.05579.

Dawson, J.W. (2015). *The Fundamental Theorem of Arithmetic*, pp. 41–49. Springer International Publishing, Cham. ISBN 978-3-319-17368-9.
Available at: https://doi.org/10.1007/978-3-319-17368-9_6

Dubey, S.R., Singh, S.K. and Chaudhuri, B.B. (2021). A comprehensive survey and performance

analysis of activation functions in deep learning. *CoRR*, vol. abs/2109.14545. 2109.14545.
Available at: https://arxiv.org/abs/2109.14545

Ensor, K.B. and Glynn, P.W. (1997). Stochastic Optimization via Grid Search — web.stanford.edu. https://web.stanford.edu/~glynn/papers/1997/EnsorG97.html.

Fradkov, A.L. (2020). Early history of machine learning. *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 1385–1390. ISSN 2405-8963. 21st IFAC World Congress.
Available at: https://www.sciencedirect.com/science/article/pii/S2405896320325027

Gao, L. and Guan, L. (2023). Interpretability of machine learning: Recent advances and future prospects. 2305.00537.

Grant, R.E. and Ghannam, T. (2019). Accurate and infinite prime prediction from novel quasi-prime analytical methodology. 1903.08570.

Gui, J., Chen, T., Zhang, J., Cao, Q., Sun, Z., Luo, H. and Tao, D. (2023). A survey on self-supervised learning: Algorithms, applications, and future trends. 2301.05712.

Hayken, S. (2014). Neural networks and learning machines third edition. https://cours.etsmtl.ca/sys843/REFS/Books/ebook_Haykin09.pdf.

Helfgott, H. (2017 12). An improved sieve of eratosthenes. *Mathematics of Computation*, vol. 89.

Hu, X., Chu, L., Pei, J., Liu, W. and Bian, J. (2021). Model complexity of deep learning: a survey. *Knowledge and Information Systems*, vol. 63, pp. 2585 – 2619.
Available at: https://api.semanticscholar.org/CorpusID:232168493

Hu Zheng, Z.J. and Yun, G. (2021 01). Handling vanishing gradient problem using artificial derivative. *IEEE Access*, vol. PP, pp. 1–1.

Khairina, N. (2019 04). The comparison of methods for generating prime numbers between the sieve of eratosthenes, atkins, and sundaram. *SinkrOn*, vol. 3, p. 293.

Martino, G. (2013 01). A sieve for prime based on extension form of not prime. *American Journal of Computational Mathematics*, vol. 03, pp. 86–89.

Mokhtari, A., Ozdaglar, A. and Pattathil, S. (2020). Convergence rate of $\mathcal{O}(1/k)$ for optimistic gradient and extra-gradient methods in smooth convex-concave saddle point problems. 1906.01115.

Morley, S.K., Brito, T.V. and Welling, D.T. (2018). Measures of model performance based on the log accuracy ratio. *Space Weather*, vol. 16, no. 1, pp. 69–88. https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2017SW001669.
Available at: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2017SW001669

Mothebe, M.F. (2023). Sieve methods and the twin prime conjecture. 1909.02205.

Murugan, P. (2018). Learning the sequential temporal information with recurrent neural networks. 1807.02857.

Murugesan, R., Mishra, E. and Krishnan, A. (2021 07). Deep learning based models: Basic lstm, bi lstm, stacked lstm, cnn lstm and conv lstm to forecast agricultural commodities prices.

Mvubu, M., Kabuga, E., Plitz, C., Bah, B., Becker, R. and Zimmermann, H. (2020). On error correction neural networks for economic forecasting. *CoRR*, vol. abs/2004.05277. 2004.05277.
Available at: https://arxiv.org/abs/2004.05277

Nandutu, I., Atemkeng, M., Mgqatsa, N., Toadoum Sari, S., Okouma, P., Rockefeller, R., Ansah-Narh, T., Ebongue Kedieng Fendji, J.L. and Tchakounte, F. (2022). Error correction based deep neural networks for modeling and predicting south african wildlife & ndash;vehicle collision data. *Mathematics*, vol. 10, no. 21. ISSN 2227-7390.
Available at: https://www.mdpi.com/2227-7390/10/21/3988

Ning, E. and Kaeli, D. (2023). Memory efficient multithreaded incremental segmented sieve algorithm. 2310.17746.

Ostertagova, E. and Ostertag, O. (2012 12). Forecasting using simple exponential smoothing method. *Acta Electrotechnica et Informatica*, vol. 12, p. 62–66.

Pfeffermann, D. and Allon, J. (1989). Multivariate exponential smoothing: Method and practice. *International Journal of Forecasting*, vol. 5, no. 1, pp. 83–98. ISSN 0169-2070.
Available at: https://www.sciencedirect.com/science/article/pii/0169207089900666

Pylov, P., Maitak, R. and Protodyakonov, A. (2023 08). Approximation model based on lstm for predicting the next prime number in an infinite sequence. *E3S Web of Conferences*, vol. 413.

Rezk, N.M., Purnaprajna, M., Nordstrom, T. and Ul-Abdin, Z. (2020). Recurrent neural networks: An embedded computing perspective. *IEEE Access*, vol. 8, p. 57967–57996. ISSN 2169-3536. Available at: http://dx.doi.org/10.1109/ACCESS.2020.2982416

Ruder, S. (2017). An overview of gradient descent optimization algorithms. 1609.04747.

Schmidt, R.M. (2019). Recurrent neural networks (rnns): A gentle introduction and overview. 1912.05911.

Schroeder, M.R. (1984). *Primes*, pp. 26–39. Springer Berlin Heidelberg, Berlin, Heidelberg.

Schuld, M., Sinayskiy, I. and Petruccione, F. (2015). Simulating a perceptron on a quantum computer. *Physics Letters A*, vol. 379, no. 7, pp. 660–663. ISSN 0375-9601. Available at: https://www.sciencedirect.com/science/article/pii/S037596011401278X

Shiri, F.M., Perumal, T., Mustapha, N. and Mohamed, R. (2023). A comprehensive overview and comparative analysis on deep learning models: Cnn, rnn, lstm, gru. 2305.17473.

Staudemeyer, R.C. and Morris, E.R. (2019). Understanding lstm – a tutorial into long short-term memory recurrent neural networks. 1909.09586.

Stekel, A., Chkroun, M. and Azaria, A. (2018). Goldbach's function approximation using deep learning. 1803.09237.

Sutskever, I., Vinyals, O. and Le, Q.V. (2014). Sequence to sequence learning with neural networks. 1409.3215.

Tibaldi, S., Magnifico, G., Vodola, D. and Ercolessi, E. (2023 January). Unsupervised and supervised learning of interacting topological phases from single-particle correlation functions. *SciPost Physics*, vol. 14, no. 1. ISSN 2542-4653. Available at: http://dx.doi.org/10.21468/SciPostPhys.14.1.005

Tran-Dinh, Q. and van Dijk, M. (2022). Gradient descent-type methods: Background and simple unified convergence analysis. 2212.09413.

Waldo, J. (2022). A comparative study of back propagation and its alternatives on multilayer perceptrons. 2206.06098.

Willmott, C.J., Matsuura, K. and Robeson, S.M. (2009). Ambiguities inherent in sums-of-squares-based error statistics. *Atmospheric Environment*, vol. 43, pp. 749–752.
Available at: https://api.semanticscholar.org/CorpusID:7929783

Xing, S., Han, F. and Khoo, S. (2022). Extreme-long-short term memory for time-series prediction. 2210.08244.

Zaman, B.U. (2023 7). New Prime Number Theory.
Available at: https://www.techrxiv.org/articles/preprint/New_Prime_Number_Theory/23589381

Zelios, A., Grammenos, A., Maria Papatsimouli, N.A. and Fragulis, G. (2022). Recursive neural networks: recent results and applications. *SHS Web Conf.*, vol. 139, p. 03007.
Available at: https://doi.org/10.1051/shsconf/202213903007