Application of Artificial Neural Network Method for Solving Nonlinear Equations

Elias Abnew Azanaw

A Thesis Submitted to the Department of Applied Mathematics

School of Applied Natural Science

Presented in Partial Fulfillment of the Requirement for the Degree of

Master's in Applied Mathematics (Numerical Analysis)

Office of Graduate Studies

Adama Science and Technology University

July, 2023

Adama, Ethiopia

Application of Artificial Neural Network Method for Solving Nonlinear Equations

Elias Abnew Azanaw

Advisor: Dr. Tekle Gemechu Dinka

Co-Advisor: Dr. Mesfin Mekuria Woldaregay

A Thesis Submitted to the Department of Applied Mathematics

School of Applied Natural Science

Office of Graduate Studies

Adama Science and Technology University

July, 2023

Adama, Ethiopia

Declaration

I declare that this thesis entitled ***"Application of Artificial Neural Network Method for Solving Non linear Equations"*** is my own work and has not been submitted to any university for similar purpose. The references used in this thesis are duly recognized by proper citations.

Name of Student	Signature	Date

Recommendation

I, the advisor of this research thesis, hereby certify that I have closely advised the student while developing this thesis and read the draft thesis entitled ***"Application of Artificial Neural Network Method for Solving Non linear Equations"*** prepared under my guidance by **Elias Abnew Azanaw**. Therefore, I recommend the submission of the thesis to the department for further review and evaluation

Major Advisor	Signature	Date
Co-advisor	Signature	Date

Approval Page

We hereby certify that the recommendation and suggestion given by the thesis review committee are appropriately incorporated into the final thesis entitled ***"Application of Artificial Neural Network Method for Solving Non linear Equations"*** by **Elias Abnew Azanaw**.

Major Advisor	Signature	Date
Co-advisor	Signature	Date

We, the undersigned, members of the Board of Reviewers of the thesis open defense by **Elias Abnew Azanaw** have read and evaluated the thesis entitled ***"Application of Artificial Neural Network Method for Solving Non linear Equations"*** and assessed the understanding of the candidate about the thesis research. This is, therefore, to certify that the thesis is accepted and we recommend the implementation of the thesis.

Chairperson	Signature	Date
Internal Examiner	Signature	Date
External Examiner	Signature	Date

Final approval and acceptance of the thesis is contingent upon submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

Department Head	Signature	Date
School Dean	Signature	Date
Office of Postgraduate Studies, Dean	Signature	Date

ACKNOWLEDGMENTS

First of all I would like to thank to GOD for giving the strength from lower grade up to now with great accomplishment to all the necessary things. Secondly, I would like to say thanks to my advisor Dr. Tekle Gemechu who helped me by giving constructive suggestion, references and the work. I would also like to thank Dr Mesfin Mekuria for providing advice and comments regarding analysis. Finally, many thanks to all participants that took part in the study and enabled this thesis to be possible.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

AE	Aberth Ehrlich method
AI	Artificial intelligence
RNN	Recurrent Neural Network
LSTM	Long short-term memory
GRU	Gated recurrent unit
FNN	Feed forward neural network
CLA	Constrained learning algorithm
BPA	Back propagation algorithm
BPTT	Back propagation through time
CNN	Convolutional neural network
RTRL	Real-Time Recurrent Learning
ANN	Artificial neural network

Abstract

In this thesis, we focused on the applications of Artificial neural network method and Aberth Ehrlich iterative method for solving nonlinear polynomial equations of arbitrary degree n. We studied Artificial neural network method for solving polynomials. We started by training the Artificial neural networks using the Long short term memory algorithm and as input the coefficients of the polynomials considered. We used feedback neural network and supervised learning method to update the network parameters. Two activation functions (Sigmoid activation function and hyperbolic tangent activation function) were used in this work. We used mean square error as loss function in order to update the weight and results were then compared in terms of accuracy and efficiency with the Aberth Ehrlich method. The comparisons are mainly based on their mean square error values and running time. The basic python code is used for implementation of the results in tables and graphs.

Keywords: Polynomial roots, artificial neural networks, long short-term memory, mean square error, Aberth method.

Chapter One

Introduction

1.1 Background of the Study

The problem of determining polynomial zeros has a great importance in science and engineering theory and practice as well. The theory of control, digital signal processing, system stability, analysis of transfer functions, various mathematical models, differential and difference equations, and eigenvalue problems are a few examples (Petković & Rančić, 2006).

For such reasons, finding the solution of nonlinear polynomial equations in the different branches of Science and Engineering has led to the development of many different methods for their numerical computations. There is also a significant interest in the development of new and efficient numerical iterative methods for determining the zeros of polynomials. These methods can approximate the zeros of a polynomial either in a sequential or simultaneous manner (Freitas et al., 2018) .

The basic conventional methods for finding a zero of a univariate polynomial generally involve making an initial guess at a single zero then iterating an algorithm which makes successively better approximations to the zero until a desired accuracy is reached. If more than one zero is required then the zero already obtained is used to deflate or reduce the order of the polynomial and the iteration process is restarted using the deflated polynomial to find another zero. Iteration methods which estimate one zero at a time include the well known Secant algorithm and Newton's method .

Although there are many iterative methods to calculate one real zero or a pair of complex conjugate zeros of a polynomial, such as the well known Laguerre's and Jenkins–Traub's methods the determination of all zeros of a given polynomial by one of such methods involves repeated deflation's, which can lead to very inaccurate results due to the problem of accumulating rounding errors when using finite accuracy floating point arithmetic.

The simultaneous approaches for polynomial zeros, on the other hand, benefit from being naturally parallel and avoid deflation. For the purpose of approximating polynomial zeros, several concurrent simultaneous iterative techniques exist.

The well-known Ehrlich-Aberth method, Chebyshev's method, and Durant-Kernel method are a few examples. These concurrent iterative algorithms, however, need accurate initial predictions to converge (Freitas et al., 2021).

Recently, a number of artificial neural network (ANN) methods have been developed to address a variety of real-world problems.

Artificial neural network (ANN) is one of the major fields of artificial intelligence (AI) computing model based on the organizational structure of the human brain (Chakraverty & Mall, 2017). The simplest definition of ANN is provided by the inventor of one of the first neuro computers, Dr. Robert Hecht-Nielsen in 1938. A neural network, according to him, is "a computing system composed of numerous simple, highly interdependent processing elements, which process information by their dynamic state response to external inputs." ANNs are processing tools (algorithms) that are modeled after the neuronal structure of the mammalian cerebral cortex but on much smaller scales. The human brain serves as a source of inspiration for computer scientist. Logician Walter Pitts and neuroscientist Warren S McCulloch created the first conceptual model of ANN (McCulloch & Pitts, 1943). In their article, they described the idea of a neuron, which is a single cell that is part of a network of cells and takes inputs, analyzes those inputs, and produces an output.

The concept behind ANNs is based on the theory that silicon and wires can be utilized to mimic the actual neurons and dendrites seen in the human brain by making the right connections. There are many billion neurons, or nerve cells, in the human brain. They are connected to a million more cells through axons. Dendrites take in information from sensory organs as well as external inputs. These inputs generate electric impulses, which quickly move through the brain network. A neuron can then communicate the issue to other neurons for resolution, as seen in the diagram below (Sejnowski & Delbruck, 2012).

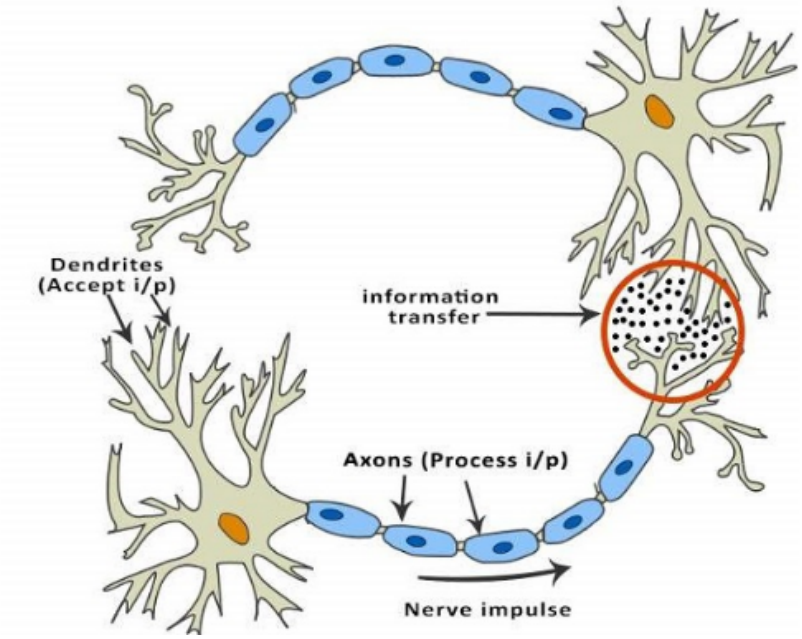


Figure 1.1: Structure of biological neural networks
This figure adopted from(A. K. Jain, Mao, & Mohiuddin, 1996).

An artificial neuron can be considered as a mathematical representation of a biological neuron. Through dendrites (tiny fibers), a biological neuron receives input messages from other neurons. Similar to this, a Perceptron gets its information from other perceptions via input neurons that accept numeric input. Synapses are the places of connection between dendrites and biological neurons. Likewise, Weights refer to the links between inputs and perceptions. Each input's level of importance is measured. In a biological neuron, the dendrite signals are used by the nucleus to generate an output signal. In a similar manner, the central region of a Perceptron makes calculations depending on input values and generates an output (Pramoditha, 2021).

The Artificial Neural Network (ANN) is a deep learning method that arose from the concept of the human brain (Biological Neural Networks). The development of ANN was the result of an attempt to replicate the workings of the human brain. The workings of ANN are extremely similar to those of biological neural networks, although they are not identical (Shanmuganathan, 2016).

ANN is a tool for data modeling that depends on a variety of variables and learning strategies. Typically, layers are used to structure neural networks. They are composed of numerous interconnected neurons/nodes,which have activation functions. ANN processes information through neurons/nodes in a parallel manner to address particular issues for a new testing input signal value. Patterns are communicated to the network via the input layer, which then connects to one or more hidden layers, where the real processing is carried out via a system of weighted connections. As seen in figure 1.2, the hidden layers then connect to an output layer, where the solution is output. Here, input nodes are indicated by A_j, weights from the input layer to the hidden layer are shown by M_{ij}, weights from the hidden layer to an output layer are indicated by H_i, and the output node is indicated by Y.

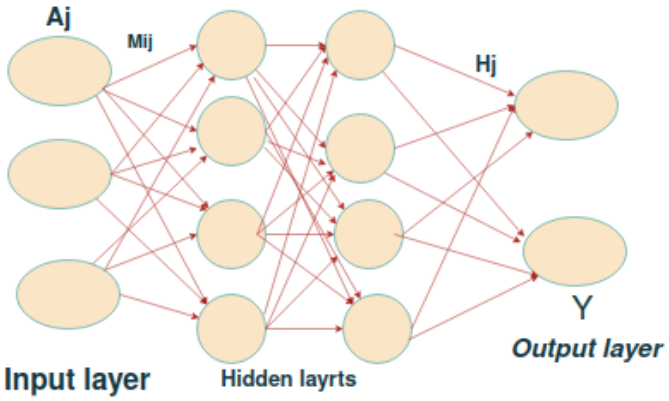


Figure 1.2: Structure of artificial neural network
Source:(Chakraverty & Mall, 2017)

There are several kinds of artificial neural networks. These types of networks are implemented based on the mathematical operations and a set of parameters required to determine the output. Among them Feed Forward Neural Network, Multilayer Perceptron, Convolutional Neural Network, Radial Basis Functional Neural Network, Recurrent Neural Network are some examples (Team, 2020). I n this thesis, we have considered Recurrent Neural Network from different types of artificial neural network.

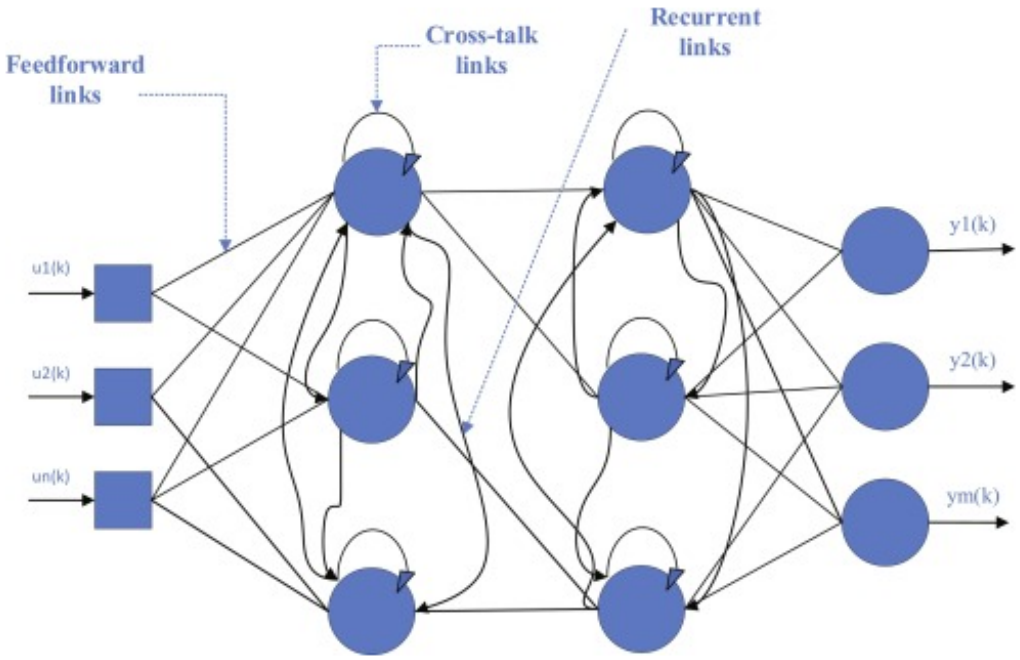


Figure 1.3: Structure of Recurrent neural networks
source:(A. Jain, Zamir, Savarese, & Saxena, 2016).

Recurrent neural networks are, as the name suggests, neural network models that have some type of internal looping structure that simulates an information flow. By allowing the network to maintain a state that depends on the input, this enables the network to properly process sequential data. It is a type of artificial neural network that utilizes sequential data or time series data. RNNs can memorize previous inputs due to their internal memory.

Due to their internal memory, RNNs are able to recall past inputs. The input and output of a typical neural networks are independent of one another, hence they unable to remember the previous inputs. However, because of its internal memory a recurrent neural network can recall those characters. It generates output, duplicates it, and feeds the copy back into the network. RNNs are equipped with a "memory" that enables them to recall data from earlier time steps and generate predictions based on long-term dependencies in the data. Therefore, one of a recurrent neural network's most important advantages is that it can simulate a simple type of memory.

RNNs have been applied to a wide range of tasks, such as speech recognition, language translation, and time series forecasting. Since they can analyze the input until they reach the output of the sequence, they are particularly useful for handling variable-length input sequences. RNNs take on a variety of forms, such as long short-term memory (LSTM) and gated recurrent unit (GRU) networks, which are created to overcome some of the difficulties associated with training ordinary RNNs, like the vanishing gradient problem (Lukoševičius & Jaeger, 2009).

LSTM is a type of recurrent neural network (RNN) that is designed to deal with the vanishing gradient problem. This issue can arise during the training of RNNs, as the gradients are getting smaller through successive iterations, causing it to be difficult for the network to learn from input. By utilizing gates, LSTM networks are able to overcome this problem. As a result, they improved long-term knowledge and can be trained on longer sequences of data (Pal, Ghosh, & Nag, 2018). Therefor,it is the main reason of using LSTM networks for approximating roots of polynomial equation.

1.2 Statement of the Problem

We consider n degree non linear equation of the form:

$$G_n(x) = d_n x^n + d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + ... + d_1 x + d_0 = 0 \tag{1.1}$$

where n is an integer. It is impossible to determine the root of an equation analytically in many situations. In addition, there is no algebraic formulas can be given for roots of nonlinear equations that have degree greater than or equal to five. when these methods are not successful, we use the concept of numerical methods. Numerical methods are used to approximate solutions of equations when exact solutions can not be determined via algebraic methods. By using the methods of iterative root finder and Artificial Neural Network based approach this study intended to answer the following basic questions;

- How to get approximate solutions of nonlinear polynomials using ANN?
- How to compare numerical iterative methods and ANN method for the solution of nonlinear polynomials?
- What is the advantage of using ANN method (particularly LSTM algorithm)?

1.3 Objectives

1.3.1 General Objective

The general objective of this study is applying ANN method for solving the roots of nonlinear polynomials.

1.3.2 Specific Objectives

The specific objective of this study are to:

- apply artificial neural network method for approximate solution of polynomials.
- compare numerical iterative methods and artificial neural network method solutions for non linear polynomials.
- study the advantage of using ANN (particularly LSTM algorithm) for solving nonlinear polynomials.

1.4 Significance of the Study

The result obtained from this study can be

- used as reference material for scholars who works on this research area.
- used to apply the ANN method for solving roots of nonlinear polynomials.

1.5 Expected Outcomes of the Study

The expected outcomes of this study is to Understand how to approximate roots of nonlinear polynomials in context of Artificial neural network (RNN,particularly LSTM). Finally, the results will be written in the form of an MSC thesis.

1.6 Delimitation of the Study

This study is delimited to the problems of nonlinear polynomials of the form:

$$G_n(x) = d_n x^n + d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + ... + d_1 x + d_0 = 0 \tag{1.2}$$

where $d_n \neq 0$, $a = d_{n-1}$, d_{n-2} , . . ., d_0 are constants, and n is a positive integer and represents an arbitrary degree of a polynomials. Our work is limited for the selective degrees of polynomials roots in context of ANN.

Chapter Two

Review of Related Works

In this section we review Some literature s of ANN method for roots of polynomials. Hormis et al. (1995) presented the ANN method with the objective of separating a two dimensional polynomial into two one dimensional polynomials with the same degree. It is the first work in finding the roots of a given polynomial with ANN-based approaches.

D.-S. Huang et al. (2003) developed a dilatation approach to change the locations of the roots in space, pulling each root farther apart. Additionally, they proposed a technique for locating close arbitrary roots of polynomials that aimed to enlarge the gap between them so that ANNs could locate them more quickly. As a result, a root-finder can quickly and effectively resolve those nearby (and complicated) roots. Also, a sophisticated version of the constrained learning algorithm is derived in their study. In addition, they used a feed forward neural network (FNN) root-finder to solve the problem of finding the roots.

To determine the arbitrary roots (even complex ones) of arbitrary polynomials, a novel neural root finder based on the root moment approach was proposed. Based on feed forward neural networks (FNN), a constrained learning algorithm (CLA) was used to train this neural root finder (D.-S. Huang, Ip, & Chi, 2004). To create a new CLA, they specifically integrated the a prior knowledge of the root moments of polynomials into the traditional back propagation algorithm (BPA). It is demonstrated that the resulting NRF can quickly estimate the distributions of polynomial roots.

D. Huang and Chi (2004) proposed a novel recursive partitioning method based on constrained learning neural networks for the purpose of determining any number of (real or complex) roots of any arbitrary polynomial, with a value below the order of the polynomials. Additionally, their paper provides a BP network constrained learning algorithm (CLA) for root-finders that is based on the restricted relationships between polynomial roots and coefficients. Their experimental findings show that, compared to conventional root-finding techniques, their method can more quickly and successfully discover the roots of any high order polynomials with higher precision.

Mourrain et al. (2006) applied a Feed forward Neural Network (FNN) to investigate how many real roots polynomials actually had. The results showed that ANNs were highly accurate at executing this task (Zhang, Zhu, & Hu, 2008) found the method for calculating the multiplicities of roots is to determine the number of different real or complex roots.

Zhang et al. (2008) proposed a novel neural network method for solving polynomials with multiple real roots. The symbolic method and the numerical method are combined in this technique. The number and multiplicities of the various real roots of polynomials can be directly determined using the full discriminating system of the polynomial. A neural network model is built to identify multiple real roots of polynomials based on the number of distinct real roots.

Das and Seal (2012) offered a completely alternative method employing a group of polynomial coefficients that are divisors, denoted by the letter D. This viewpoint states that r is a potential polynomial root if it divides D and the coefficient of the lowest degree term. The coefficients of the polynomial under consideration are fed into an ANN by the input neurons, and the first hidden neuron in the first hidden layer divides the coefficients list into two. Each list provides input to the two hidden neurons in the second hidden layer. The following hidden layer's next two hidden neurons are tasked with optimizing the roots using a learning algorithm after receiving the coefficients of these two nodes that are also divisors of the coefficient of the highest degree term.

Andoni et al. (2014) investigated how well the gradient descent method for learning low degree polynomials in neural networks worked. They investigated that, although gradient descent has been frequently utilized in practice to learn neural networks and neural networks have been shown to have high expressive power, there are few known theoretical guarantees for such techniques.

Freitas et al. (2018) adopted a different approach based on neural networks for finding the zeros of real polynomials with only real zeros. This approach is tested with random polynomials of different degrees. The results they obtained , although preliminary and limited, indicate that their approach seems to be quite robust and promising, and faster when compared with the well known Durand–Kerner method.

Chapter Three

Methodology

3.1 Study Design

This study applied both the documentation review and numerical experimentation.

3.2 Mathematical Procedures

To attain the objective of the study, we followed the following steps.

- Describing the problem.
- Describing the numerical method (AE) to find roots of polynomial equation.
- Describing ANN (LSTM) method for the problem.
- Generating a synthetic dataset of polynomial coefficients and roots to train and test our polynomial models.
- Splitting the dataset randomly into training and testing sets using $4:1$ ratio respectively.
- Training the ANNs using the LSTM algorithm.
- Comparing results in terms of accuracy and efficacy to the AE method.

Chapter Four

Preliminaries

In this chapter, we present the definition of non linear polynomials, and the components, architecture, learning methods and learning rules of artificial neural networks. We also discussed the definitions and types of some activation functions.

4.1 Nonlinear Equations

A polynomial expression is an algebraic expression that has one or more terms with nonnegative integral variable exponents. A polynomial in x of degree n, where $n \geq 0$ is an integer, is an expression of the form

$$G_n(x) = d_n x^n + d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + ... + d_1 x + d_0, \tag{4.1}$$

where $d_n \neq 0$,d_{n-1},...,d_0 are constants.
When $G_n(x)$ is set equal to zero, the resulting equation

$$G_n(x) = d_n x^n + d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + ... + d_1 x + d_0 = 0 \tag{4.2}$$

is called polynomial equation of degree n .

If the polynomial $G_n(x)$ contains functions of the kind, such as trigonometric, logarithmic, exponential etc., then $G_n(x) = 0$ is called a transcendental equation (Jeswal & Chakraverty, 2018). There are numerous solutions of nonlinear equations that can be solved, and there are different methods for doing so depending on the type of solution (real or complex, rational or irrational). We call values of x that satisfy equation (1.2) roots or solutions of the equation. They are also called zeros of the polynomial $G_n(x)$. When $n = 1$, is called a linear equation (or equation of degree 1).

$$d_1 x + d_0 = 0 \tag{4.3}$$

Its only solution is x $= \frac{-d_0}{d_1}$. A linear equation is an algebraic equation where each term has

an exponent of 1 and when this equation is graphed, it always results in a straight line. It is also known as a one degree equation. A linear equation may have more than one variable. The standard form of a linear equation in one variable is of the form $Ax+B=0$. Here, x is a variable, $A \neq 0$ is a coefficient and B is constant. The standard form of a linear equation in two variables is of the form $Ax+By=C$. Here, x and y are variables, $A \neq 0$ and $B \neq 0$ are coefficients and C is a constant.

The equation in any problem cannot be written as a linear combination of unknown variables or functions for a nonlinear system . Quadratic equations (equations of degree 2) are obtained when $n=2$. It is customary in this case to denote coefficients as follows

$$d_2x^2+d_1x+d_0=0 \tag{4.4}$$

The next simplest polynomial equation after linear and quadratic is the cubic,

$$d_3x^3+d_2x^2+d_1x+d_0=0 \tag{4.5}$$

and after that the quartic,

$$d_4x^4+d_3x^3+d_2x^2+d_1x+d_0=0 \tag{4.6}$$

A numerical value of x that satisfies the equation is a solution to the equation or called the root of the equation. There are procedures that give roots for all of these equations. There are various algebraic methods to solve non linear equations. These methods include the quadratic formula, cardan's method,descartes' method e.t.c (Barbeau, 2003). However it is impossible to determine the root of an equation analytically in many situations. In addition,there is no algebraic formulas can be given for roots of nonlinear equations that have degree greater than or equal to five.

when these methods are not successful, we use the concept of numerical methods. Numerical methods are used to approximate solutions of equations when exact solutions can not be determined via algebraic methods. A numerical solution of an equation is a value of x that satisfies the equation approximately. For such equations, it is usually necessary to use numerical methods to find roots(McNamee & Pan, 2013).

4.2 Artificial Neural Network

It is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy (Chartrand et al., 2017).

4.2.1 Components of Artificial Neural Network

Artificial neural network is usually a computational network based on biological neural networks its structure is constructed with the following components .

Neurons(nodes)

A node is a location where information or communication can enter or set off in a network. They used to represent data point and the relationships between them are represented by the connections between the nodes. Nodes can be linked together in a wide variety of ways, and the weights of the connections between them can vary depending on how strongly the data pieces they represent are related to one another. Each node receives several inputs from other nodes via connections that have corresponding weights, which are similar to the synapse's strength. The node activates and sends the signal through a transfer function, and then transfers it to nearby nodes when the weighted total of inputs exceeds the threshold value (Zou et al., 2009).

Input layer

The input layer is composed of nodes that brings in the initial data. Most of the time, inputs come in the form of vectors, where each block of the vector is fed into every node of the input layer. The inputs/information from the outside world are given to the model in this layer so it can learn from them and draw conclusions. A distribution framework for the data being presented to the network is provided by an input layer. At this layer, no processing is carried out. The input layer is followed by one or more real processing layers, and inputs/information

pass to them i.e Hidden layer (Paola & Schowengerdt, 1995).

Hidden layer

The hidden layer is the layers between the input layer and the output layer. It uses the weighted nodes as its input and an activation function to generate an output. This is the layer where the actual learning takes place. Hidden layer is the set of neurons where all the computations are performed on the input data. A neural network might have countless hidden layers.

Output layer

It is the last layer of a neural network. The output layer is the output/conclusions of the model derived from all the computations performed. There can be single or multiple nodes in the output layer. If we have a binary classification problem the output node is 1 but in the case of multi-class classification, the output nodes can be more than 1.

Parameters

A list of numbers that specifies the internal structure of the neural network. There are two types of parameters these are model parameters and hyper parameters. Model parameters are internal configuration variables that the model learns by itself. These are specified or estimated while training the model. They are dependent on the training dataset. The values of parameters can be estimated by the optimization algorithms, such as Gradient Descent and Adams algorithm. The model's performance on unobserved data is determined by the final parameters computed after training. Some examples of model parameters include weights and biases. Weights and bias are both learnable parameters inside the network. A teachable neural network will randomize both the weight and bias values before learning initially begins. As training continues, both parameters are adjusted toward the desired values and the correct output.

Where as Hyper parameters are the explicitly specified parameters that regulate the training process and are crucial for the model's optimization. These are set before the beginning of the training of the model. The model's quality is determined by the chosen or adjusted hyper parameters. Hyper parameters are the variables that affect how the network is trained as well as the variables that decide the network's structure. The learning rate for a

neural network's training, the number of epochs, batch size, and other parameters are some examples of model hyper parameters.

Summation Function

A function that combines the various input activations into a single activation.

4.2.2 Architectures of ANN

ANN is a part of Machine Learning and also very crucial because its structure is similar to the human brain. The structure of the information processing system is the network's essential component. ANN processes information and function is similar to the manner in which the human brain does by sending neural signals from one end to the other. The network is made up of numerous, deeply connected processing units called neurons that collaborate to address a particular issue simultaneously (Chakraverty & Mall, 2017). Different classes of neural network architecture can be recognized based on the structure of these interconnections, as will be explained more below.

Feed-Forward Neural Network

Neurons are organized in layers in a feed-forward neural network. A layer's neurons use data or input from the layer before as input and send their output to the layer next to them. There can be no network connections to the same layer or a layer before it. Here, data only flows in a feed-forward fashion from the input neuron to the output neuron. Because there are no feedback loops, the output of one layer does not affect another layer.

Feedback Neural Network

The output of one layer in a feedback neural network travels back to the first layer. By introducing loops to the network, this network can have signals moving in both directions. This network is highly strong and can occasionally become very convoluted. Neurons can link in any way that makes sense. When solving optimization problems, feedback neural networks can be used to find the most effective arrangement of interrelated factors. They are dynamic, and until they arrive at equilibrium, their state will alter continuously .

4.2.3 Learning Methods of ANN

Learning in artificial neural networks refers to the process of adjusting the weights of connections between specific network nodes.The way that a parameter is changed determines the type of learning that occurs. The learning situations in neural networks may be classified into different types .The followings are some types of ANN learning methods.

Supervised Learning or Associative Learning

Associative learning, also known as supervised learning, works by giving input and matching output patterns to a network. These input-output pairings may be provided by the neural network's system or by an external trainer. After processing the inputs, the network compares the obtained outputs to the desired outputs. To identify the inaccuracy, a comparison is done between the network's computed output and the predicted output that has been corrected. The network parameters can then be altered in response to the fault, improving performance. In other words, it is presumed that inputs come first and outputs come last in the causal chain. Mediating variables between inputs and outputs can be included in models.

Unsupervised or Self-Organization Learning

A unit (the output) learns how to respond to sets of patterns in the input through a process known as unsupervised learning or self-organization. Unsupervised training involves giving the network inputs but not the expected results. The system must next decide which features it will use in order to group the supplied data into categories. This is frequently referred to as adaptation or self-organization (McClelland et al., 1986).

Reinforcement learning

An artificial neural network's parameters are adjusted using the machine learning technique known as reinforcement learning, where data is typically generated by interactions with the environment rather than being provided. This sort of learning can be thought of as a middle ground between the first two forms. In this case, the learning machine interacts with the environment and receives input from it. Based on the response from the environment, the learning system determines whether an activity is beneficial (rewarding) or poor (punishable), and it alters its parameters accordingly. Typically, parameter adjustments are

carried out repeatedly until an equilibrium condition is reached, at which point they stop completely (Dongare, Kharde, Kachare, et al., 2012).

4.2.4 Learning Rules or Learning Processes

It is technique or a mathematical theory It is applied throughout the network, which enhances the performance of the ANN. As a result, when a network simulates in a particular data environment, learning rules update the weights and bias levels of the network. The process of applying learning rules is iterative. A neural network can gain knowledge from the state of the world and enhance its performance. Every neural network has knowledge that is represented in the connection weight values. The weights of the connections in most kinds of ANNs are adjusted in accordance with the input patterns.

There are many different kinds of learning rules for ANN, For example:Error back propagation learning algorithm or delta learning rule, Hebbian learning rule, Perceptron learning rule, Widrow–Hoff learning rule, Winner-Take-All learning rule, etc.

Error Back-Propagation Learning Algorithm or Delta Learning Rule

Error back-propagation learning algorithm has been introduced by Rumelhart et al. It is also known as the delta learning rule and is one of the most commonly used learning rule. Back propagation algorithm is one of the well-known algorithms in neural networks. This learning algorithm is valid for continuous activation function and is used in the supervised/ unsupervised training method (McClelland et al., 1986).

By evaluating the partial derivative of the network's error with respect to each of its weights, we can establish the network's error flow direction. If we take the negative derivative and add it to the weights, the error will drop until it gets close to a local minimum. If the derivative is negative, we must either add a negative value to the weight or perform the opposite. Then, starting from the weights of output layer, continuing to the weights of hidden layer , and then returning to the weights of input layer weights, each of the weights is subjected to these partial derivatives. In general, bringing samples into the network as input vectors, determining the output layer's error, and then modifying the network's weights to minimize the error are the steps involved in training the network. To simplify the derivative, the average of all squared errors E for the outputs is determined. The weights can be updated one at a time after the error has been calculated. (Chakraverty & Mall, 2017): $\bigtriangledown E$

for the network's training, By randomly initializing the weights $(w_i)$and bias(b) in this learning algorithm, we can update them as follows:

$$w_{ij}{}^{n+1} = w_{ij}{}^{n} + \triangle w_{ij}{}^{n}$$
$$b_{ki}{}^{n+1} = b_{ki}{}^{n} + \triangle b_{ki}{}^{n}$$

4.2.5 Activation Functions

In ANNs, activation functions are utilized specifically to convert signal inputs into output signals that are then send the input to the following layer (Sharma et al., 2017). They are functions that operate on the net (input) to obtain the output of the network. The activation function decides whether a neuron should be activated by creating a weighted sum and adding bias to it. By applying basic mathematical procedures, it will determine whether or not the neuron's input to the network is significant during the prediction process. It functions as a squashing function, forcing the neural network's output to fall within a specified range (often between 0 and 1, or -1, and 1,) The common types of activation function are: Unit step (threshold) function ,linear function , Sigmoid function ,Tangent hyperbolic function , ReLU (Rectified Linear Unit) Activation Function

Sigmoid activation function

The sigmoid activation function is also called the logistic function. It is a non-linear function, it is the most commonly used activation function . It takes any real value as input and outputs values in the range 0 to 1.
This function is calculated as follows:

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.7}$$

Where e is a mathematical constant, which is the base of the natural logarithm.

Tanh Activation Function

The hyperbolic tangent activation function is also referred to simply as the tanh function. Similar to the sigmoid function, the function of tanh is symmetric at the origin. Tanh function's values occur between -1 and 1, and it is both continuous and differentiable. As a result, the outputs from previous layers that will be applied as input for the next one will

display different signs. Tanh prevails over sigmoid function because it is zero-centered and are not restricted to vary in a certain direction. The larger the input is the closer the output value will be to 1, whereas the smaller the input is the closer the output will be to -1. The slope of the tanh function is steeper when compared to that of the sigmoid function. The Tanh activation function is calculated as follows:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.8}$$

Where e is a mathematical constant, which is the base of the natural logarithm.

ReLU Activation Function

The rectified linear activation function, or ReLU activation function, is the most common function used for hidden layers. It is more effective than other functions because only a small number of the neurons are activated at once, rather a all of them activated at the same time. This can be written as:

$$f(z) = max(0, z) \tag{4.9}$$

This function has the disadvantage of making every negative value exactly equal to zero, which hinders the model's ability to properly match or learn from the data. As a result, every negative input to the activation function of ReLU just become zero in the graph, which affects the final graph by wrongly mapping the values that are negative (Sharma et al., 2017).

Chapter Five

Result and Discussion

5.1 The numerical method

The Aberth method, also known as the Ehrlich-Aberth method or the Aberth-Ehrlich method developed by Oliver Aberth and Louis W. Ehrlich to estimate simultaneous roots of a uni variate polynomial (Fatheddin & Sajadian, 2022).

For the derivation of the AE algorithm, we start from Newton's method by considering a uni variate n^{th} order polynomial of the form:

$$p_n(x) = b_n y^n + b_{n-1} y^{n-1} + b_{n-2} y^{n-2} + ... + b_1 y + b_0 = 0 \tag{5.1}$$

In which the real or complex coefficients are indicated by b_i. Then, since a polynomial with n degree needs n different roots, we now assume n distinct roots in the following manners: $s_1^{(0)}$,$s_2^{(0)}$, $s_3^{(0)}$,..., $s_n^{(0)}$.

The Newton's method of iteration is given by:

$$s_i^{(k+1)} = s_i^{(k)} - \frac{G(s_i^{(k)})}{G'(s_i^{(k)})} \tag{5.2}$$

G is a polynomial function in this instance. Now assume:

$$G(x) = \frac{p_n(x)}{\prod_{j=1, j\neq j}^{n}(x - s_j^{(k)})} \tag{5.3}$$

Instead of just writing $F(x) = p_n(x)$,then:

$$\begin{aligned} \frac{G'(x)}{G(x)} = \frac{d}{dx} ln|G(x)| = \frac{d}{dx}[ln|p_n(x)| - \sum_{j=1, j\neq k}^{n} ln(|x - s_j^{(k)}|)] \\ = \frac{p_n'(x)}{p_n(x)} - \sum_{j=1, j\neq k}^{n} \frac{1}{(x - s_j^{(k)})} \end{aligned} \tag{5.4}$$

We now have:

$$\frac{G(x)}{G'(x)} = \frac{1}{\frac{p_n'(x)}{p_n(x)} - \frac{1}{\sum_{j=1,j\neq k}^{n}(x-s_j^{(k)})}} \tag{5.5}$$

If we insert this equation into the Newton's iteration (equation 5.2),we obtain the AE algorithm:

$$s_i^{(k+1)} = s_i^{(k)} - \frac{\alpha_i^{(k)}}{1-\alpha_i^{(k)}\beta_i^{(k)}} \tag{5.6}$$

where:

$$\alpha_i^{(k)} = \frac{p_n(s_i^{(k)})}{p_n'(s_i^{(k)})}\text{ , and } \beta_i^{(k)} = \sum_{i\neq j}^{n}\frac{1}{s_i^{(k)}-s_j^{(k)}}$$

Now equation (5.6) be comes:

$$s_i^{(k+1)} = s_i^{(k)} - \frac{\frac{p_n(s_i^k)}{p_n'(s_i^k)}}{1-\frac{p_n(s_i^k)}{p_n'(s_i^k)}\sum_{i\neq j}^{n}\frac{1}{s_i^{(k)}-s_j^{(k)}}} \tag{5.7}$$

Convergence of Aberth Ehrlich Method

By recalling the AE iterative function F: $C^m \to C^m$ with regard to $P(x)$ is described as follows:

$$F(\bar{a}) = [F_1(\bar{a}), F_2(\bar{a}), ..., F_m(\bar{a})]^t$$

For $i = 1, ..., m$ we have

$$F_i(\bar{a}) = \begin{cases} a_i, & \text{if } p(a_i) = 0 \\ a_i - \frac{1}{\frac{p'(a_i)}{p(a_i)}\sum_{i\neq j}\frac{1}{(a_i-a_j)}}, & \text{otherwise} \end{cases} \tag{5.8}$$

The following lemma determines the error influencing the starting point in terms of the approximation error affecting the method's initial iteration.

Lemma 5.1. *Suppose $m \geq 2$ is the degree of p(x), let k be a real number, such that 0 <k< 1, and $\beta = \frac{k}{\sqrt{m-1}}$, and suppose $\bar{a} \in C^n$ be a complex vector in a way that*

$$|a_i - x_i| \leq \beta |a_i - a_j|, \quad \forall i \neq j \tag{5.9}$$

Then

$$|F_i(\bar{a}) - x_i| \ \leq \ rk^2|a_i - x_i| \quad i = 1,2,3...m,$$

where

$$r = max\left\{\frac{1}{1-\frac{k}{\sqrt{m-1}}-k^2}, \quad \frac{1}{1-(2+k)\frac{k}{\sqrt{m-1}}}\right\}. \tag{5.10}$$

Lemma 5.2. *Under the same assumptions as in the previous lemma, if there were more*

$$r = max\left\{\frac{1}{1-\frac{k}{\sqrt{m-1}}-k^2}, \quad \frac{1}{1-(2+k)\frac{k}{\sqrt{m-1}}}\right\}. \tag{5.11}$$

satisfies $rk \leq \frac{1}{2}$ *(see that* $m > 9$ *and* $k \leq \frac{1}{3}$ *for instance, ensure that this condition is always satisfied), then*

$$|F_i(\bar{a}) - x_i| \ \leq \ r^2k^3 \ |F_i(\bar{a} - F_j(\bar{a})|, \quad \forall i \neq j.$$

Theorem 5.1. *Let* $p(x) = \prod_{i=1}^{m}(x - x_i)$ *be a polynomial of degree m with pairwise distinct roots. Suppose we are given a real number k,* $0 < k \leq \frac{1}{3}$ *and a complex vector* $\bar{a} \in C^n$ *such that*

$$|a_i - x_i| \leq \frac{k}{\sqrt{m-1}}|a_i - a_j|, \quad \forall i \neq j \tag{5.12}$$

$$rk \leq \frac{1}{2} \tag{5.13}$$

where we have set

$$r = max\left\{\frac{1}{1-\frac{k}{\sqrt{m-1}}-k^2}, \quad \frac{1}{1-(2+k)\frac{k}{\sqrt{m-1}}}\right\}. \tag{5.14}$$

Observe that (5.14) is satisfied for $n > 9$ *and* $k \leq \frac{1}{3}$ *(as in the preceding lemma). Let's define the following sequences of vectors recursively:*

$$\begin{cases} \bar{a}^{(0)} = \bar{a}, \\ \bar{a}^{(l+1)} = A(\bar{a}^{(l)}), \quad l = 0,\ 1,\ 2\ldots \end{cases}$$

$F : C^n \to C^n$ *is, as stated in equation (5), the Aberth function with respect to p(x). Then it holds*

$$||\bar{a}_i^{(m)} - \bar{x}||_\infty \ \leq \ rk^{(3^n)}b,$$

.

$$where \quad b = \frac{k}{\sqrt{m-1}} \max_{j \neq i} |a_i^{(0)} - a_j^{(0)}|$$

.

Proof. Lemma 5.2 suggests that equation (5.12) still holds for $\bar{a}_i{}^{(m)}$ with r^2k^3 in place of k, as we have seen before, and Lemma 5.1 offers a reduction ratio for the distance between the iteration vector and the root vector of p. By using an argument based on inductive reasoning, the following inequality holds for each positive integer n:

$$\left|a_i{}^{(n)} - x_i\right| \leq \frac{r^{3^{n-1}} k^{3^n}}{\sqrt{m-1}} \left|a_i - a_j\right|, \quad \forall i \neq j$$

$$\left|a_i{}^{(n+1)} - x_i\right| \leq r^{2.3^{n-1}-1} k^{2.3^{n-1}} \left|a_i{}^{(n)} - x_i\right| \quad i = 1,2,3...m,$$

In fact, by modifying the value after each iteration, we may determine the equalities because the contraction factor k decreases geometrically together with each iteration. In any case, the previous inequalities hold and continue to produce cubic convergence because of r (as stated by equation (5.14)) is a decreasing function of k. By induction from the second inequality again using n, we have

$$\left|a_i{}^{(n)} - x_i\right| \leq \left|a_i{}^{(0)} - x_i\right| \prod_i^n r^{2.3^{j-1}-1} k^{2.3^{j-1}} \leq bk \prod_i^{n-1} r^{2.3^j-1} k^{2.3^j}$$

for every i = 1,..., n. we have

$$bk \prod_i^{n-1} r^{2.3^j-1} k^{2.3^j} < bk \prod_i^{n-1} (r^2k^2)^{3^j} = b(rk)^{3^n}$$

Hence the proof. We can conclude that the previous theorem guarantees cubic convergence for AE method not only asymptotically but also from the first iteration because rk < $\frac{1}{2}$. □

5.2 Long short-term memory (LSTM)

This is a well known architecture of RNN , introduced by Sepp Hochreiter and Juergen Schmidhuber in order to solve vanishing gradient problem. It is a type of RNN can address the vanishing gradient problem faced by traditional RNN. It has feedback connections in

contrasted with standard feed forward neural networks (Computation, 2016).

Every LSTM unit consists of an input gate,a forget gate, an output gate and a cell. The cell retains the values throughout arbitrary time intervals and the three gates govern the passage of information in and out of the cell. By giving a value between 0 and 1 forget gates can decide what information to eliminate from a prior iteration. The value close to 1 means keep the information, and when the value is close to 0 means delete it. Likewise the forget gates, input gates determine which new pieces of information can be stored in the existing state. By providing each piece of information in the present state a value between 0 and 1, output gates also determine which information to output by taking into account both prior and existing states. The LSTM network can maintain valuable, long-term dependencies to make predictions, both in present and future time-steps, by only releasing relevant information from the current state (Gers et al., 2000).

5.2.1 Training LSTM-RNNs

LSTM utilized two algorithms of learning to maintain the error in LSTM memory block cells. These are RTRL and BPTT algorithms. RTRL used for training the network components prior to and including cells, and BPTT for training network elements after cells. Since some partial derivatives have to be computed at each step regardless of whether a target value is given at that step or not, the second units are compatible with RTRL. Consequently, we divide units into two categories: those whose weight changes are computed using a form of BPTT are (output units, hidden units, and output gates) and those whose weight changes are computed using a variation of RTRL are (input gates, forget gates, and cells). We describe the discrete time steps as the form T = 1, 2, 3. Each step contains a forward pass and a backward pass; the forward pass evaluates the output/activation of all units, whereas the backward pass computes the error signals for every single weights.

The Forward Pass

Let B represent the set of memory blocks, let b_c be the c^{th} cell of memory in the memory block b,and $M_{[u,v]}$ be a weight that connect unit u to unit v. In the formulation of LSTM, each memory block b has one input gate in_b and one output gate out_b . The internal state of a memory cell b_c at time $T+1$ is updated in accordance with its state $r_{b_c}(T)$ and according to the weighted input $l_{b_c}(T+1)$ multiplied by the input gate's activation $p_{in_b}(T+1)$. Then, we

utilize the activation of the output gate $l_{out_b}(T+1)$ to determine the cell's degree of activation $p_{b_c}(T+1)$. The activation p_{in_b} of the input gate in_b is computed as:

$$p_{in_b}(T+1) = h_{in_b}(l_{in_b}(T+1)) \tag{5.15}$$

The LSTM block has One or more cells with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is symbolized as r_c. The input gate, p_{in}, and the output gate,p_{out}. The internal cell state is determined by multiplying the output of the squashed input, h(x), by the output of the input gate p_{in} and then add the state of the last time step, $r_c(t-1)$. Finally, the cell output is obtained from the product of the cell state, r_c., and the activation of the output gate, p_{out}.

Here,for the input gate input:

$$\begin{aligned} l_{in_b}(T+1) &= \sum_u M_{[in_b,u]} Q_{[u,in_b]}(T+1),\ with\ u \in Pre(in_b) \\ &= \sum_{v \in U} M_{[in_b,v]} p_v(T) + \sum_{i \in I} M_{[in_b,i]} p_i(T+1) \end{aligned} \tag{5.16}$$

The activation of the output gate out_b is

$$p_{out_m}(T+1) = h_{out_m}(l_{out_m}(T+1)) \tag{5.17}$$

For the input of output gate

$$\begin{aligned} l_{out_b}(T+1) &= \sum_u M_{[out_b,u]} Q_{[u,out_b]}(T+1), with u \in Pre(out_b) \\ &= \sum_{v \in U} M_{[out_b,v]} p_v(T) + \sum_{i \in I} M_{[out_b,i]} p_i(T+1) \end{aligned} \tag{5.18}$$

The results of the gates are adjusted by using the non-linear squashing function, $h_{in_b} = h_{out_b} = h$ defined by

$$h(r) = \frac{1}{1+e^{-}r} \tag{5.19}$$

so that they lie between [0, 1]. Thus, the signal at the input gate must be close to "1" in order to pass the input for the memory cell. The weighted input $l_{b_c}(T+1)$ for a memory cell b_c in

the memory block b is determined by

$$\begin{aligned} l_{b_c}(T+1) &= \sum_u M_{[b_c,u]} Q_{[u,b_c]}(T+1), with \;\; u \in Pre(b_c) \\ &= \sum_{v \in U} M_{[b_c,v]} p_v(T) + \sum_{i \in I} M_{[b_c,i]} p_i(T+1) \end{aligned} \tag{5.20}$$

As we previously discussed, a different computation is used to determine the internal state $r_{b_c}(T+1)$ of the unit in the memory cell at time $(T+1)$; the weighted input is squashed, multiplied by the activation of the input gate, and finally the state of the previous time step $r_{b_c}(T)$ is added. The corresponding equation is

$$r_{b_c}(T+1) = r_{b_c}(T) + p_{in_b}(T+1) f(l_{b_c}(T+1)) \tag{5.21}$$

With the non-linear squashing function for the cell input and $r_{b_c}(0) = 0$,then f(l) becomes:

$$f(l) = \frac{4}{1+e^{-l}} - 2 \tag{5.22}$$

This adjusts the outcome in this instance to the range [-2, 2]. The output p_{b_c} can be obtained by multiplying and squashing the output gate activation p_{out_b} by the cell state r_{b_c}:

$$p_{b_c}(T+1) = p_{out_b}(T+1) g(r_{b_c}(T+1)) \tag{5.23}$$

With the non-linear squashing function with range $[-1,1]$:

$$g(l) = \frac{2}{1+e^{-l}} - 1 \tag{5.24}$$

Assuming a multi layer, recurrent neural network with standard input, output, and a hidden layer made up of memory blocks. The activation of the output unit o is calculated as

$$p_0(T+1) = h_0(l_0(T+1)) \tag{5.25}$$

with

$$l_0(T+1) = \sum_{u \in U-G} M_{[0,u]} p_u(T+1) \tag{5.26}$$

We can once more use the logistic sigmoid from Equation (5.19) to serve as squashing func-

tion for this case G is the set of gate units (Staudemeyer & Morris, 2019).

Forgat gate

In a usual LSTM network, a fixed weight of '1' is allocated to the self-connection in order for maintaining the cell state within time. Unfortunately, the cell states r_b have a tendency to expand linearly as time series supplied as a continuous input flow progress. The primary disadvantage is that the entire memory cell stops its ability to memorize information and starts functioning more like a typical RNN network neuron. However When the data that was previously saved is no longer required, forget gates can learn to reset the internal state of the memory cell (Gers et al., 2000). In order to achieve this, we swap the weight "1.0" to the forget gate activation "p_β" which is calculated in a manner similar to that used for the other gates.

$$p_{\beta_b}(T+1) = f_{\beta_b}(l_{\beta_b}(T+1+d_{\beta_b}) \tag{5.27}$$

where f is the [0, 1]-ranged squashing function from Equation (5.19), d_{β_b} is the forget gate's bias, and

$$\begin{aligned} l_{\beta_b}(T+1) &= \sum_u M_{[\beta_b,u]} Q_{[u,\beta_b]}(T+1), with u \in Pre(\beta_b) \\ &= \sum_{u\in U} M_{[\beta_b,v]} p_v(T) + \sum_{i\in I} M_{[\beta_b,i]} p_i(T+1) \end{aligned} \tag{5.28}$$

Initially d_{β_b} is set to 0, but in order to enhance the performance of LSTM, we fix d_{β_b} to 1 in accordance with (Jozefowicz, Zaremba, & Sutskever, 2015). The revised formula for determining the internal cell state r_{b_c} is

$$r_{b_c}(T+1) = r_{b_c} p_{\beta_b}(T+1) + p_{in_b}(T+1) f(l_{d_c}(T+1)) \tag{5.29}$$

Without forget gate $p_{\beta_b}(T+1) = 1$, utilizing the squashing function in (5.22) with a range of [-2, 2] and $r_{b_c}(0) = 0$. All it takes to get the prolonged forward pass is to swap out Equation (5.21) for Equation (5.29). Input and output gates' bias weights are initialized with negative values, whereas the forget gate's weights are initialized with positive values. This implies that the forget gate activation will be close to '1.0' at the start of training. Without a forget gate, the memory cell will function like a typical LSTM memory cell. By doing this, the LSTM memory cell is kept from forgetting information before it has actually learned it.

Backward Pass

By applying the back propagation the unfolded network can be trained. A training sequence ends with the network being unfolded in time. Using a selected error measure, the error is determined for the output units with the current target values. The weight updates for all time steps are then calculated once the fault has been introduced backwards into the network. The sum of the network's deltas over all time steps is used to update the weights in the recurrent version (Staudemeyer & Morris, 2019).

Using the iterative back propagation algorithm below, we calculate the error signal for a unit for all time steps in a single run. We take into account discrete time steps 1, 2, 3,.. etc, that are indexed by the variable T. The network begins at time t' and continues until time t, when it ends. The interval between t' and t is referred to as an epoch. Let h_u is the differentiable, non-linear squashing function of the unit $u \in U$; Let U be the set of non-input units. Then the output $p_u(T)$ of u at time T is given by.

$$p_u(T) = h_u(l_u(T)) \tag{5.30}$$

with the weighted input

$$l_u(T+1) = \sum_q M_{[u,q]} Q_{[q,u]}(T+1), with q \in Pre(u) \tag{5.31}$$

It implies that

$$l_u(T+1) = \sum_v M_{[u,v]} p_v(T) + \sum_i M_{[u,i]} p_i(T+1) \tag{5.32}$$

where $v \in U \bigcap Pre(u)$ and $i \in I$ are the set of input units. Keep in mind that there are two different sorts of inputs to u at time T+1: the recurrent output from all of the network's non-input units that was created at time T, and the environmental input that arrives at time T+1 via the input units. If all nodes are linked to the network, then $U \bigcap Pre(u)$ equals the set U of non-input units. Let L(T) be the collection of non-input units for which, at time T, the output value p_u (T) of the unit $u \in L(T)$ ought to coincide with some target value $a_u(T)$. Using a learning algorithm, we aim to minimize the total error Etotal (t', t) for the epochs t', t' + 1,..., t is the cost function. Such an error is defined as follows:

$$E_{total}(t', t) = \sum_{T=t'}^{t} E(T) \tag{5.33}$$

using the squared error as an objective function to define the error E(T) at time T.

$$E(T) = \frac{1}{2} \sum_{u \in U} e(u(T))^2 \tag{5.34}$$

and with the non-input unit u, error $e_U(T)$ at the time T given by

$$e_U(T) = \begin{cases} a_U(T) - p_U(T), & \text{if } u \text{ is in L} \\ 0, & \text{otherwise} \end{cases} \tag{5.35}$$

Following the notation used in previous sections, and using Equations (5.33) and (5.35), the overall network error at time step T is

$$E(T) = \frac{1}{2} \sum_{o \in O} (a_0(T) - p_0(T))^2 \tag{5.36}$$

Where $a_0(T) - p_0(T) = e_0(T)$, Let's start by thinking about BPTT compatible units. The notion of individual error of a unit u at time T is defined by

$$u(T) = -\frac{\partial E(T)}{\partial l_u(T)} \tag{5.37}$$

where l_u is the weighted input of the unit. The notion for contributions of weight can be expanded in the following way:

$$\begin{aligned} \triangle M[u,v](T) &= -\eta \frac{\partial E(T)}{\partial M[u,v](T)} \\ &= -\eta \frac{\partial E(T) \partial l_u(T)}{\partial l_u(T) \partial M[u,v](T)} \end{aligned} \tag{5.38}$$

The factor $\frac{\partial z_u(T)}{\partial W[u,v](T)}$ corresponds to the signal of input from unit v to unit u. However, depending on the nature of u, the individual error varies. When an output unit o equals to u, then

$$\vartheta_o(T) = h_0'(l_0(T))(a_0(T) - p_0(T));$$

Thus, the output units' weight contribution is

$$\triangle M[o,v](T) = \eta \, \vartheta_o(T) Q[v,o](T)$$

At this point, if u is equivalent to a hidden unit hi located between the cells and the outputs

then

$$\vartheta_{hi}(T) = h'_{hi}(l_{hi}(T)) \left(\sum_{o \in O} M_{[o,hi]} \vartheta_0(T) \right)$$

Here O represents the set of output units, and the weight contribution of the hidden units is

$$\triangle M[hi,v](T) = \eta \, \vartheta_{hi}(T) Q[v,hi](T)$$

Lastly, if u equals the memory block $b's$ of the output gate out_b, then

$$\vartheta_{out_b}(T) \stackrel{tr}{=} h'_{out_b}(l_{out_b}(T)) \left(\sum_{b_c \in b} hi(r_{b_c}(T) W_{[o,b_c]} \vartheta_0(T)) \right)$$

where $\stackrel{tr}{=}$ refers to the equality is valid only if the error is truncated to avoid "too much" propagation that is, it prevents the error from returning to the unit through its own feedback connection. Lastly, output gates' weight contribution is

$$\triangle M[out_b,v](T) = \eta \, \vartheta_{out_b}(T) Q[v,out_b](T)$$

Let us consider units that are working with RTRL. In this instance,Each of the errors of the input gate and the forget gate centered on the errors of the memory block's individual cells. The cell b_c's individual error of the memory block b is defined as

$$\begin{aligned}
\vartheta_{b_c}(T) &\stackrel{tr}{=} -\frac{\partial E(T)}{\partial r_{bc}(T)} + \vartheta_{b_c}(T+1) p_{\beta_b}(T+1) \\
&\stackrel{tr}{=} \frac{\partial p_{b_c}(T)}{\partial r_{bc}(T)} \left(\sum_{o \in O} \frac{\partial l_0(T)}{\partial p_{bc}(T)} - \frac{\partial E(T)}{\partial l_0(T)} \right) + \vartheta_{b_c}(T+1) y_{\beta_b}(T+1) \\
&\stackrel{tr}{=} p_{out_b}(T) hi'(r_{bc}(T)) \left(\sum_{o \in O} M[o,b_c](T) \vartheta_0(T) + \vartheta_{b_c}(T+1) p_{\beta_b}(T+1) \right)
\end{aligned}$$

In this equation the error that occurs can only be propagated backward in time via the recurrent link between the cell and other units (that accounts for the impact of the forget gate). The recurrent link between the cell and other units is not taken into account.

To raise the weight contribution for the cell, we use the following partial derivatives.

$$\begin{aligned}\triangle M[b_c,v](T) &= -\eta\frac{\partial E(T)}{\partial M[b_c,v]}\\ &= -\eta\frac{\partial E(T)\partial r_{b_c}(T)}{\partial r_{b_c}(T)\partial M[b_c,v]}\\ &= \eta\,\vartheta_{m_c}(T)\frac{\partial r_{b_c}(T)}{\partial M[b_c,v]}\end{aligned} \tag{5.39}$$

additionally the forget and input gate weight contributions are the following:

$$\begin{aligned}\triangle M[u,v](T) &= -\eta\frac{\partial E(T)}{\partial M[u,v]}\\ &= -\eta\sum_{b_c\in b}\frac{\partial E(T)\partial r_{b_c}(T)}{\partial r_{b_c}(T)\partial M[u,v]}\\ &= \eta\sum_{b_c\in m}\vartheta_{b_c}(T)\frac{\partial r_{b_c}(T)}{\partial M[u,v]}\end{aligned} \tag{5.40}$$

We now need to determine the value of $\frac{\partial r_{b_c}(T+1)}{\partial M[u,v]}$ As predicted, these also depend on the features of the unit u. If u is equivalent to the value of cell b_c , after that

$$\frac{\partial r_{b_c}(T+1)}{\partial M[b_c,v]} \overset{tr}{=} \frac{\partial r_{b_c}(T)}{\partial M[b_c,v]}p_{\beta_b}(T+1) + f'(l_{b_c}(T+1))h_{in_b}(l_{in_b}(T+1))p_v(T) \tag{5.41}$$

At this instance, if u is equivalent to the input gate in_b, then

$$\frac{\partial r_{b_c}(T+1)}{\partial M[in_b,v]} \overset{tr}{=} \frac{\partial r_{b_c}(T)}{\partial M[in_b,v]}p_{\beta_b}(T+1) + f(l_{b_c}(T+1))h'_{in_b}(l_{in_b}(T+1))p_v(T) \tag{5.42}$$

Lastly, if u is equivalent to a forget gate β_b , then

$$\frac{\partial r_{b_c}(T+1)}{\partial M[\beta_b,v]} \overset{tr}{=} \frac{\partial r_{b_c}(T)}{\partial M[\beta_b,v]}y_{\beta_b}(T+1) + r_{b_c}(T+1)h'_{\beta_b}(l_{\beta_b}(T+1))p_v(T) \tag{5.43}$$

with $r_{b_c}(0) = 0$ (Gers et al., 2000).

Solving the Vanishing Error Problem

Higher time steps cannot be bridged by a standard RNN (Gers et al., 2000). This is because back-propagated error signals have a tendency to increase or decrease with each time step. As a result, the error often grows or vanishes over a large number of time steps (Bengio et al.,

1994). While vanished error signals result in either no learning at all or learning that takes an unacceptably long time, blown-up error signals immediately cause oscillating weights.

The following describes how the common back propagation algorithm and the fundamental vanishing error analysis compute gradients: After the network has trained, we use the formula to update weights from time t' to time t.

$$\triangle M[u,v] = -\eta \frac{\partial E_{total}(t',t)}{\partial M[u,v]}$$

with

$$\frac{\partial E_{total}(t',t)}{\partial M[u,v]} = \sum_{T=t'}^{t} \vartheta_u(T) Q[u,v](T),$$

where the back propagated error signal at time t (with t' $\leq$ T $<$ t) of the unit u is

$$\vartheta_u(T) = h'_u(l_u(T)) \left(\sum_{u \in U} M_{vu} \vartheta_u(T+1) \right) \tag{5.44}$$

Consequently, given a fully recurrent neural network with a set of non-input units U , the error signal that occurs at any chosen output-layer neuron $o \in O$,at time-step T , is propagated back through time for t - t' time-steps, with t' $<$ t to an arbitrary neuron v. This causes the error to be scaled by the following factor: The error signal which exists at every selected output layer neuron $o \in O$ at time-step T, is consequently propagated back via time for t - t' time-steps, with t' $<$ t to any neuron v provided a completely recurrent neural network having a set of non input units U. As a result, the error is transformed by the following factor:

$$\frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)} = \begin{cases} h'_v(l_v(t'))M[o,v], & \text{if t-t'=1} \\ h'_v(l_v(t'))(\sum_{u \in U} \frac{\partial \vartheta_u(t'+1)}{\partial \vartheta_o(t)} M[u,v]), & \text{if t-t'$>$ 1} \end{cases} \tag{5.45}$$

We gradually unroll the equation in order to find the answer. For $t' < T < t$, consider u_T to be a non input layer neuron in one of the unrolled network's copies at time T. Now, by putting $u_t = v$ and $u_{t'} = o$,we get the equation

$$\frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)} = \sum u_{t'} \in U \ldots \sum_{u_{t-1} \in U} \left(\prod_{T=t'+1}^{t} h'_{u_T}(l_{u_T}(t-T+t'))M[u_T, u_{T-1}] \right) \tag{5.46}$$

from equation (5.46) it follows that if

$$|h'_{u_T}(l_{u_T}(t-T+t'))M[u_T,u_{T-1}]| > 1 \tag{5.47}$$

In addition, clashing error signals that occur at neuron v might result in oscillating weights and unstable learning. Then for all T the product grows exponentially and blowing up the error. If now

$$|h'_{u_T}(l_{u_T}(t-T+t'))M[u_T,u_{T-1}]| < 1 \tag{5.48}$$

The error vanishes and the product falls exponentially for all T, which prohibits the network from learning in a reasonable time period. Lastly, the equation will be

$$\sum_{o\in O} \frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)} \tag{5.49}$$

It demonstrates that if the local error vanishes, the global error does too (Bengio et al., 1994).

5.3 Datasets

In this section, we described the steps taken to generate a training dataset, a test dataset, and train ANNs to approximate polynomial real zeros . We generated a synthetic dataset of polynomial coefficients and roots to train and test our polynomial regression models. We then randomly generated n samples set of n degree roots uniformly within this range and sorted them in ascending order. The dataset consists of $100,000$ samples per polynomial, each with a polynomial of degree $5, 10, 15, 20$ *and* 25. Besides that, there are no missing values. The following tables represent the sample of input datasets (the coefficients) and out put datasets (the roots) of polynomial of degree five.

Table 5.1: The samples of input datasets of degree 5 to train ANN

c_0	c_1	c_2	c_3	c_4	c_5
1	1.265042	-0.482262	-0.43442	0.042327	0.02548
1	2.414208	2.000097	0.61563634	0.01323377	-0.01675147
1	-3.46314	4.762822	-3.25162058	1.10197941	-0.148305
1	-1.04586	-0.011242	-0.0112420	0.8773	-0.05246885
1	-2.447667	-0.7622626	-2.5088	0.06979785	0.00825952
1	-2.3954	1.782560	-0.305351	-0.089848736	0.0087514
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Table 5.2: The samples of output datasets of degree 5 to train ANN

r_1	r_2	r_3	r_4	r_5
-0.974996	-0.75212	-0.22313075	0.321141	0.4849622
-0.998081	-0.58208494	-0.563887529	-0.39847796	0.12832285
0.531107	0.6202462	0.678651795	0.75821971	0.8749195
0.67303	0.837787616	0.987892845	-0.0173	0.54087
-0.064333	0.342127	0.663707	0.71143313	0.794731
-0.958551	-0.8917109	-0.445623	0.10887134	0.800691
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

The dataset was split randomly into training and testing sets using 4 : 1 ratio. The polynomial coefficients were generated uniformly at random in the range $[-1,1]$, and the corresponding roots were sorted in ascending order. For the case when polynomials have only real roots, roots were generated in the closed interval of -1 to 1. For the case when polynomial have both real and complex roots, coefficients were generated in the closed interval of 0 to 1. However,in this work we did not include complex roots. After generating the random roots only the real zeros (r_i ,where i=1,2,...n) of polynomials of degree n ($p_n(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1 x + a_0 = 0$) by using the real coefficients (c_i for i=1,2,..n) can be approximated using ANNs as follows.

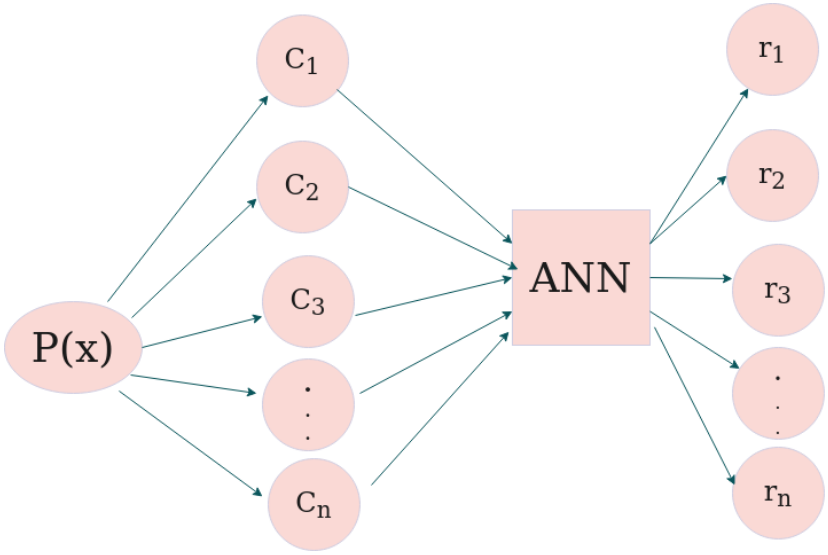


Figure 5.1: The block diagram of ANN approach for real roots of polynomials

The 80% of our data were used for training the model and 20% of our data used for testing the neural network and AE algorithm. The idea to build this dataset is to use it for testing and comparing tools that compute the roots of polynomials. We have used to test artificial intelligence tools (ANN particularly LSTM algorithm) and the dataset was used to compare our solution to the other methods . In this thesis we made comparison between the result of Recurrent Neural Networks(LSTM algorithm) and AE method.

Parameters used

In this work we used hyper parameters to regulate the training process and to optimize the model. These are : batch size= 2^7 ,epoch=100 and learning rate= $10^{-}4$. Since Adam is relevant to LSTM neural networks and the objective is to determine a set of parameters that reduce mean squared error, We used Adam optimization algorithm with the following steps.

1. Initialization

   $t \leftarrow 0$;

   for i$\in$N do

   $m_{i,0} \leftarrow m_{i,0}/||m_{i,0}||_2$;

   $l_o m_i \leftarrow 0$;

   $u_o m_i \leftarrow 0$;

2. Conduct T training iterations

   while t < T do

   $t \leftarrow$ t+1

3. Update θ^u

   for i$\in$ N do

$$\bar{h_t}(m_i) \leftarrow \frac{\partial R}{\partial m_i};$$
$$h_t(m_i) \leftarrow \bar{h_i}(m_i) - \left(\bar{h_i}(m_i).m_{i,t-1}\right) m_{i,t-1} ;$$
$$l_t(m_i) \leftarrow \beta_1 l_{i,t-1}(m_i) + (1-\beta_1) h_t(m_i);$$
$$u_t(m_i) \leftarrow \beta_2 u_{i,t-1}(m_i) + (1-\beta_2)||h_t(m_i)||_2^2;$$
$$\hat{l}_t(m_i) \leftarrow l_t(m_i)/(1-\beta_1^t);$$
$$\hat{u}_t(m_i) \leftarrow u_t(m_i)/(1-\beta_2^t);$$
$$m_{i,\bar{t}-1} \leftarrow m_{i,t-1} - \alpha_t^u \hat{l}_t(m_i)/\left(\sqrt{\hat{u}_t(m_i)} + \varepsilon\right);$$
$$m_{i,t} \leftarrow \bar{m_{i,t}}/||\bar{m_{i,t}}||_2;$$

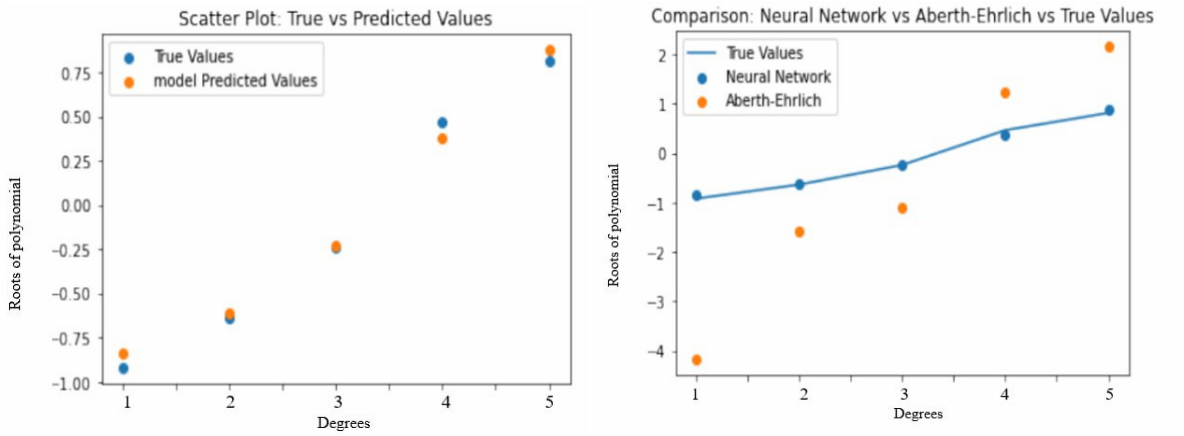
   Update θ^s in similar manner as θ^u

4. return θ_T which is resulting parameter

where θ is the training data which divided into θ^u and θ^s trainable parameters, hyper Parameters β_1 and β_1 regulate how quickly the moving averages decay. The two sets of parameters' learning rates are represented by the symbols α_t^u and α_t^s respectively, h is the gradient at step t,and also l and u are the exponential moving averages.

5.4 Results

The following figures show the comparisons, in terms of accuracy, between the approximations to the real zeros of five random real polynomials of degrees 5, 10, 15, 20, and 25 produced by the ANN approach, Aberth Ehrlich and true values.

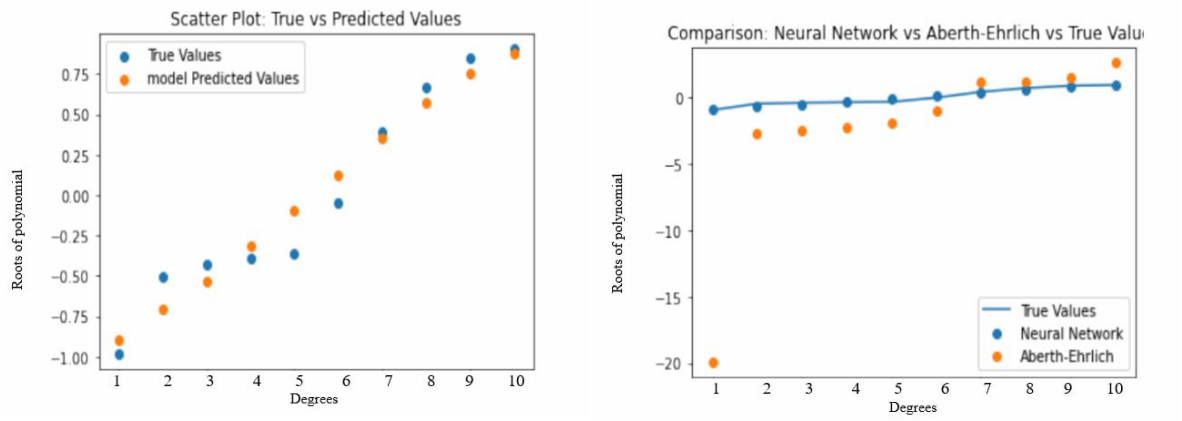


(a) ANN and True value

(b) ANN, AE and True value

Figure 5.2: Comparison between ANN, AE methods and true values for degree 5.

In this figure (a) we have seen that the relation ship between the values we got from ANN method and the predicted values for polynomials of degree five. Initially their is a big difference between ANN method and the true values, but as degree increase the gap becomes minimum so that as the degree increase the gap decreases and they becomes closer and closer. And for figure (b) we plotted the two methods and the true values on the same graph. As we saw here the values we got from AE method is extremely difference from both ANN method and the true values at the begging, after some distance it becomes closer. There is a huge difference between the AE method and both ANN method values and the true values for first degrees. Which means that up to third degree the value we got by AE method is too much small than both the two values. For the next degrees the values of AE method are greater than both the model predicted values and ANN values. However,the gab is decrease relative to the very first degrees.

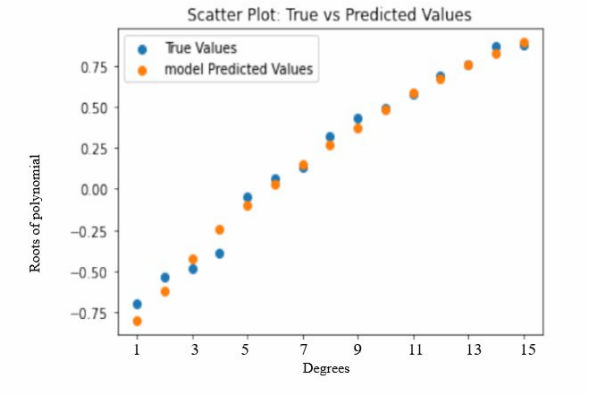


(a) ANN and True value

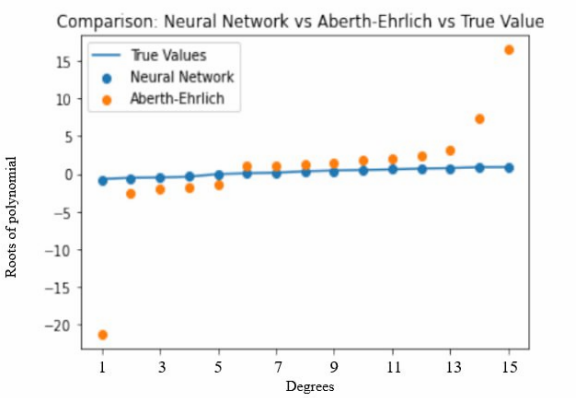
(b) ANN, AE and True value

Figure 5.3: Comparison between ANN, AE methods and true values for degree 10.

This figure (a) shows that the relation ship between the values which obtained from ANN method and the true values up to tenth degree. As we saw on the figure a big variation between the values is visible at the beginning. we get this variation starting from degree one up to degree five . At the first degree the model predicted value is larger than the true value, but for second and third degrees the predicted value is smaller than the true values. For degree four values are closer when it compared with degree one. For degree five the variation happened again. However after degree five the two values become closer and closer. Here in figure (b) we have constructed the values we got by using AE method, the values we got from ANN method and the true values on the same coordinate. As we see on the figure initially the values we got from the AE method is extremely different from both the values of ANN method and the true values, which is very small. It is happened particular for the very first degrees ,but this difference is decrease when degrees increase. when we see the graph for the six, seven, and eight degrees the values are very close to each other. How ever it is not continue, that is for the tenth degree the value of AE method is slightly greater than from the two values.



(a) ANN and True value

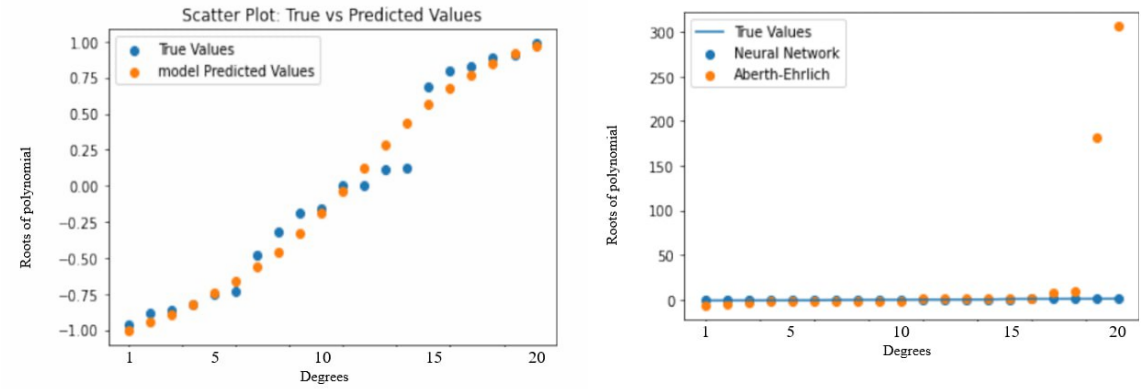


(b) ANN, AE and True value

Figure 5.4: Comparison between ANN, AE methods and true values for degree 15.

This figure (a) describes the relation ships between the values from the artificial neural network method and the true values for the polynomials of degree fifteen. As we saw the graph the values we got by ANN method is slightly differ from the true value. For the first two degrees the model predicted value(the ANN value) is less than the true values. For the next two degrees the model predicted value is greater than the true value. However, starting from degree five the two values are very close each other, they are also attached each other.

In addition on figure (b) we can see that how the three values can relate each other. Like the previous figures the AE method values, the ANN values, and the true values plotted on the same graph. When we see the figure at the begging the values we got by using AE method is very far from the two values ,that is very small. Even if after some points the values of AE method become closer, they are still less than the two values. After degree five the Aberth method values have good agreement between both the ANN method values and the true values but, the AE method values slightly greater than from both values. This agreement continues up to degree ten but, after degree ten the values we get by using AE are rapidly increase.

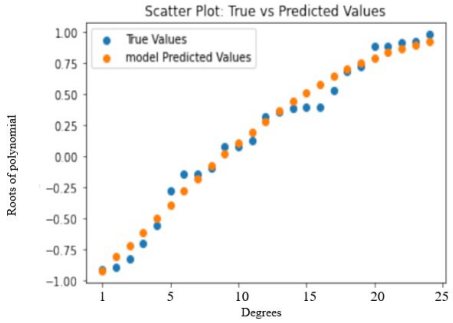


(a) ANN and True value

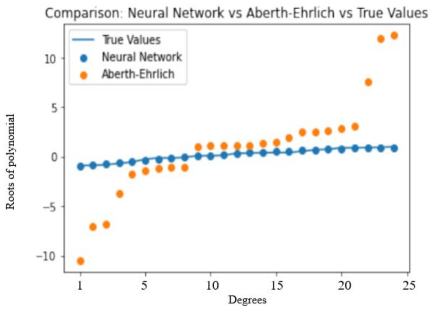
(b) ANN, AE and True value

Figure 5.5: Comparison between ANN, AE and true values for degree 20.

This graph (a) tell us the relation ship between the values we got from the ANN method and the values we got from training datasets which means the true values of degree twenty. The values have a good agreement for the beginning degrees that is, up to degree four. specially for degree three and five they are overlap each other,it indicates that the two values are equal. After degree six there is a little variation between the values of the two methods which means the value of ANN is greater than the true value. Around degree ten they have a good relation but, after that there is a small variation. At the end part (for larger degree) they becomes again closer. Here figure (b) shows the relation ship between the values we got from training datasets, the values we got from ANN method and the values we got from AE method for real polynomials of degree twenty. In this graph we saw the same thing up to polynomials of degree fifteen. It indicates there is a very good agreement between the values we got from the two methods and also the true values. However for the degrees greater than seventeen the value we got from AE method is rapidly increase and becomes very large. There for there is a big difference between AE values and the other two values for degrees greater than seventeen.



(a) ANN and True value



(b) ANN, AE and True value

Figure 5.6: Comparison between ANN, AE methods and true values for degree 25.

On this figure (a) we see the comparison between ANN approximation values and true values to the real zeros of a random real polynomial of degree twenty five. Here the ANN values have a good agreement with the true values. At the beginning the model predicted value is greater than the true value but, it continues until fifth degree. For degree five and six the values are far apart and the ANN values are smaller than the true values. From degree seven up to fifteen there is a good agreement between the two values. For some degrees after degree fifteen the ANN value again show a small variation but, it is only for degree 16,17 and 18. For the last degrees the values become closer and they looks like similar.

Similarly the previous figures figure (b) shows the Comparison between ANN method values, AE approximations and true values to the real zeros of a random real polynomial of degree twenty five. As shown in the figure, at the beginning the AE values and the other two values are very much different. From degree four up to degree eight despite they are closer, the AE values are still less than the true values. For degrees between nine and fifteen the values are closer each other but, here the AE method values are slightly greater than from the two values. This continues with small variation until twenty degree. However after twenty degree the value of AE is rapidly increase and the relation ship is absolutely disagree.

Evaluation Metrics

Evaluation metrics are used to measure the quality of the statistical or machine learning model. Evaluating machine learning models or algorithms is essential for any work related to it. There are many different types of evaluation metrics available to test a model. A model evaluation metric is a mechanism for assessing the performance of a machine learning model. Mean absolute error(MAE), mean squared error(MSE),root mean squared error(RMSE) and root mean squared log error(RMSLE) are some examples of widely used metrics for Regression problems (Zhou et al., 2021).

Our experiments evaluated the performance of a neural network model and the AE algorithm in solving a set of polynomials. The models were evaluated using mean squared error (MSE) and execution time. The MSE measures the difference between the predicted values and true values of the solutions, while execution time indicates the efficiency of the algorithms to compute the roots of the polynomial function.

Mean Squared Error(MSE)

MSE is a most used and very simple metric with a little bit of change in mean absolute error. Mean squared error states that finding the squared difference between actual and predicted value. It represents the squared distance between actual and predicted values. we perform squared to avoid the cancellation of negative terms and it is the benefit of MSE. The advantages of MSE is the graph of MSE is differentiable, so we can easily use it as a loss function (Chicco et al., 2021).

The Mean Square Error (MSE) for each of the five different degrees of polynomials, computed as follows, where T_i denotes the true values we get from the training datasets $i-^{th}$ zero $(i = 1, ..., n)$ and M_i the corresponding approximation obtained by using one of the two methods that is, either ANN method and AE is as follows :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(T_i - M_i)^2 \tag{5.50}$$

Our findings, despite being constrained, are highly encouraging and provide strong evidence for the effectiveness and potential of our artificial neural network (ANN) methodology in approximating real polynomial roots. The execution time and MSE results for both approaches, displayed on table 5.3 and table 5.4, were generated using a personal computer

featuring a 5th generation Intel Core i5 processor and 8 GB of RAM, they show the efficiency and accuracy of our approach.

Table 5.3: Results for AE method

Degree	Execution time(in seconds)	MSE results
5	0.935653906	1.447162665
10	0.714002817	65.747391349
15	1.010222591	65.698129439
20	2.096060629	90.377939718
25	3.664746237	696.382299944

Table 5.4: Results for ANN method

Degree	Execution time(in seconds)	MSE results
5	0.00019742	0.007829159285412592
10	0.000206834	0.0008035661550521418
15	0.000655497	0.0011202595846592914
20	0.000365223	0.0011014252429451003
25	0.000354652	0.0011277572109779613

5.5 Discussion

Figures 5.2 to 5.6 compare the accuracy of the AE method and the ANN approach's approximations to the real zeros of five randomly chosen real polynomials with degrees of 5, 10, 15, 20, and 25. When analyzing the plots, it is evident that ANN method yields outcomes that are quite comparable to those attained using the AE method. However, it is feasible to see a minor increase in the disparities between the approximations provided by the two approaches as the degree of the polynomial minimum or maximum.

Table 5.4 demonstrates that the execution time with ANN method almost remains constant as the degree of the polynomial grows. The AE method on table 5.3, on the other hand, results in longer execution times as the degree of the polynomial increases. When comparing the execution timings of the two approaches, we can see that the AE method takes a lot longer to compute the approximations to the zeros using ANN. This is because, unlike ANN, computing polynomial zeros using the AE method is a strictly iterative process.

Additionally, we evaluated the two methods potential for extrapolating their results to different results. For this, new datasets were used together with training network unused sample sets. Tables 5.3 and 5.4 compare the calculated outputs with the target values (MSE). Because every value of the MSE for ANN method in Table 5.4 is significantly lower than MSE value of AE method in Table 5.3 , we may conclude that the ANN method have a good ability to generalize the results. Thus, we can say that the ANN method can solve any real univariate polynomial of the corresponding degree using only real zeros.

Conclusion and Recommendation

Conclusion

In this study we presented the definition, types, components, architectures, learning methods, and learning rules of artificial neural networks, and we studied the derivation and convergence of AE method. We used ANN approach for approximating the real roots of a given polynomial, based on the inductive inference capabilities of the ANNs. We generated a synthetic dataset of polynomial coefficients and roots to train and test our polynomial models. The dataset was split randomly into training and testing sets using $4:1$ ratio respectively. We started by training the ANNs using the LSTM algorithm and as input the coefficients of the polynomials considered. LSTM provides a possible solution to this problem by introducing a constant error flow through the internal states of special memory cells. We used feedback neural network and supervised version to update the network parameters, two activation functions (Sigmoid activation function and hyperbolic tangent activation function) were used in this work. The weights of each ANN were then updated based on the error between the true root values and the values produced by the ANN. Results were then compared in terms of accuracy and efficacy to the AE method. We showed the accuracy and efficiency of ANN method by using evaluation metrics (MSE values and execution times) with compared to AE method. we also plotted the graphs by using python code in order to investigate the accuracy levels of the two methods.

Recommendation

Here, we discussed how LSTM algorithm works to determine real roots of polynomial and took selective polynomials to show the efficiency of LSTM algorithm. However, in this study we considered only AE method. Future studies will evaluate the suggested method against additional iterative techniques for polynomial root finding, we will also consider several ANN models. In addition, it is essential to contrast various methods for estimating the initial approximations for the roots of a particular polynomial based on its coefficients and to take certain classes of polynomials into account in addition to randomly generated polynomials.

Research Fund Acknowledgment

This research project is funded by Adama Science and Technology University under the grant number $ASTU/SM-R/764/23$, Adama, Ethiopia.

References

Andoni, A., Panigrahy, R., Valiant, G., & Zhang, L. (2014). Learning polynomials with neural networks. In *International conference on machine learning* (pp. 1908–1916).

Barbeau, E. J. (2003). *Polynomials*. Springer Science & Business Media.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166.

Chakraverty, S., & Mall, S. (2017). *Artificial neural networks for engineers and scientists: solving ordinary differential equations*. CRC Press.

Chartrand, G., Cheng, P. M., Vorontsov, E., Drozdzal, M., Turcotte, S., Pal, C. J., ... Tang, A. (2017). Deep learning: a primer for radiologists. *Radiographics*, *37*(7), 2113–2131.

Chicco, D., Warrens, M. J., & Jurman, G. (2021). The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation. *PeerJ Computer Science*, *7*, e623.

Computation, N. (2016). Long short-term memory. *Neural Comput*, *9*, 1735–1780.

Das, M., & Seal, P. (2012). Polynomial real roots finding using feed forward neural network: A simple approach. In *2012 national conference on computing and communication systems* (pp. 1–4).

Dongare, A., Kharde, R., Kachare, A. D., et al. (2012). Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, *2*(1), 189–194.

Fatheddin, H., & Sajadian, S. (2022). Improved aberth–ehrlich root-finding algorithm and its further application for binary microlensing. *Monthly Notices of the Royal Astronomical Society*, *514*(3), 4379–4384.

Freitas, D., Guerreiro Lopes, L., & Morgado-Dias, F. (2021). A neural network-based approach for approximating arbitrary roots of polynomials. *Mathematics*, *9*(4), 317.

Freitas, D., Lopes, L. G., & Morgado-Dias, F. (2018). A neural network based approach for approximating real roots of polynomials. In *Proceedings of the international conference on mathematical applications (icma), funchal, portugal* (pp. 9–12).

Gers, F. A., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: Continual predic-

tion with lstm. *Neural computation*, *12*(10), 2451–2471.

Hormis, R., Antoniou, G., & Mentzelopoulou, S. (1995). Separation of two-dimensional polynomials via a sigma-pi neural net. In *Proc. int. conf. on modelling and simulation* (pp. 304–306).

Huang, D., & Chi, Z. (2004). Finding roots of arbitrary high order polynomials based on neural network recursive partitioning method. *Science in China Series F: Information Sciences*, *47*, 232–245.

Huang, D.-S., Ip, H. H., & Chi, Z. (2004). A neural root finder of polynomials based on root moments. *Neural Computation*, *16*(8), 1721–1762.

Huang, D.-S., Ip, H. H., Chi, Z., & Wong, H. (2003). Dilation method for finding close roots of polynomials based on constrained learning neural networks. *Physics Letters A*, *309*(5-6), 443–451.

Jain, A., Zamir, A. R., Savarese, S., & Saxena, A. (2016). Structural-rnn: Deep learning on spatio-temporal graphs. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 5308–5317).

Jain, A. K., Mao, J., & Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. *Computer*, *29*(3), 31–44.

Jeswal, S. K., & Chakraverty, S. (2018). Solving transcendental equation using artificial neural network. *Applied Soft Computing*, *73*, 562–571.

Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *International conference on machine learning* (pp. 2342–2350).

Lukoševičius, M., & Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer science review*, *3*(3), 127–149.

McClelland, J. L., Rumelhart, D. E., & Hinton, G. E. (1986). The appeal of parallel distributed processing. *MIT Press, Cambridge MA*, *3*, 44.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*, 115–133.

McNamee, J. M., & Pan, V. (2013). *Numerical methods for roots of polynomials-part ii*. Newnes.

Mourrain, B., Pavlidis, N. G., Tasoulis, D. K., & Vrahatis, M. N. (2006). Determining the number of real roots of polynomials through neural networks. *Computers & Mathe-*

matics with Applications, *51*(3-4), 527–536.

Pal, S., Ghosh, S., & Nag, A. (2018). Sentiment analysis in the light of lstm recurrent neural networks. *International Journal of Synthetic Emotions (IJSE)*, *9*(1), 33–39.

Paola, J. D., & Schowengerdt, R. A. (1995). A review and analysis of backpropagation neural networks for classification of remotely-sensed multi-spectral imagery. *International Journal of remote sensing*, *16*(16), 3033–3058.

Petković, M., & Rančić, L. (2006). A family of root-finding methods with accelerated convergence. *Computers & Mathematics with Applications*, *51*(6-7), 999–1010.

Pramoditha, R. (2021). *The concept of artificial neurons (perceptrons) in neural networks.* Medium.

Sejnowski, T., & Delbruck, T. (2012). The language of the brain: The brain makes sense if our experiences by focusing closely on the timing of the impulses that flow through billions of nerve cells. *Scientific American*, *307*(4), 54.

Shanmuganathan, S. (2016). *Artificial neural network modelling: An introduction*. Springer.

Sharma, S., Sharma, S., & Athaiya, A. (2017). Activation functions in neural networks. *Towards Data Sci*, *6*(12), 310–316.

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

Team, G. L. (2020). Types of neural networks and definition of neural network. *URL https://www. mygreatlearning. com/blog/types-of-neural-networks/https://www. mygreatlearning. com/blog/types-of-neural-networks/{#} typesofneuralnetworks*.

Zhang, X., Zhu, D., & Hu, W. (2008). Finding multiple real roots by neural networks based on complete discrimination system of polynomial. In *2008 ieee conference on cybernetics and intelligent systems* (pp. 236–241).

Zhou, J., Gandomi, A. H., Chen, F., & Holzinger, A. (2021). Evaluating the quality of machine learning explanations: A survey on methods and metrics. *Electronics*, *10*(5), 593.

Zou, J., Han, Y., & So, S.-S. (2009). Overview of artificial neural networks. *Artificial neural networks: methods and applications*, 14–22.