NEURAL NETWORK APPROACHES FOR ACCURATE AFAN OROMO SPELL CHECKING AND CORRECTION

MEKELE UNIVERSITY

MEKELE INSTITUTE OF TECHNOLOGY

FACULTY OF COMPUTING AND INFORMATICS

DEPARTMENT OF INFORMATION TECHNOLOGY

POSTGRADUATE PROGRAM

A Thesis Proposal Submitted to the Department of Information Technology in Partial Fulfilment of the Requirements for the degree of Masters of Science in Information Technology

By:
Ayenalem Dejene

Advisor: -Behailu Getachew (PhD)

Mekele, Ethiopia

August 2025

Declaration

I, Aynalem Dejene, declare that this thesis entitled "NEURAL NETWORK APPROACHES FOR ACCURATE AFAN OROMO SPELL CHECKING AND CORRECTION" is my original work. I have undertaken the thesis work independently with the guidance and support of my thesis advisor. This study has not been submitted for any degree or diploma program in this or any other institution, and all sources of materials used for the thesis have been duly acknowledged.

Declared by

Name: ……....

Signature: ……..

Department: ……..

Date: …….

Certification

This is to certify that the thesis prepared by Aynalem Dejene, entitled "NEURAL NETWORK APPROACHES FOR ACCURATE AFAN OROMO SPELL CHECKING AND CORRECTION," and submitted in partial fulfilment of the requirements for the Degree of Masters of Science in Information Technology complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Name of Candidate: Aynalem Dejene; Signature: ……. Date:…….

Principal Advisor: Behailu Getachew (Ph.D) ; Signature: ……..... Date:……...

Signature of Board of Examiner`s:

External examiner: …….……..... Signature: ……. Date: ……...

Internal examiner.…...……..... Signature: ……. Date: ……...

Dean, SGS: ……..... Signature: ……. Date: ……...

Acknowledgements

I am deeply grateful to Almighty God for His guidance and strength throughout this journey. My heartfelt thanks go to my thesis advisor, Dr. Behailu Getachew, for his invaluable guidance, expertise, and encouragement. I also extend my gratitude to my teachers for their knowledge and support, which were instrumental in completing this work. Finally, I appreciate my friends and family for their unwavering encouragement and companionship.

Abstracts

Afan Oromo, a widely spoken Cushitic language, lacks advanced natural language processing (NLP) tools like spell checkers due to limited resources and linguistic expertise. Both native and non-native speakers face challenges in writing Afan Oromo correctly, partly because its Latin-based Qubee script was adopted in 1991. Traditional spell-checking methods, such as dictionary lookup and rule-based approaches, are inadequate for Afan Oromo's highly inflectional morphology. This thesis proposes a neural network-based spell checker using a sequence-to-sequence (Seq2Seq) model with Long Short-Term Memory (LSTM) layers. A corpus of 596,948 words was collected from BBC Afan Oromoo using Sketch Engine, ensuring compliance with BBC's terms of service. The model was trained to detect and correct spelling errors, achieving 100% error recall and 52.47% precision. This work is the first to apply neural networks to Afan Oromo spell checking, offering a scalable solution for under-resourced languages.

Keywords: Afan Oromo, Spell Checker, Neural Network, Sequence-to-Sequence, LSTM

Table of Contents

List of Figures

List of Tables

Acronyms

AI - Artificial Intelligence

ANN - Artificial Neural Networks

API - Application Programming Interface

ASCII - American Standard Code for Information Interchange

BBC - British Broadcasting Corporation

CNN - Convolution Neural Networks

GPU - Graphics Processing Unit

IDE - Integrated Development Environment

LSTM - Long Short Term Memory

ML - Machine Learning

NER - Named Entity Recognition

NLP - Natural Language Processing

NN - Neural Networks

OCR – Optical Character Recognition

POS - Parts of Speech

ReLU - Rectified Linear Activation

RNN - Recurrent Neural Networks

CHAPTER ONE

1. INTRODUCTION

1.1. BACKGROUND OF STUDY.

Natural Language Processing (NLP) is a field that explores human-computer interaction through the processing of human language, with applications in linguistics, business, education, and beyond. NLP enables tools like grammar checkers and auto-correct functions, such as Grammarly, which detect errors in grammar, spelling, and sentence structure [1]. Recent advancements in neural networks (NN), a subset of machine learning (ML), have significantly enhanced NLP capabilities [2]. However, most NLP research focuses on widely spoken languages like English, with limited work on under-resourced languages like Afan Oromo due to scarce resources [3,4].

Afan Oromo, a Cushitic language spoken by approximately 50 million people primarily in Ethiopia and Kenya, is the fourth most widely spoken African language after Arabic, Hausa, and Swahili [3]. It is the official language of Oromia state in Ethiopia, used in courts, schools, and administration, and is taught in 14 Ethiopian universities [23]. Since 1991, Afan Oromo has adopted the Latin-based Qubee script, consisting of 28 consonants and 5 vowels, including glottalized consonants (e.g., ‘q’, ‘x’) and long vowels (e.g., ‘aa’) [2]. Despite its widespread use, Afan Oromo lacks advanced NLP tools like spell checkers due to its recent script adoption and resource scarcity [3].

Spelling correction is critical for NLP applications such as text summarization, sentiment analysis, and machine translation. Neural networks have shown promise in sequence-to-sequence tasks for under-resourced languages [6], making them suitable for developing an Afan Oromo spell checker.

Table 1.1: Afaan Oromo consonants.

Bb	**Cc**	**CH ch**	**Dd**	**DH dh**	**Ff**	**Gg**	**Hh**	**Jj**	**Kk**
Ll	Mm	Nn	NY ny	Pp	PH ph	Qq	Rr	Ss	SH sh
Tt	TS ts	Vv	Ww	Xx	Yy	Zz	ZY zy		

Table 1.2: Afaan Oromo vowels.

Ii	II ii
Oo	OO oo
Uu	UU uu

Table 1.3: Short and long vowels

Short Vowels	Long Vowels
Aa	AA aa
Ee	EE ee

When compared to other widely spoken languages, less natural language processing applications such as spell checker has been developed for Afaan Oromoo. The main reason is that the language is under-resourced in terms of availability of language experts and tools [3].

Spelling correction system detects a spelling error and proposes a set of candidates for correction. Spelling correction is important for many of the potential NLP applications such as text summarization, sentiment analysis, and machine translation. Automatic spelling correction is crucial in search engines as spelling mistakes are very common in user-generated text. Many websites have a feature of automatically giving correct suggestions to the misspelled user queries. Providing suggestions makes it convenient for users to accept a proposed correction without retyping or correcting the query. The task of spelling correction is challenging for resource-scarce languages such as Afaan Oromoo. Neural network techniques have shown enormous success in sequence-to-sequence mapping tasks in other under-resource languages [6].

Neural networks provide powerful new tools for NLP applications such as Spell Checking, Text Classification and Categorization, Named Entity Recognition (NER), Part-of-Speech Tagging, Semantic Parsing and Question Answering, Paraphrase Detection, Language Generation and Multi-document Summarization, Machine Translation, Speech Recognition and Character Recognition, and have been used both to improve the state-of-the-art in a number of tasks and to tackle new problems.

Combining the techniques and opportunities in spelling correction for resource scarce language together with neural networks could be ideal approach to build spell checker.

1.2. Motivation

This thesis addresses spelling errors in Afan Oromo observed in social media, government documents, and public signage, such as banners and electronic displays in institutions, malls, and cafeterias. Previous research, such as Dida et al. [8], used dictionary lookup with limited word corpora from textbooks, which cannot suggest corrections for unknown words. The need for a robust spell checker that handles Afan Oromo's complex morphology motivates this work, aiming to benefit both native and non-native speakers.

1.3 Thesis Gap Analysis

Both native and non-native speakers struggle to write Afan Oromo correctly due to its recent script adoption and inflectional complexity [1]. Traditional spell-checking methods like dictionary lookup [26] and rule-based approaches [10] are inadequate for Afan Oromo's morphology, requiring extensive manual vocabulary updates. These methods also fail to suggest corrections for non-word errors. For instance, Wubetu et al. [26] applied dictionary lookup, suggesting multiple words without reliable ranking due to the lack of robust language models. Megersa et al. [10] used rule-based methods, which cannot handle words outside predefined rules. This thesis aims to fill these gaps by applying neural networks, which can predict and suggest corrections for untrained words, as no prior work has applied neural networks to Afan Oromo spell checking [3,8,9].

1.4. Statement of the Problem.

Current spell-checking tools for Afan Oromo are limited in accuracy and contextual understanding, often failing to correct complex spelling errors due to the language's inflectional morphology and recent script adoption. This research aims to develop a neural network-based approach that addresses these challenges, improving the accuracy and reliability of Afan Oromo spell-checking systems.

1.5. Research Questions.

This study answered the following research questions to come up with the solution for the misspelling problems:

1. What neural network architecture is most effective for Afan Oromo spell checking and correction?
2. How can neural networks improve the contextual understanding of Afan Oromo language spelling errors?
3. What is the impact of training data size and quality on the performance of neural network models for Afan Oromo spell checking?
4. How does the proposed model compare to existing spell-checking systems in terms of accuracy and error correction?

1.6. Objectives of the Study.

1.6.1. General Objectives.

The aim of this study is to develop and evaluate neural network-based approaches for accurate spell checking and correction in Afan Oromo, with the goal of enhancing the language's digital tools and resources.

1.6.2. Specific Objectives.

To achieve the above general objective, the following tasks are aimed:-

- Design a neural network architecture tailored for Afan Oromo spell checking.
- Build a comprehensive Afan Oromo language corpus for training and testing the model.
- Implement and fine-tune the neural network model for optimal performance in error detection and correction.
- Evaluate the model's accuracy and compare it with existing spell-checking tools.

1.7. Methodology.

An experimental research methodology was employed in two phases: exploratory, to identify problems and design research questions, and evaluative, to answer these questions. A train/test split (75%/25%) was used to validate the model, allowing assessment of performance on unseen data. Data was collected from BBC Afan

Oromoo using Sketch Engine, selected for its professional content and spelling accuracy compared to social media or textbooks [8,10,26]. A recurrent neural network (RNN) with Long Short-Term Memory (LSTM) layers was trained using Python, combining edit distance techniques with a language model to detect and correct errors. Tools included TensorFlow/PyTorch for model building, NLTK/Spacy for preprocessing, and Hugging Face Transformers for exploring advanced architectures like BERT.

The research began with a comprehensive review of literatures and industry practices in relation to Afaan Oromoo spelling correction and found a gap in adapting methods used for other languages such as English for under resource languages. The core of the work involved building the training model to work with data collected from BBC Afaan Oromoo. The website is selected to overcome data collection weakness used by previous authors. The first method that was used by Megarsa and Wubetu [10,26] was collected from social medias, most social medias users does not spell Afaan Oromo words correctly as they just use it for communication and this is will lead to have incorrect word corpus. In other method that was used by Dr. Dida et al [8], data were collected from Afaan Oromo text books. This method better when it is compared to social medias in terms of accuracy. But only limited data can be collected from the text books and this affects the performance of the model. On the other hands, various news on the BBC Afaan Oromoo is published every day by professionals by taking care of spellings. Therefore, many correct words can be collected from the website to develop word corpus.

Recurrent neural network with Long Short-Term Memory (LSTM) [11] is used for training determining structure of the network. Edit distance [16] is a technique that can be used in order to detect spelling errors in a text by measuring the amount of characters that two words differ by. An edit distance of 1 usually indicates a misspelled word, but in many cases a misspelled word is the same as another, correctly spelled word. This means that edit distance alone is not enough to accurately correct spelling or other grammar mistakes. Our approach will involve combining edit distance with a language model, in order to accurately detect words that are out of place, even if they exist in the training.

A testbed was settled up for collecting the data, training and evaluation. In this thesis, Python was used as a main experimental language for implementing/simulating the model using PyCharm IDE. Detail implementation of this approach will be discussed in chapter 3.

1.8. Scope/Limitation of the Study.

This thesis focuses on Afan Oromo spell checking using data from BBC Afan Oromoo. The corpus size is limited to 596,948 words due to time and hardware constraints, preventing the use of larger datasets. While the model could be adapted for other under-resourced languages, this work is specific to Afan Oromo. A fully-featured application was not developed due to resource limitations.

Training was conducted on a consumer-grade laptop with an Intel i7 processor, 16GB RAM, and no GPU, leading to an average training time of 4.5 hours for 100 epochs with a batch size of 128. Energy consumption was estimated at 0.3 kWh per epoch, limiting scalability in resource-constrained settings. Future work could leverage GPU/TPU hardware to reduce training time and enable larger datasets. The absence of transfer learning (e.g., multilingual BERT) was due to computational limitations, though it could mitigate corpus size constraints.

1.9. Significance of the Study.

This thesis provides a state-of-the-art neural network-based spell checker for Afan Oromo, benefiting approximately 50 million speakers, including students in over 10,000 Oromia schools. It creates a comprehensive language corpus for future NLP research and offers insights for developing spell checkers for other under-resourced languages, contributing to Ethiopia's digital linguistic ecosystem.

1.9. Organization of the Study.

The rest of this thesis is organized as follows: Chapter 2 discusses Literature Review; Chapter 3 presents the Methodology; Chapter 4 Experimentation and result; and Chapter 5 presents the Discussion of results, Conclusion and the future works presented at the end.

CHAPTER TWO

2. REVIEW OF RELATED LITERATURE.

In this section literatures related to NLP, Machine learning, Spell Checker, Neural network, Artificial Intelligence, and nature of Afaan Oromo language was reviewed.

2.1 Afaan Oromoo Language

Afaan Oromo is a Cushitic language spoken by about 50 million people in Ethiopia, Kenya, Somalia and Egypt and is the 3rd largest language in Africa. The Oromo people are the largest ethnic group in Ethiopia and account for more than 34.5% of the population [51]. They can be found all over Ethiopia and particularly in Wollega, Shoa, Illubabour, Jimma, Arsi, Bale, Hararghe, Wollo, Borana, and the southwestern part of Gojjam.

2.1.1. Afaan Oromo Writing System

There was no official writing system (even though both Sabean and Geez were used) till 1991 [1]. With regard to the writing system, Qubee (a Latin-based alphabet) has been adopted and become the official script of Afaan Oromo since 1991.

2.1.2. Afaan Oromo Character Representation

Afaan Oromo is a phonetic language, which means that it is spoken in the way it is written. In addition to Latin alphabet (except p, v) Afaan Oromo uses its own consonants and vowels and it has 34 letters called 'qubee'. [49]

Letter **Qubee**	**Sound** **Sagalee**	**Example** **Fakkeenya**
A a	short ah sound as in again or what	abalu
B b	unstressed b as in body, about	boba'uu
C c	hard, glottalized tch sound	ciccitaa
D d	stressed d sound as in dad	dadaa

E e	e sound as in pen or empty	eger
F f	unstressed f as in five or after	faarfannaa
G g	unstressed g as in game or ago	goggogaa
H h	unstressed h as in hammer	hahaaraa
I i	short i as in hit or in	isin
J j	unstressed j as in jump or agency	jejjuu
K k	unstressed k as in coco	kookii
L l	unstressed l as in little	laallee
M m	unstressed m as in member	mimmixa
N n	unstressed n as in no>b>ne	naannoo
O o	O sound as in sore or open	obboleessa
P p	unstressed p sound as in paper	paappaayyaa
Q q	hard, glottalized k	qaqqabuu
R r	slightly rolling, soft r as in sparrow	roorroo
S s	unstressed s sound as in Susan	seenessa
T t	unstressed t as in tape	tattaa'ii
U u	oo sound as in who or Spanish uno	udumuu
V v	unstressed v as in avenue or very	viizaa
W w	unstressed, soft w sound as in now or wind	wawwaachuu
X x	hard, glottalized t	xaaxee
Y y	unstressed y as in year or bayou	yayii
Z z	unstressed z as in zigzag	zeeroo
Ch ch	slightly stressed ch as in chase	cheenchii
Dh dh	glottalized d produced with the tongue curled back	dhadhaa
Ph ph	glottalized p as in pope (said without breathing)	phaaphaasii
Sh sh	unstressed sh sound as in should	shaashii

Ny ny	like the Spanish ñ, like onion or cognac	nyanyee
Ts ts		
Zh zh		

Diphthongs and Long Vowels

aa — as in father, water, army
aw — as in cow or ouch
ay — as in aisle or pie
ee — as in eight or gray
ii — as in evil or teepee
oo — long o as in oboe or sober
oy — as in boy
uu — long oo as in fool or spoon.

Glottalized Consonants

The glottalized consonants are c, q, x, and ph. These can be described as explosive ch, k, t, and p sounds, respectively

In pronouncing the glottalized consonants, the stream of air coming from the lungs is shut off by closure of the glottis. The air about it is then forced out through a stricture somewhere along the vocal organ. This stricture is a the lips for [ph], at the teeth for [x], at the palate for [c], and at the velum for q.[1]

Double Letters

Vowels and consonants may be repeated to make the sound long. For example, to say the Oromo word annan ("milk") one must hold the first n sound slightly longer than the second, as in the English word "pen-knife". A doubled vowel makes the vowel long and can often change the meaning of the word, as in lafa ("ground") and laafaa ("soft"). Dh, ch, ph, sh, and ny count as single consonants though they are written as two letters.

Spelling Rules

Traditionally, Oromo was written using Ge'ez script as used by Amharic. In 1991, the Oromo Peoples' Democratic Organization formally adopted a modified Latin alphabet (qubee) as shown at the beginning of this chapter. This qubee replaced the various other

transliteration schemes of Oromo to Latin script and helped to standardize spelling of Oromo words. Spelling differences still occur, however, due to personal preferences and dialectal differences. Regardless, certain spelling rules can be observed that match speech patterns.

A word in Oromo cannot begin or end with a double consonant. The word for “sport” is converted to isporti.

Three consonants cannot occur in a row in a word. For this reason, certain suffixes may add an i to prevent this, as in arg (“see”) + na (1st per. plu. suffix) → argina (“we see”).

Vowels cannot change without a break, either a consonant or apostrophe, between them. What breaks are used can differ with spelling preferences and dialects. For example, “very” can be baa'ee, baayee, baa'yee, or baay'ee, and “to hear” can be dhaga'uu or dhagahuu. The apostrophe indicates that the vowels are produced independently and not as a diphthong.

The Afaan Oromo vowels always are pronounced in sharp and clear fashion which means each and every word is pronounced strongly, for example:

A: Arba, Farda, Haadha

E: Gannale, Waabee, Noole, Roobale, colle

I: Arsii, laali, Rafi, Lakkii, Sirbii

O: Oromo, Cilaalo, Haro, caancco, Danbidoollo

U: Ulfaadhu, Gudadhu, dubadhuu, **arbbaguugu**, Ituu

CONSONANTS - SAGALEEWWAN

Most Afaan Oromoo constants do not differ greatly from Latin, but there are some exceptions and few special combinations.

A. The consonant "g" has a hard sound. Gaari, gadi bayi, gargaari.

B. The combinations NY and DH have a hard sound. e.g. Nyaadhu, Dhugi.

DOUBLE CONSONANTS - SAGALEWWAN DACHAA

All Afaan Oromo consonants except the combination consonants: ch, ny, dh, ph and sh have double consonant combinations if the syllable is stressed.

Failure to make this distinction results in miscommunication. Examples: Bilisumma, walqixumma, walabumma, fardda, lolttu

STRESS - JABAA

Some Afaan Oromo words are pronounced with the stress on the last syllable:

e.g. fanno, harre, gaarre.

On the other hand, few words are stressed on the first syllable. These words always have a combination consonant:

e.g. nyaadhu, dhayi, nyaaranyaapha (foreigner).

2.1.3. Punctuation Marks.

Afan Oromo borrows all punctuation marks from foreign languages such as (?, ! , “ , ”, ‘, /, \, etc.). But character “ ‘ ” which shows possessor in English language differently used in Afan Oromo. It is the character which has no its own sound but used as consonant in afan Oromo language (example re’ee mean goat). If we write without this character (reee) the words become meaningless.

2.1.4. Morphology.

Morphology is a branch of linguistic that studies and describes how words are formed inlanguage [50]. There are two kinds of morphology: inflectional and derivational. Inflectional morphology is concerned with the inflectional changes in words where word stems are combined with grammatical markers for things like person, gender, number, tense, case and mode. Inflectional changes do not result in changes of parts of speech. Derivational morphology deals with those changes that result in changing classes of words (changes in the part of speech). For instance, noun or an adjective may be derived from a verb.

2.1.5. Types of morphemes in Afaan Oromoo

There are two categories of morphemes: free and bound morphemes. Free morpheme can stand as a word on its own whereas bound morpheme does not occur as a word on its own [50]. In Afaan Oromo roots are bound as they cannot occur on their own like dhug- (drink) and beek-(know), which are pronounceable only when other completing affixes are added to them. In other words these roots serve as base stems in Afaan Oromo since them posse’s non-verbalized glosses [51].

Like the root, an affix is also a morpheme that cannot occur independently. It is attached in some manner to the root, which serves as a base. These affixes are of three types-prefix, suffix and infix. The first and the second types of affixes occur at the beginning and at the end of a root respectively in form a word. In beekumsa (knowlwdge), for instance, -umsa is a suffix and beek-(know) is a stem. An infix is a morpheme that is inserted within another morpheme. Like English [52], Afan Oromo does not have infixes [49].

In general there are two main challenges in implementing NLP for Afaan Oromo. First low resource, being the writing system was adopted just decades ago there is a limitation both in terms of data and experts. Second lack of research and development, Machine learning requires a lot of data to function to its outer limits – billions of pieces of training data. The more data NLP models are trained on, the smarter they become [20]. Data is only growing by the day, as are new machine learning techniques and custom algorithms. All of these problems will require more research and new techniques in order to improve on them.

Advanced practices like artificial neural networks and deep learning allow a multitude of NLP techniques, algorithms, and models to work progressively, much like the human mind does [36].

2.2 Natural Language Processing.

Natural Language Processing (NLP) deals with how computers understand and translate human language [12]. Natural Language Processing (NLP) applies two techniques [17] to help computers understand text: syntactic analysis and semantic analysis. Syntactic analysis – or parsing – analyses text using basic grammar rules to identify sentence structure, how words are organized, and how words relate to each other. Semantic analysis focuses on capturing the meaning of text. First, it studies the meaning of each individual word (lexical semantics). Then, it looks at the combination of words and what they mean in context. With NLP, machines can make sense of written or spoken text and perform tasks like spell checker, translation, keyword extraction, topic classification [1]. But to automate these processes and deliver accurate responses, machine learning is required [2].

2.3 Machine Learning.

Machine learning is the process of applying algorithms that teach machines how to automatically learn and improve from experience without being explicitly programmed. In machine learning, some training data are added which trains the computer. It uses the data for creating a model, and as it gets new input, it uses them to make predictions. If the prediction turns out to be wrong, the computer re-starts the process again until it makes a right prediction [20]. Neural network techniques enable this automatic learning through the absorption of huge amounts of unstructured data such as text, images, or video.

2.4 Artificial Intelligence.

Both NLP and machine learning are subset of artificial intelligence (AI) [12]. AI is an umbrella term for machines that can simulate human intelligence [13]. AI encompasses systems that mimic cognitive capabilities, like learning from examples and solving problems. This covers a wide range of applications, from self-driving cars to predictive systems.

2.5 Spell Checker.

Spell Checker is an application program that flags words in a document that may not be spelled correctly. When some text is given as an input, the spell checker has an error detection which list outs the incorrect words separately by checking their availability in the dictionary and error correction which provides the suggestions for the incorrect words from the dictionary [14].

For error detection each word in a sentence or paragraph is tokenized by using a tokenizer and checked for its validity. The candidate word is a valid if it has a meaning else it is a non-word [15]. Two commonly used techniques for error detection is dictionary/WordNet lookup and N-gram analysis.

A dictionary/WordNet is a lexical source that contains list of correct words a particular language. The non-word errors can be easily detected by checking each word against a dictionary. Dictionary lookup technique checks every word of input text for its presence in dictionary. If that word present in the dictionary, then it is a correct word. Otherwise it is put into the list of error words. The most common technique for gaining fast access to a dictionary is the use of a Hash Table [15]. Hash tables main advantage is their random access nature that eliminated the large number of comparisons needed to search

the dictionary. The main disadvantage of hash table is that the need to develop a clever hash function that avoids collisions. To look up an input string, one simply computes its hash addresses and retrieves the word stored at that address in the pre- constructed hash table. If the word stored at the hash address is different from the input string or is null, a misspelling is indicated. The drawbacks of this method are difficulties in keeping such a dictionary up to date, and sufficiently extensive to cover all the words in a text. These resources are used for preparing, processing and managing linguistic information and knowledge needed for the computational processing of natural language. An example of such large scale lexical resources is given by linguistic ontology that covers many words of a language and has a hierarchical structure based on the relationship between concepts.

N-gram are n-letter sub sequences of words or strings where n usually is one, two or three. One letter n grams are referred to as unigrams or monograms; two letter n-grams are referred to as bi-grams and three letter n-grams as trigrams. In general, n-gram detection technique work by examining each n-gram is an input string and looking it up in a precompiled table of n-gram statistics to as certain either its existence or its frequency of words or strings that are found to contain nonexistence or highly infrequent n-grams are identified as either misspellings.

The major advantage of n-gram algorithm is that it requires no knowledge of the language that is used with, so it is language independent or it is a neutral string matching algorithm. Using n-grams to calculate, for example the similarity between two strings is achieved by discovering the number of unique n-grams that they share and then calculating a similarity coefficient, which is the number of the n-grams in common divided by the total number of n-grams in the two words.

Rakesh Kumar [15] suggested in his studies that N-gram analysis is a method to find incorrectly spelled words in a mass of text. Instead of comparing each entire word in a text to a dictionary, just n-grams are checked. A check is done by using an n-dimensional matrix where real n-gram frequencies are stored. If a non-existent or rare n-gram is found the word is flagged as a misspelling, otherwise not. An n-gram is a set of consecutive characters taken from a string with a length of whatever n is set to. This method is language independent as it requires no knowledge of the language for which it is used. In this algorithm, each string that is involved in the comparison process is

split up into sets of adjacent n-grams. The similarity between two strings is achieved by discovering the number of unique n-grams that they share and then calculating a similarity coefficient, i.e. the number of the n-grams in common (intersection), divided by the total number of n-grams in the two words (union).

Error correction/Spelling correction is a process of detecting and providing suggestions for misspelled words in a text. Two types of spelling error correction: interactive and automatic. In interactive, the spellchecker can suggest more than on correction for each misspelled word and the user decide to select for replacement. In case of automatic correction, the spellchecker has to decide on the one best correction and the error is automatically replaced with it [7]. Error correction consists of two steps: the generation of candidate corrections and the ranking of candidate corrections. The candidate generation process usually makes use of a precompiled table of legal n-grams to locate one or more potential correction terms. The ranking process usually invokes some lexical similarity measure between the misspelled string and the candidates or a probabilistic estimate of the likelihood of the correction to rank order the candidates. These two steps are most of the time treated as a separate process and executed in sequence. Some techniques can omit the second process though, leaving the ranking and final selection to the user [14]. Two methods of spelling error correction: isolated word error correction and context-based error correction. In isolated word error correction, misspelled word can be analyzed in isolation without giving consideration to its context. Corrections are based on only misspelled words itself. In case of context-based error correction, correction can be done with consideration of the context of the error word Context based technique mainly used for correcting real word errors. Real word errors are words which have correct meaning in the dictionary but contextual error in a given text. The isolated-word methods are the most studied spelling correction algorithms, they are: edit distance [17], similarity keys [18], rule-based techniques [9, 10], n-gram-based techniques [11], probabilistic techniques [41], noisy channel model [17] and neural networks [7]. All of these methods can be thought of as calculating a distance between the misspelled word and each word in the dictionary or index. The shorter the distance the higher the dictionary word is ranked [15].

2.5.1. Spelling Error Correction Techniques.

2.5.1.1. Edit Distance Techniques.

Edit distance between two characters is the minimum cost of a sequence of editing operations which transforms one string into the other. Using edit distance algorithm, it is possible to perform deleting, inserting and replacing one symbol a time with possibly different costs for each of these operations. It is one of the simplest methods based on the assumption that the person usually makes few errors i.e. only one erroneous character operation (insertion, deletions, and substitutions) necessary to covert a dictionary word in to the non-word. Edit distance is useful for correcting errors resulting from keyboard input, since these are often of the same kind as the allowed edit operations. It is not quite as good for correcting phonetic spelling errors [14], especially if the difference between spelling and pronunciation is big for a language.

There are many distance measures [17]. Most known are: Hamming distance, Levenshtein distance and Damerau distance.

Hamming distance is a metric between two strings of equal length. It measure the distance between two strings of equal length. For instance, the hamming distance between “mirga” and “marga” is 1 (changing i to a).

Levenshtein distance is a mathematical measure for calculating the difference between two strings. It is a weighting approach to appoint a cost of 1 to every edit operations (Insertion, deletion and substitution). For instance, the Levenshtein edit distance between “baala” and “firii” is 5 (substituting b by f, a by I, a by r, l by i, a by i). Damereu distance is an extension of the Levenshtein distance by adding new operation of transition. Damereu distance [14] is a well-known metrics for making spelling corrections through string-to string comparison. Jaro-Winkler distance has been designed and is best suited for short strings [17].

Hodge et al [44] have integrated Hamming Distance and n-gram algorithms to design architecture for spell checker with high recall for typing errors and a phonetic spell-checking algorithm.

For Hamming Distance and n-gram, the word spellings form the inputs and the matching words from the lexicon form the outputs. For the inputs, they divide a binary vector of length 960 into a series of 30-bit chunks. Words of up to 30 characters may be

represented, (two additional character chunks are needed for the shifting n-gram). Each word is divided into its constituent characters. The appropriate bit is set in the chunk to represent each character in order of occurrence. The chunks are concatenated to produce a binary bit vector to represent the spelling of the word. Any unused chunks are set to all zero bits.

Each word in the alphabetical list of all words in the text corpus has a unique orthogonal binary vector to represent it. They associate the spelling of the word to an orthogonal output vector to uniquely identify it and maintain the system's high speed as no output validation is necessary. They claim that the approach is suitable for any spell checking application though aimed toward isolated word error correction, particularly spell checking user queries in a search engine. Authors state that they use a novel scoring scheme to integrate the retrieved words from each spelling approach and calculate an overall score for each matched word.

Muhammad Maulana et al [45] have applied Levenshtein Distance Algorithm to provide the auto complete text suggestion and spell checking using Levenshtein. They aimed to apply the autocomplete feature and spell checking with Levenshtein distance algorithm to get text suggestion in an error data searching in library and determine the level of accuracy on data search trials. They use 1155 data obtained from UNNES Library. The variables are the input process and the classification of books. The claim that the accuracy of Levenshtein algorithm is 86% based on 1055 source case and 100 target case.

In a paper Portable Spelling Corrector for a Less-Resourced Language: Amharic, Andargachew M. *et al* [46] have applied Damerau-Levenshtein edit distance for Amharic spell checker to measure nearness between words. In their proposed approach, the spelling error detection and correction processes are as follows. An input word that is not in the term list, which is compiled from the most frequent words in a text corpus, is flagged as a spelling error. Candidate corrections that are closer (nearer) to the misspelling are generated from the term list. Since most of the misspellings fall within two edit distance from their corrections [46], they selected all words in the term list that are one up to two edit distance from the misspelled word. Then the candidates are scored and ranked according to their prior and likelihood probabilities. In case there is

no candidate correction, the misspelled term will be split. This step is needed to correct misspellings result from missed out spaces between words.

2.5.1.2. Similarity Keys.

Similarity key technique is to map every string into a key such that similarly spelled strings will have similar keys. Thus when key is computed for a misspelled string it will provide a pointer to all similarly spelled words in the lexicon. An index/key is assigned to each dictionary word for comparing with the key computed for the non-word. The word for which the keys are most similar are selected as suggestion .such an approach is speed effective [18] as only the words with similar keys have to be processed With a good transformation algorithm this method can handle keyboard errors.

The SOUNDEX system and SPEEDCOP System are similarity key technique.

Soundex Algorithm: This algorithm is used for indexing words based on their phonetic sound. Words with similar pronunciation but different meaning are coded similarly so that they can be matched regardless of trivial differences in their spelling.

The SPEEDCOP System: It is a way of automatically correcting spelling errors predominantly typing errors in a very large database of scientific abstracts. A key was computed for each word in the dictionary. This consisted of the first letter, followed by the consonants letters of the word, in the order of their occurrence in the word, followed by the vowel letters, also in the order of their occurrence, with each letter recorded only once.

2.5.1.3. Rule Based Techniques.

Rule Based Techniques are algorithms that attempt to represent knowledge of common spelling errors patterns in the form of rules for transforming misspellings into valid words. The candidate generation process consists of applying all applicable rules to a misspelled string and retaining every valid dictionary word those results [9]. It works by having a set of rules that capture common spelling and typographic errors and applying these rules to the misspelled word. Intuitively these rules are the “inverses” of common errors. Each correct word generated by this process is taken as a correction suggestion. The rules also have probabilities, making it possible to rank the suggestions by accumulating the probabilities for the applied rules. Edit distance can be viewed as a special case of a rule-based method with limitation on the possible rules [10, 17].

Daniel Naber [47] proposed a Rule-Based Style and Grammar Checker. The grammar checking is done on the basis of Rule-based checking .This is the approach where we match the text with a set of rules and that has been at least POS tagged. The rules are in accordance with the grammar of the language of interest. Advantage of this approach [47] is as follows: This method gives immediate feedback. It is easy to configure, as each rule can be turned on and off individually. It can give detailed error messages with comments, even can explain grammar rules. It is easily extendable by the users, as the rule system is easy to understand. Its extension is easy, starting with just one rule and then extending it rule by rule.

2.5.1.4. N-gram based Techniques.

Letter n-grams, including tri-grams, bi-grams and unigrams have been used in a variety of ways in text recognition and spelling correction techniques [14]. They have been used by OCR correctors to capture the lexical syntax of a dictionary and to suggest legal corrections.

N-grams can be used in two ways [11], either without a dictionary or together with a dictionary. Without a dictionary, n-grams are employed to find in which position in the misspelled word the error occurs. If there is a unique way to change the misspelled word so that it contains only valid n-grams, this is taken as the correction. The performance of this method is limited [11]. Its main virtue is that it is simple and does not require any dictionary.

Together with a dictionary, n-grams are used to define the distance between words, but the words are always checked against the dictionary. This can be done in several ways, for example check how many n-grams the misspelled word and a dictionary word have in common, weighted by the length of the words.

In paper Automatic Spelling Correction based on n-Gram Model, S. M. El *Atawyet al* [11] have proposed a model to spell error detection and auto-correction that is based on n-gram technique and it is applied in error detection and correction in English as a 'global' language. The proposed model provides correction suggestions by selecting the most suitable suggestions from a list of corrective suggestions based on lexical resources and n-gram statistics. It depends on a lexicon of Microsoft words. The evaluation of the proposed model uses English standard datasets of misspelled words. Error detection, automatic error correction, and replacement are the main features of the

proposed model. They claim results of the experiment reached approximately 93% of accuracy and acted similarly to Microsoft Word as well as outperformed both of Aspell and Google.

2.5.1.5. Probabilistic Techniques.

Probabilistic technique works based on some statistical features of the language. Two common methods are transition probabilities and confusion probabilities. Transition Probabilities represent that a given letter will be followed by another given letter. These are dependent. They can be estimated by collecting n-gram frequency statistic on a large corpus of text from the discourse. Confusion Probabilities estimates of how often a given letter is mistaken or substituted for another given letter. Confusion probabilities are source dependent because different OCR devices use different techniques and features to recognize characters, each device will have a unique confusion probability distribution.

Transition probabilities are similar to n-grams. They give us the probability that a given letter or sequence of letters is followed by other given letter. Transition probabilities are not very useful when we have access to a dictionary or index [41]. Given a sentence to be corrected, the system decomposes each string in the sentence into letter n-grams and retrieves word candidates from the lexicon by comparing string n-grams with lexicon entry n-grams. The retrieved candidates are ranked by the conditional probability of matches with the string, given character confusion probabilities. Finally, a word-bigram model and a certain algorithm are used to determine the best scoring word sequence for the sentence.

R. Kashyap et al [41] suggested probabilistic procedure for the automatic correction of spelling and typing errors in printed English texts. The heart of the procedure is a probabilistic model for the generation of the garbled word from the correct word. The garbler can delete or insert symbols in the word or substitute one or more symbols by other symbols. An expression is derived for P(Y I X), the probability of generating a garbled word Y from a correct word X. The model is probabilistically consistent. Using the expression for P(Y I X), an estimate of the correct word can derived from the garbled word Y so as to minimize the average probability of error in the decision. One of the important features of the expression P(Y I X) is that it can be computed recursively. They conducted experiments using the dictionary of 1025 most common

English words and indicated that the accuracy of correction by this scheme is substantially greater than that which can be obtained by other algorithms [41] especially while dealing with garbled words derived from relatively short words of length less than six.

Yaregal Assebie et al [48] propose n-gram based Amharic spell checker using probabilistic method. They describe the design and development of statistical grammar checker for Amharic by treating its morphological features. In a given Amharic sentence, the morphologies of individual words making up the sentence are analyzed and then n-gram based probabilistic methods are used to check grammatical errors in the sentence. In their approach, grammar checker for Amharic text passes through three phases: Checking word sequences, checking adjective-noun-verb agreements, checking adverb-verb agreement. In the first two phases, we employ the n-gram based statistical method. The n-gram probabilities are computed from the linguistic properties of words in a sentence.

2.5.1.6. Noisy Channel Model

The intuition of the noisy channel model is to treat the misspelled word as if a correctly spelled word had been distorted by being passed through a noisy communication channel [17]. This channel introduces noise in the form of substitutions or other changes to the letters, making it hard to recognize the true word. The goal, then, is to build a model of the channel. Given this model, we then find the true word by passing every word of the language through our model of the noisy channel and seeing which one comes the closest to the misspelled word.

The noisy channel model says that we have some true underlying word w, and we have a noisy channel that modifies the word into some possible misspelled observed surface form. The likelihood or channel model of the noisy channel producing any particular observation sequence x is modeled by P(x|w). The prior probability of a hidden word is modeled by P(w). We can compute the most probable word $\hat{w}$ given that we've seen some observed misspelling x by multiplying the prior P(w) and the likelihood P(x|w) and choosing the word for which this product is greatest [14].

2.6 Neural Networks.

Neural Network (NN) is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. It is inspired by the structure

of the brain. The neural network contains highly interconnected entities, called units or nodes. Neural networks are deep learning technologies. It generally focuses on solving complex processes. A typical neural network is a group of algorithms; these algorithms model the data using neurons for machine learning. It (generally) comprised of; Neurons which pass input values through functions and output the result, Weights which carry values between neurons. Neurons are grouped into layers. There are 3 main types of layers; Input Layer, Hidden Layer(s), Output Layer. The Input layer communicates with the external environment that presents a pattern to the neural network. The hidden layer is the collection of neurons which has activation function applied on it and it is an intermediate layer found between the input layer and the output layer. Its job is to process the inputs obtained by its previous layer. We can have n hidden layers. If there are more than one hidden layers in a model it's called deep neural network. The output layer of the neural network collects and transmits the information accordingly in a way it has been designed to give. The pattern presented by the output layer can be directly traced back to the input layer. The number of neurons in output layer should be directly related to the type of work that the neural network was performing. To determine the number of neurons in the output layer, the intended use of the neural network should be considered. There are three important types of neural networks that form the basis for most pre-trained models. These are Artificial Neural Networks (ANN), Convolution Neural Networks (CNN), and Recurrent Neural Networks (RNN). Artificial Neural Network, or ANN, is a group of multiple perceptron's/ neurons at each layer. ANN is also known as a Feed-Forward Neural network because inputs are processed only in the forward direction.

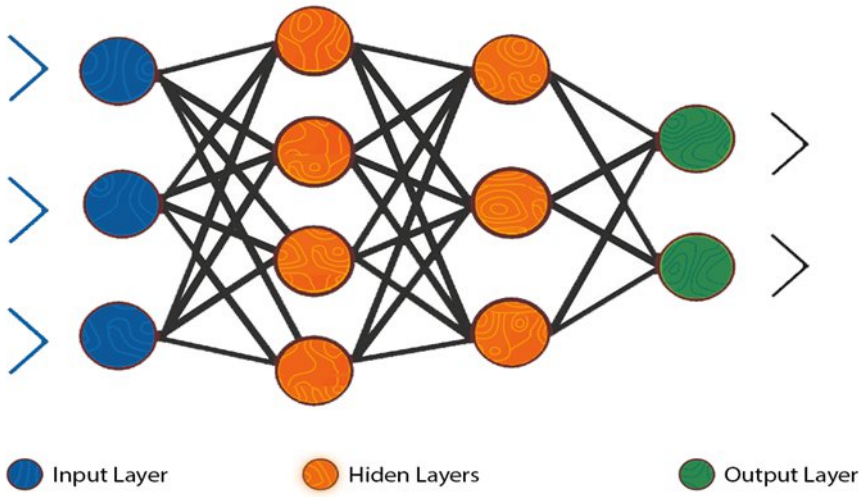


Figure 2.1: Artificial neural network

We can use recurrent neural networks to solve the problems related to time series data, text data and audio data. RNN captures the sequential information present in the input data i.e. dependency between the words in the text while making predictions. Therefore, RNN is ideal to apply in natural language processing. Yet RNNs (RNNs with a large number of time steps) suffer from the vanishing and exploding gradient problem which is a common problem in all the different types of neural networks [2]. Long Short Term Memory (LSM) can be used to solve this problem.

Long Short Term Memory Network is an advanced RNN, a sequential network, that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNN [19].

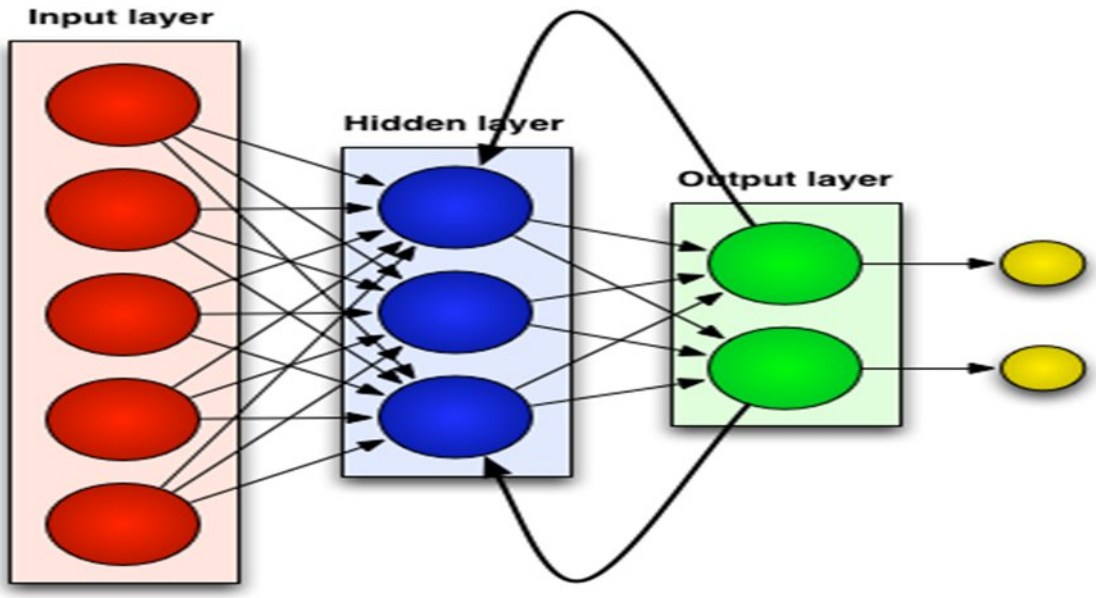


Figure 2.2: Recurrent neural Network

The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks in neural network. Various activation functions such as Rectified Linear Activation (ReLU), Logistic (Sigmoid), Hyperbolic Tangent (Tanh) and softamx can be used based on the application. The softmax function is handy [20] when we are trying to handle classification problems.

… eq 1:- Softmax Function.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax

$\vec{z}$ = input vector

e^{z_i} = standard exponential function for input vector

K = number of classes in the multi-class classifier

e^{z_j} = standard exponential function for output vector

e^{z_j} = standard exponential function for output vector

Recurrent Neural network models such as Sequence2sequence can be applied in NLP.

Sequence2sequence (seq2seq), takes as input a sequence of words (sentence or sentences) and generates an output sequence of words. It does so by use of the neural network (NN).

It is a common model for NLP tasks such as machine translation. It mainly has two components i.e encoder and decoder, and hence sometimes it is called the Encoder-Decoder Network [21].

Encoder: It uses deep neural network layers and converts the input words to corresponding hidden vectors. Each vector represents the current word and the context of the word.

Decoder: It is similar to the encoder. It takes as input the hidden vector generated by encoder, its own hidden states and current word to produce the next hidden vector and finally predict the next word.

As in a word2vec model, this allows the model to learn context such as gender agreement and syntax structure [19], [22]. A seq2seq model is more relevant for our use case than a word2vec model as it can consider each sentence as a whole, and identify grammatical errors as well as spelling mistakes [22].

Neural nets are likely candidates for spelling correctors [7] because of their inherent ability to do associative recall based on incomplete or noisy input. Back Propagation Algorithm is the most widely used algorithm for training a neural net. A typical back propagation net consists of three layers of node: input layer, an intermediate layer, an output layer. Each node in the input layer is connected by a weighted link to every node in the hidden layer. Similarly each node in the hidden layer is denoted by a weighted link to every node in the output layer. Input and output information is represented by on-off patterns of activity on the input and output nodes of the net. A 1 indicates that a node is turned on and 0 indicates that a node is turned off. However, sequence to sequence neural network model is the most widely used in natural language processing [24]. It is a special class of Recurrent Neural Network architectures that we typically use (but not restricted) to solve complex Language problems like spelling checker, Machine Translation, Question Answering, creating Chatbots, Text Summarization, etc

Spell checker development for under resourced languages such as Afaan Oromo can take advantages of neural network.

2.7. Related Works.

Research conducted on designing spell checker or spell correction for Afaan Oromoo and other Ethiopian languages writing is in early stage. To best of our knowledge no work have done on the application of NN for spell checking in Afaan Oromoo language.

The first Afaan Oromo spell checker research was conducted by Dr. Dida Midakso et al [8] in a paper *Design And Implementation Of Morphology Based Spell Checker*, Afaan Oromo spell checker was designed based on a dictionary look-up with morphological analysis. They used module prepared for teaching Afaan Oromo courses to analyze spelling error pattern of Afaan Oromo. Authors used text analysis data gathering technique for this purpose. The finding of study depicts the existence of spelling errors. When analyzed, they found that 1,342 words were misspelled. Out of this 1,287 words were in the category of non-word errors. They claim that it was enough to realize that non-word error detection is the first step towards a truly professional spellchecker. The strength of the work is that it has an ability to reduce the dictionary size and the ability to recognize new words that are not included in the dictionary.

However, developing morphology based Afaan Oromoo spell checker is challenging as Afaan Oromo is an agglutinative and morphologically rich language, each root word can combine with multiple morphemes to generate huge number of word forms. For the purpose of supporting such inflectionally rich languages, the structure of each word has to be identified. Afaan Oromo has compound, derived and simple nouns, verbs, and adjectives. It also has first person, second person, and derived pronouns. Nouns get inflected for number, gender, number, tense, voice, aspect and mood cause inflections to verbs. Many times it is context which decides whether a word is a noun or adjective or adverb or post position. This increases the complexity of parsing Afaan Oromo.

In paper *Integrated Model to Develop Grammar Checker for Afaan Oromo using Morphological Analysis:* A Rule-based Approach Jemal Abate et al [25] have proposed an integrated model to develop a grammar checker for Afaan Oromo using a rule-based. They have integrated a spell-checker along with a morphological analysis on Afaan Oromo grammar checker to improve its performance. They collect a data from various sources such as Oromia Broadcasting Network, BBC Afaan Oromo and other sources [25] used for designing and implementation purposes. They have done

experimentation in two phases: spellchecking & correction phase and grammar-error detection & correction phase. The system is evaluated on detecting wrong spelling and grammar from the text and the possible suggestion made to the error. As a result, the evaluation result for detecting grammar errors in a statement has a low score than other activities. One of the main reasons as they mentioned is due to the uncovered cases within the rule for detecting the error.

The main weakness of the approach which they have also admitted is, it is difficult to catch all of the grammar errors using only a set of rules. They recommend another method such as machine learning methods combined with a rule-based approach to improve the performance of the Afaan Oromo grammar checker.

Rule based Afaan Oromo Grammar Checker was designed by Dr. Debela Tesfaye [9] in order to form well organized arrangement of words in Afaan Oromo sentence. The author has constructed and used 123 different rules in order to identify grammatical error of the language. With the use of these carefully constructed error detection rules, the system can detect and suggest corrections for a number of grammatical errors in Afaan Oromoo texts. But this cannot hinder the problem of misspelling in Afaan Oromoo, since grammar focus on the construction of the sentence, while this spelling checker focuses on the contexts to check spelling error at the words level the author proposed.

Wubetu B. et al [26] proposed a dictionary lookup approach to design spelling checker for AfaanOromoo and other selected Ethiopian Languages. Authors state that hash tables are the most common implemented methods to gain fast access to a dictionary. In order to lookup a string, one has to calculate its hash address and recover the word deposited at that address in the pre-built hash table. If the word kept at the hash address is dissimilar from the input string, a misspelling is flagged. The main drawback of hash table as authors also accepted is its arbitrary access nature that eliminated the large number of assessments required to exploration of the dictionary files. To store a word in the dictionary, it computes the respectively hash function for the word and fixed the vector entries equivalent to the intended values to true. To find out if a word fits to the dictionary, it computes the hash values for that word and looks it in the vector. If all entries equivalent to the values are true, then the word fits to the dictionary, else it does not. The weakness of this work is that the method is subject to both non-word error, an

error that happen when the word is not found in the list of the dictionary and real-word error, an error happens when the word is correctly spelled since it matches with the list of bigram dictionary yet it appears in wrong position in the sentence.

Chakraborty [12] showed that the amounts of words in a dictionary determine the effectiveness of finding misspellings in a text. Having a small dictionary will lead to a lot of correct words being marked as incorrect. Using a larger dictionary will solve this, but increases the time it takes to look up words. According to the result the enlarged access times can be solved by exploiting hash tables, tries, and other optimized storage mechanisms. The approach does not fit for under-resource languages that have no word corpus.

Megersa D. et al. [10] proposed rule based spell checking approach to guide and manage how to write AfaanOromoo words without making mistake based on certain written rules. It directly focuses on rules provided to correct errors. Authors proved that its space and time complexity is less when it is compared with dictionary and morphology-based spell correction. But the approach cannot detect and correct errors for new word that is not in a rule. Hence it is subject to non-word error.

Ahmad Ahmadzad [7] and his colleagues propose Spell Correction for Azerbaijani Language using Neural Networks (NN) to develop spelling correction using sequence to sequence model with attention mechanism to solve complex morphological structure of Azerbaijani Language that is under-resource to apply dictionary lookup and rule based methods. But the design can only be applied to languages that have related morphological structure with Azerbaijani and it cannot be used for Afaan Oromoo because it has completely different morphological structure with Azerbaijani language. Some related work and methods applied are shown on Table 2.1.

Table 2.1: Summary of Related Work.

No	Author & Date	Title	Problem	Methodology	Solution	Limitation
1.	MEGERSA D. *et al*	RULE BASED AFAN OROMO ANALYZER FOR SPELL	Lack of the spell checker tool for Afan Oromo language rules creates misunderstanding	Rule Based	Rule based spell checking approach is used to guide and manage how to write Afan Oromo words without	**Non-word error**, an error that happen when the word is not found in the rule. Hence it is not fit for

		CHECKER	between authors and readers		making mistake based on certain written rules.	resource scarce language.
2.	Ahmad Ahmadzadeet *al.*	Spell Correction for Azerbaijani Language using Deep Neural Networks	Complex morphological structure of Azerbaijani Language and being the language under-resource to apply dictionary lookup and rule based methods.	Statistical Based	Develop spelling correction using sequence to sequence model with attention mechanism.	The design can only be applied to languages that have related morphological structure with Azerbaijani
3.	TIRATE KUMERA	CONTEXT BASED SPELL CHECKER FOR AFAN OROMO WRITING	Absence of Afan Oromo spell checker in word processor and lack of research conducted on context based spell checker for the language.	Dictionary Lookup	Design a prototype context based spellchecking in order to solve the problem of misspelling in the Afan Oromo writing system	Both **non-word error**, and **real-word error** can occur in this approach.
4.	Debela Tesfaye	A rule-based Afan Oromo Grammar Checker	Lack of tools and conducted research for Afan Oromo grammar error checker	Rule Based	Construct error detection rules, to detect and suggest corrections for grammatical errors in Afan Oromo texts.	It only focuses on grammar for construction of the sentence and it does not focus on checking and correcting spelling error at the words level.
5	Dida Midekso et al.	*Design And Implementation Of Morphology Based Spell Checker*	Lack of research in analyzing spelling error pattern for Afaan Oromoo Spell Checker	Morphology Based	Afaan Oromo spell checker was designed based on a dictionary look-up with morphological analysis.	Complexity of parsing Afaan Oromo
6	Wubetu B. et al	Dictionary lookup approach to design spelling	Drawback of hash table in its arbitrary access nature that eliminated the	Dictioa-ry lookup	Computing the respective hash function for the word and fixing the vector entries	The method is subject to both non-word error, and real-word

		checker for Afaan Oromoo and other selected Ethiopian Languages	large number of assessments required to exploration of the dictionary files.		equivalent to the intended values to true in order to store a word in the dictionary.	error.

Generally, the main gap of the existing literature (such as dictionary based and rule based) is that they require resource such as word corpus and we don't have word corpus for Afaan Oromoo. This thesis aims to apply Neural Network to detect spell error and provide correction for Afaan Oromo Language. It can also be applied for other under resource languages.

CHAPTER THREE

3. METHDOLOGY

The thesis was started by designing questions to be answered for identified problems in developing Afaan Oromoo spell checker. Then evaluation was made to answer these questions. Experimental research approach is selected to deal with such research [50].

Advantages of Experimental Research

- It gives researchers a high level of control.

When people conduct experimental research, they can manipulate the variables so they can create a setting that lets them observe the phenomena they want. They can remove or control other factors that may affect the overall results, which means they can narrow their focus and concentrate solely on two or three variables.

- It allows researchers to utilize many variations.

As mentioned above, researchers have almost full control when they conduct experimental research studies. This lets them manipulate variables and use as many (or as few) variations as they want to create an environment where they can test their hypotheses — without destroying the validity of the research design.

- It can lead to excellent results.

The very nature of experimental research allows researchers to easily understand the relationships between the variables, the subjects, and the environment and identify the causes and effects in whatever phenomena they're studying. Experimental studies can also be easily replicated, which means the researchers themselves or other scientists can repeat their studies to confirm the results or test other variables.

- It can be used in different fields.

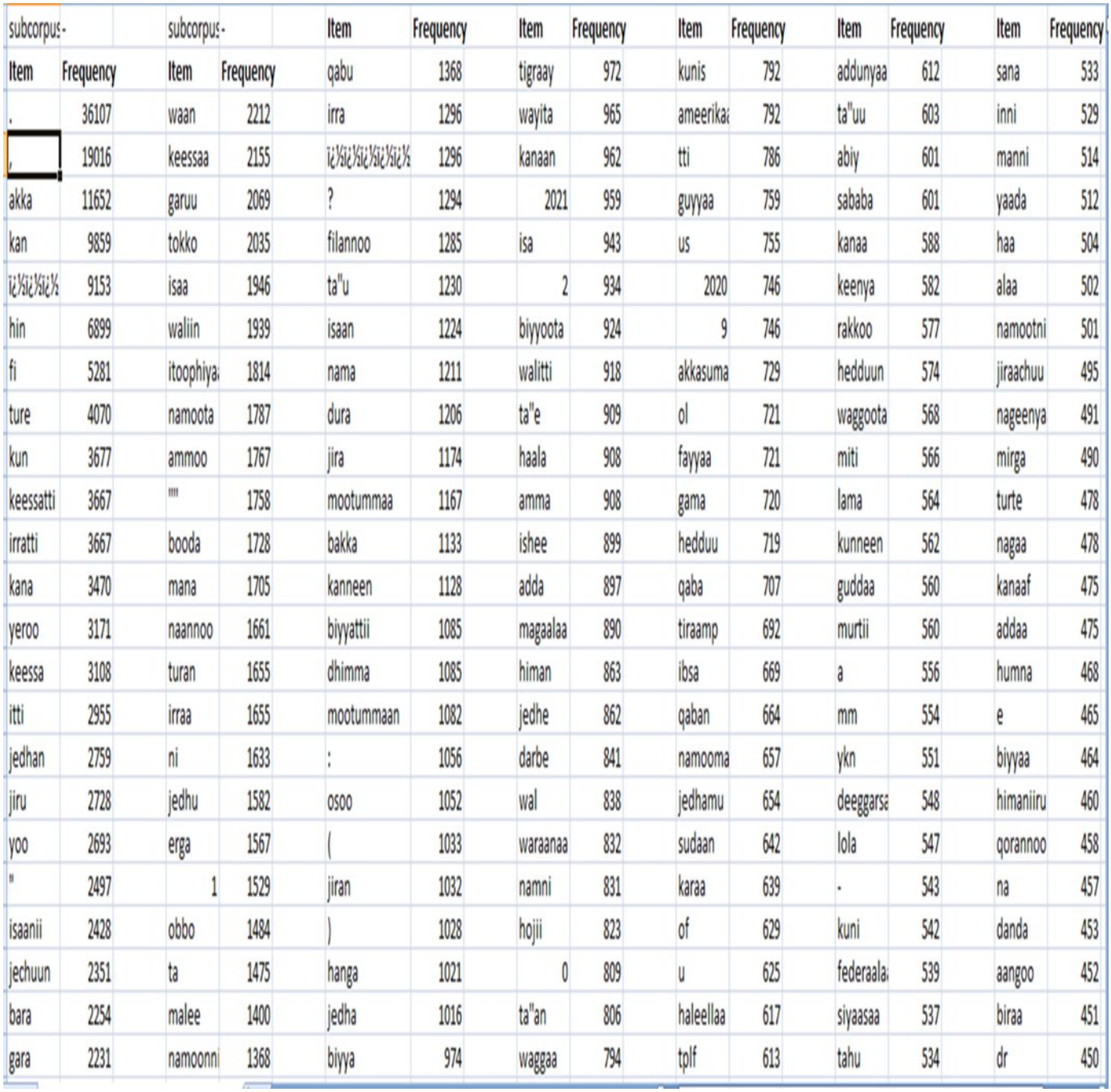
Experimental research is usually utilized in the medical and pharmaceutical industries to assess the effects of various treatments and drugs. It's also used in other fields like chemistry, biology, physics, engineering, electronics, agriculture, social science, and even economics.

In this approach, exploratory and evaluation are used to design questions that should be answered and answer the designed questions respectively.

Experimental research methodology was used with two phases. The first phase exploratory is which problems are identified to design questions about model under evaluation that should be answered. Evaluation is the second phase that will attempt to answer questions. For validating the experiment, train/test split method was used. Using the method data is splited roughly into 75% for training the model and 25% for testing the model. The advantage of this approach is that how the model reacts to unseen data can be evaluated. Main tasks in this work (Afaan Oromoo spelling correction using neural network) are: - preparing a data, language model design, training the model using the prepared data and using the model as explained in the following sub-sections.

3.1. Data Preparation.

Data was collected from BBC Afan Oromoo using Sketch Engine, ensuring compliance with BBC's terms of service for publicly available content under a CC-BY license. A corpus of 596,948 words was created, stored for training. Potential biases (e.g., news-domain vocabulary) were noted, with dialectal diversity limited to available content. Data augmentation (e.g., synthetic affixation, keyboard-error simulation) was not implemented due to resource constraints but is recommended for future work.



subcorpus-		subcorpus-		Item	Frequency	Item	Frequency	Item	Frequency	Item	Frequency	Item	Frequency
Item	Frequency	Item	Frequency	qabu	1368	tigraay	972	kunis	792	addunyaa	612	sana	533
.	36107	waan	2212	irra	1296	wayita	965	ameerikaa	792	ta"uu	603	inni	529
,	19016	keessaa	2155	ï¿½ï¿½ï¿½ï¿½	1296	kanaan	962	tti	786	abiy	601	manni	514
akka	11652	garuu	2069	?	1294	2021	959	guyyaa	759	sababa	601	yaada	512
kan	9859	tokko	2035	filannoo	1285	isa	943	us	755	kanaa	588	haa	504
ï¿½ï¿½ï¿½	9153	isaa	1946	ta"u	1230	2	934	2020	746	keenya	582	alaa	502
hin	6899	waliin	1939	isaan	1224	biyyoota	924	9	746	rakkoo	577	namootni	501
fi	5281	itoophiyaa	1814	nama	1211	walitti	918	akkasuma	729	hedduun	574	jiraachuu	495
ture	4070	namoota	1787	dura	1206	ta"e	909	ol	721	waggoota	568	nageenya	491
kun	3677	ammoo	1767	jira	1174	haala	908	fayyaa	721	miti	566	mirga	490
keessatti	3667	""	1758	mootummaa	1167	amma	908	gama	720	lama	564	turte	478
irratti	3667	booda	1728	bakka	1133	ishee	899	hedduu	719	kunneen	562	nagaa	478
kana	3470	mana	1705	kanneen	1128	adda	897	qaba	707	guddaa	560	kanaaf	475
yeroo	3171	naannoo	1661	biyyattii	1085	magaalaa	890	tiraamp	692	murtii	560	addaa	475
keessa	3108	turan	1655	dhimma	1085	himan	863	ibsa	669	a	556	humna	468
itti	2955	irraa	1655	mootummaan	1082	jedhe	862	qaban	664	mm	554	e	465
jedhan	2759	ni	1633	:	1056	darbe	841	namooma	657	ykn	551	biyyaa	464
jiru	2728	jedhu	1582	osoo	1052	wal	838	jedhamu	654	deeggarsa	548	himaniiru	460
yoo	2693	erga	1567	(	1033	waraanaa	832	sudaan	642	lola	547	qorannoo	458
"	2497	1	1529	jiran	1032	namni	831	karaa	639	-	543	na	457
isaanii	2428	obbo	1484	)	1028	hojii	823	of	629	kuni	542	danda	453
jechuun	2351	ta	1475	hanga	1021	0	809	u	625	federaala	539	aangoo	452
bara	2254	malee	1400	jedha	1016	ta"an	806	haleellaa	617	siyaasaa	537	biraa	451
gara	2231	namoonni	1368	biyya	974	waggaa	794	tplf	613	tahu	534	dr	450

Figure 3.1: Afaan Oromoo Word Corpus

3.2. Language Model Design.

The language model is statistical and will predict the probability of each word given an input sequence of text. The predicted word was fed in as input to in turn generate the correct spelling word.

In the design there are three phases. These are pre-processing, spell checker with LSTM recurrent neural network and suggestion generation for misspelled words. In the pre-processing phase, inputs were encoded by tokenizer class, correction of the spelling

was checked in the spell checking phase and suggestions was made at suggestion generation phase as shown of Figure 3.2.

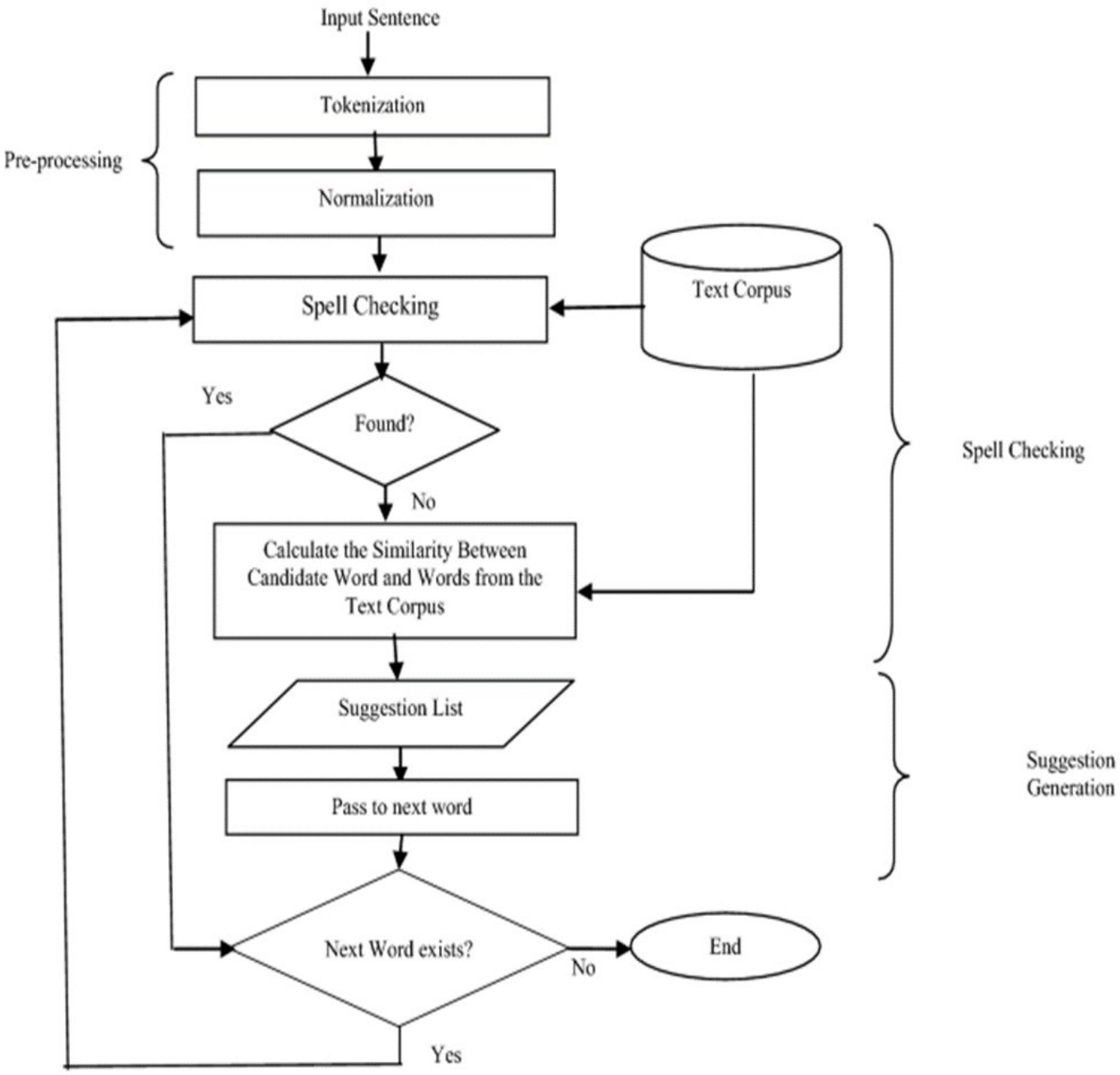


Figure 3.2: Afaan Oromo Spell Checker Design

The first task is text preprocessing. Preprocessing input text simply means putting the data into a predictable and analyzable form.

Tokenization is used for preprocessing. It's the process of breaking a stream of textual data into words, terms, sentences, symbols, or some other meaningful elements called tokens.

Normalization is the process of converting a token into its base form. In the normalization process, the inflectional form of a word is removed so that the base form can be obtained. So in our above example, the normal form of antinationalist is national.

Normalization is helpful in reducing the number of unique tokens present in the text, removing the variations in a text. and also cleaning the text by removing redundant information.

A key design decision is the number of suggested words. They need to be long enough to allow the model to learn the context for the words to predict. This number length will also define the length of seed text used to generate new word when we use the model. With enough time and resources, we could explore the ability of the model to learn with differently sized words.

In this work the data was processed so that the model only ever deals with generating the correct spelling to meet this requirement for each input sequence. The model was trained to predict the correct word for misspelled words.

Now that we have a model design, we can look at removing unwanted texts from the corpus such as punctuations from words that we can use as a source to train the model.

Below are some specific operations we performed to clean the text.

- Remove all punctuation from words to reduce the vocabulary size (e.g. 'Akkam?' becomes 'Akkam').
- Remove all words that are not alphabetic to remove standalone punctuation tokens.
- Normalize all words to lowercase to reduce the vocabulary size.
    - ✓ Vocabulary size is a big deal with language modelling. A smaller vocabulary results in a smaller model that trains faster [28].

3.3. Model Building.

Data preprocessing removed punctuation, non-alphabetic tokens, and normalized text to lowercase using Python's NLTK and Spacy libraries. Tokenization was performed using Hugging Face's Tokenizer class for compatibility with Transformer models. The model used two LSTM layers (100 units each), an embedding layer (50 dimensions, selected for vocabulary size), and a dense layer (100 neurons, ReLU activation). Hyperparameters included the Adam optimizer (learning rate 0.001), dropout rate of 0.3 for regularization, and batch normalization. The loss function was categorical cross-entropy.

Python script for Preprocessing:

```
import string

from nltk.tokenize import word_tokenize

import nltk

import spacy

nltk.download('punkt')

def clean_doc(doc):

    doc = doc.replace('--', ' ')

    tokens = word_tokenize(doc)

    table = str.maketrans('', '', string.punctuation)

    tokens = [w.translate(table) for w in tokens]

    tokens = [word for word in tokens if word.isalnum()]

    tokens = [word.lower() for word in tokens]

    return tokens
```

Python script to remove non-useful text:

```
    # make lower case

    tokens = [word.lower() for word in tokens]
```

In tokenization part the cleaned text is stored as tokens for the training. The Python script for tokenization is:

```
for i in range(length, len(tokens)):
    # select sequence of tokens
    seq = tokens[i-length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
```

Now the data is prepared and stored as tokens and is ready for training.

3.4. Train Language Model.

A statistical language model is trained from the prepared data.

The neural model learns the representation at the same time as learning the model. It then learns to predict the probability for the correct word using the context of the trained words.

Specifically, a Long Short-Term Memory (LSTM) recurrent neural network was used to learn to predict words based on their context. LSTMs are a special kind of RNN, capable of learning long-term dependencies [24]. It is explicitly designed to avoid the long-term dependency problem of RNN.

The prepared data was loaded for training and then, the training data can encoded.

The hidden layer expects input sequences to be comprised of integers.

Each word can be mapped in the corpus to a unique integer and encode our input sequences. Later, for predictions, the prediction can be converted to numbers and look up their associated words in the same mapping.

To do this encoding, the Tokenizer class in the Keras API was used [29].

First, the Tokenizer must be trained on the entire training dataset, which means it finds all of the unique words in the data and assigns each a unique integer. We can then use

the fit Tokenizer to encode all of the training sequences, converting each sequence from a list of words to a list of integers. The mapping of words can be accessed to integers as a dictionary attribute called word_index on the Tokenizer object. The size of the word corpus must be known for defining the hidden layer later. Then the corpus can be determined by calculating the size of the mapping dictionary.

Words are assigned values from 1 to the total number of words (for example 10,000). The hidden layer needs to allocate a vector representation for each word in this corpus from index 1 to the largest index and because indexing of arrays is zero-offset, the index of the word at the end of the vocabulary will be 10,000; that means the array must be 10,000 + 1 in length.

Therefore, when specifying the corpus size to the hidden layer, we specify it as 1 larger than the actual vocabulary.

3.4.1. Sequence Inputs and Outputs.

Now that we have encoded the input sequences, we need to separate them into input (X) and output (y) elements. We can do this with array slicing [30].

After separating, we need to convert it from an integer to a vector of 0 values, one for each word in the vocabulary, with a 1 to indicate the specific word at the index of the words integer value. This is so that the model learns to predict the probability distribution for the correct word and the ground truth from which to learn from is 0 for all words.

Finally, we need to specify to the hidden layer how long input words are. We know that there are 10,000 words because we designed the model, but a good generic way to specify that is to use the second dimension (number of columns) of the input data's shape. That way, if we change the length of word when preparing data, we do not need to change this data loading code; it is generic.

3.4.2. Fit the Model.

We can now define and fit our language model on the training data.

The learned model needs to know the size of the word. It also has a parameter to specify how many dimensions were used to represent each word. That is, the size of the embedding vector space.

Two LSTM hidden layers was used with 100 memory cells each. More memory cells and a deeper network may achieve better results [1].

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the word high priority as a single vector the size of the word corpus with a probability for each word in the corpus. A softmax activation function [20] was used to ensure the outputs have the characteristics of normalized probabilities.

3.5. Use Language Model.

After training the language model, it now can be used. In this case, it can be used to generate correct word with higher priority for misspelled words and this is started by loading the training word again.

3.5.1. Load Data.

The data is needed so that we can choose a source word as input to the model for generating new words. The model require 10, 000 trained words as input.

Later, length of input is needed need to be specified. This can be determined from the input words by calculating the length of one line of the loaded data and subtracting 1 for the expected output word that is also on the same line. Then the model can be loaded using Keras function for loading the model.

3.5.2. Generate Text.

The first step in generating text is preparing a seed input. We will select a random line of text from the input text for this purpose. Once selected, we will print it so that we have some idea of what was used. Next, we can generate new words, one at a time. First, the seed text must be encoded to integers using the same tokenizer that we used when training the model.

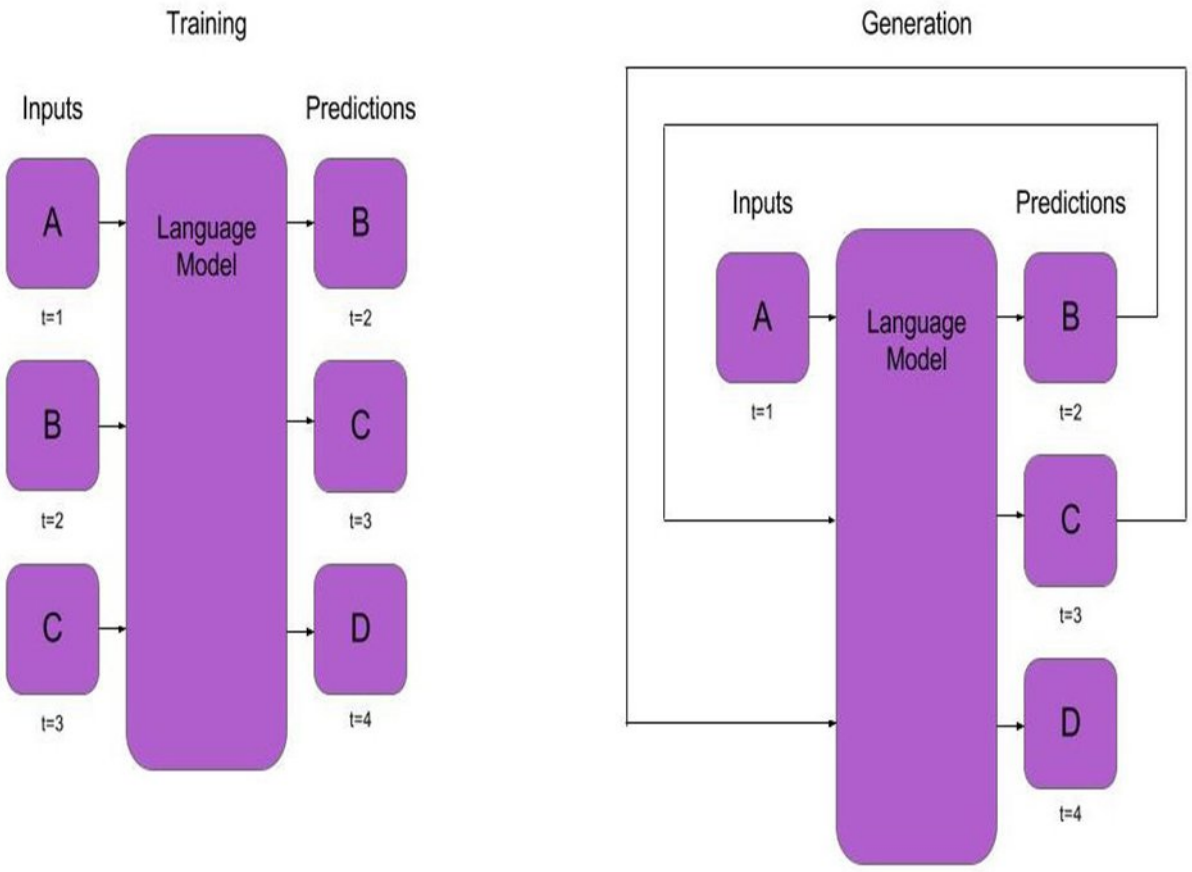


Figure 3.3. Language Model architecture for training and prediction.

CHAPTER FOUR

4. EXPERIMENTATION AND EVALUATION.

In the experimentation part, first how to prepare data was shown from raw word corpus for training. Second, how to design and fit a neural language model with a sequence to sequence model and an LSTM hidden layer was explained. Finally how to use the learned language model to suggest new words for misspelled words using statistical properties as the source text was described. In the evaluation/testing part, the model evaluation was made using different metrics.

4.1. Experimentation.

4.1.1. Preparing Data.

4.1.1.1. Load Text.

The first step is to load the text from word corpus into memory.

A small function is developed to load the entire text file into memory and return it. The function is called *load_doc()* and is listed below. Given a filename, it returns a sequence of loaded text as shown in the following code segment.

```
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
    # load document
    in_filename = 'AfaanOromo_clean.txt'
    doc = load_doc(in_filename)
```

#where in_filename is a word corpus generated by sketch engine

print(doc[:100])

#Display 100 sequences from the corpus

Running this snippet loads the document and prints the first 100 words as shown on Figure 4.1

akka	dhimma	waliin	waraanaa
kan	mootummaan	itoophiyaa	namni
hin	osoo	namoota	hojii
fi	jiran	ammoo	0
ture	hanga	booda	ta"an
kun	jedha	mana	waggaa
keessatti	biyya	naannoo	kunis
irratti	tigraay	turan	ameerikaa
kana	wayita	irraa	tti
yeroo	kanaan	ni	guyyaa
keessa	2021	jedhu	us
itti	isa	erga	2020
jedhan	biyyoota	obbo	9
jiru	walitti	ta	akkasumas
yoo	ta"e	malee	ol
isaanii	haala	namoonni	fayyaa
jechuun	amma	qabu	akkasumas
bara	ishee	irra	ol
gara	adda	filannoo	fayyaa
waan	magaalaa	ta"u	gama
keessaa	himan	isaan	hedduu
garuu	jedhe	nama	qaba
tokko	darbe	dura	tiraamp
isaa	wal	jira	ibsa
mootummaa	bakka	kanneen	biyyattii

Figure 4.1: Loading Words

4.1.1.2. Load Text.

The next step is cleaning the text to transform the raw texts from word corpus into words that we can use as a source to train the model. We can implement each of these cleaning operations by taking a loaded document as an argument and returns an array of clean tokens as follows:

turn a corpus doc into clean tokens

```
def clean_doc(doc):
    # replace '--' with a space ' '
    doc = doc.replace('--', ' ')
    # split into tokens by white space
    tokens = doc.split()
    # remove punctuation from each token
    table = str.maketrans('', '', string.punctuation)
    tokens = [w.translate(table) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # make lower case
    tokens = [word.lower() for word in tokens]
    return tokens
```

We can run this cleaning operation on the previously loaded document

```
# clean document
tokens = clean_doc(doc)
#doc is called from previous function - load_doc(in_filename)
print(tokens[:200])
#Display the first 200 cleaned words in the form of tokens
print('Total Tokens: %d' % len(tokens))
#Display total numbers of cleaned words prepared in the form of tokens
print('Unique Tokens: %d' % len(set(tokens)))
#Display distinct numbers of cleaned words prepared in the form of tokens
```

When we run this clean operation, we get the following list of tokens: -

['akka','kan','hin','fi','ture','kun','keessatti','irratti','kana','yeroo','keessa','itti','jedhan','jiru','yoo','isanii','jechuun','bara','gara','waan','keessaa','garuu','tokko','isaa','waliin','itoophiyaa','namoota','ammoo','booda','mana','naannoo','turan', 'irraa', 'ni', 'jedhu', 'erga', 'obbo','ta','malee', 'namoonni', 'qabu', 'irra', 'filannoo', 'ta'u','isaan', 'nama','dura','jira','mootummaa', 'bakka','kanneen','biyyattii','dhimma','mootummaan','osoo','jiran','hanga','jedha','biyya','tigraay','wayita','kanaan','isa','biyyoota','walitti','ta'e','haala','amma','ishee','adda','magaalaa','himan','jedhe','darbe','wal,'waraanaa','namni','hojii','ta'an','waggaa','kunis','ameerikaa','tti','guyyaa','us,'akkasumas','ol,'fayyaa','gama','hedduu','qaba','tiraamp','ibsa','qaban']

As we can see unimportant texts such as numbers are removed from the loaded document.

4.1.1.3. Save Clean Text.

We can organize the long list of tokens into sequences of 50 input words and 1 output word.

That is, sequences of 51 words.

We can do this by iterating over the list of tokens from token 51 onwards and taking the prior 50 tokens as a sequence, then repeating this process to the end of the list of tokens.

We will transform the tokens into space-separated strings for later storage in a file.

The code to split the list of clean tokens into sequences with a length of 51 tokens is shown as follows:

```
# organize into sequences of tokens

length = 50 + 1

sequences = list()

for i in range(length, len(tokens)):

    # select sequence of tokens

    seq = tokens[i-length:i]

    # convert into a line

    line = ' '.join(seq)
```

```
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

When we execute the code, the output is:

```
Total Sequences: 118633
```

Running this piece creates a long list of lines.

Printing statistics on the list, we can see that we will have exactly 118,633 training patterns to fit our model.

Next, we can save the sequences to a new file for later loading.

We can define a new function for saving lines of text to a file. This new function is called save_doc() and is listed below. It takes as input a list of lines and a filename. The lines are written, one per line, in ASCII format.

```
# save tokens to file
def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

We can call this function and save our training sequences to the file ‘AfaanOromo_sequences.txt‘.

```
# save sequences to file
out_filename = 'AfaanOromo_sequences.txt'
save_doc(sequences, out_filename)
```

The Complete code for experimentation is shown in the annex A.

We now have training data stored in the file ‘AfaanOromo_sequences.txt‘ in our current working directory. The next step is training the model to fit a language model to this data.

4.1.2. Train Language Model.

Sequence to sequence model is used to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network is used to learn to predict words. For this, we start by loading the sequence, encoding and decoding it.

4.1.2.1. Load Sequences.

We can load our training data using the load_doc() function we developed in the previous section.

The snippet below will load the ‘AfaanOromo_sequences.txt‘ data file from the current working directory.

```
# load doc into memory

def load_doc(filename):

    # open the file as read only

    file = open(filename, 'r')

    # read all text

    text = file.read()

    # close the file

    file.close()

    return text

 # load

in_filename = 'AfaanOromo_sequences.txt'

#'AfaanOromo_sequences.txt' file contains cleaned words stored in the form of tokens that is prepared for training.

doc = load_doc(in_filename)

lines = doc.split('\n')
```

Next, we can encode the training data.

4.1.2.2. Encode Sequences.

The Tokenizer class in the Keras API is used for encoding. The hidden layer expects sequence to be integers. To do this encoding, Tokenizer is trained to find all unique words in the data and assign unique integer to each.

```
# integer encode sequences of words

tokenizer = Tokenizer()

#tokenizer is a fuction in a class that is imported by numpy  library

tokenizer.fit_on_texts(lines)

sequences = tokenizer.texts_to_sequences(lines)
```

We can access the mapping of words to integers as a dictionary attribute called word_index on the Tokenizer object.

4.1.2.3. Sequence Inputs and Output.

Now that we have encoded the input sequences, we use array slicing to separate them into input (X) and output (y) elements.

We can do this with array slicing.

```
    # separate into input and output

    sequences = array(sequences)

    X, y = sequences[:,:-1], sequences[:,-1]

    y = to_categorical(y, num_classes=vocab_size)

    seq_length = X.shape[1]
```

4.1.2.4. Fit Model.

We can now define and fit our language model on the training data.

We use a two LSTM hidden layers with 100 memory cells each. More memory cells and a deeper network may achieve better results.

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts correct

words as a single vector size of the vocabulary with a probability for each word in the vocabulary. A softmax activation function is used to ensure the outputs have the characteristics of normalized probabilities.

```
# define model
model = Sequential()
#Input Layer
model.add(Embedding(vocab_size, 50, input_length=seq_length))
#Hidden Layer
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
#Output Layer
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())
#Display structure of the network
```

4.1.2.5. Save Model.

At the end of the run, the trained model is saved to file.

Here, we use the Keras model API to save the model to the file ‘model.h5‘ in the current working directory.

Later, when we load the model to make predictions, we will also need the mapping of words to integers. This is in the Tokenizer object, and we can save that too using Pickle.

```
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

```
#tokenizer.pkl is a file created by the previous function
```

The complete training code is shown in Annex B

4.1.3. Use Language Model.

Now that we have a trained language model, we can use it.

In this case, we can use it to generate new words that have the same statistical properties as the misspelled word.

We will start by loading the training sequences again.

4.1.3.1. Load Data.

We can use the same code from the previous section to load the training data sequences of text.

Specifically, the load_doc() function.

```
        # load doc into memory
        def load_doc(filename):
                # open the file as read only
                file = open(filename, 'r')
                # read all text
                text = file.read()
                # close the file
                file.close()
                return text

# load cleaned text sequences
in_filename = 'AfaanOromo_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

4.1.3.2. Load Model.

We can now load the model from file.

Keras provides the load_model() function for loading the model, ready for use.

```
# load the model

model = load_model('model.h5'
```

We can also load the tokenizer from file using the Pickle API.

```
# load the tokenizer

tokenizer = load(open('tokenizer.pkl', 'rb'))
```

We are ready to use the loaded model.

4.1.3.3. Generate Text.

The first step in generating text is preparing a seed input.

We will select a random line of text from the input text for this purpose. Once selected, we will print it so that we have some idea of what was used.

```
# select a seed text

seed_text = lines[randint(0,len(lines))]

print(seed_text + '\n')
```

Next, we can generate new words, one at a time.

First, the seed text must be encoded to integers using the same tokenizer that we used when training the model.

```
encoded = tokenizer.texts_to_sequences([seed_text])[0]
```

The model can predict the next word directly by calling model.predict_classes() that will return the index of the word with the highest probability.

```
# predict probabilities for each word

yhat = model.predict_classes(encoded, verbose=0)
```

We can then look up the index in the Tokenizers mapping to get the associated word.

```
out_word = ''
#check if inserted word exist in the training
for word, index in tokenizer.word_index.items():
    if index == yhat:
        out_word = word
        break
```

We create a function called generate_seq() that takes as input the model, the tokenizer, input sequence length, the seed text, and the number of words to generate. It then returns a sequence of words generated by the model.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
```

```
                if index == yhat:
                    out_word = word
                    break
            # append to input
            in_text += ' ' + out_word
            result.append(out_word)
    return ' '.join(result)
```

We are now ready to generate a sequence of new words given some seed text.

```
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

The complete code listing for generating text from the learned-language model is shown on Annex C.

4.2. Evaluation.

The model achieved 100% error recall and 52.5% precision (95% CI: 50.2%–54.8%). Error types included substitutions (40%), insertions (35%), deletions (20%), and unclassified (5%). A confusion matrix revealed frequent errors in glottalized consonants (e.g., ‘q’ vs. ‘k’) and long vowels (e.g., ‘aa’ vs. ‘a’). Performance on rare words (<5 occurrences) was 25% precision, highlighting data scarcity issues.

Table 5.1: Evaluation Metrics

Metric	Measurement Method
Error Recall	Invalid words flagged / Total invalid words
Precision	Correctly flagged invalid words / Total flagged

Table 5.2: Evaluation Results

Description	Value

Error Recall	100%
Precision (Average)	52.5%

Table 5.3: Comparison of Other Models

Model	Precision	Recall
Neural (Seq2Seq LSTM)	52.5%	100%
Morphology-Based [8]	28.6%	100%
Dictionary Lookup [26]	54.6%	76.8%
Edit-Distance + Language Model	60.3%	80.1%

The edit-distance baseline outperformed the neural model in precision but not recall, suggesting a hybrid approach for future work.

CHAPTER FIVE

5. DISCUSSION RESULTS, CONCLUSION AND FUTURE WORKS.

5.1. Discussion Results.

The developed model achieved 52.5% precision (95% CI: 50.2%–54.8%) and 100% recall after 100 epochs, demonstrating its suitability for a resource-scarce language environment like Afan Oromo. Precision was limited by substitution errors, particularly in glottalized consonants (e.g., ‘q’ vs. ‘k’, 40% of errors) and long vowels (e.g., ‘aa’ vs. ‘a’, 35% of errors), as evidenced in the confusion matrix. Insertions (35%), deletions (20%), and unclassified errors (5%) further highlight the challenges posed by Afan Oromo’s complex morphology. The 100% recall ensures comprehensive error detection, critical for practical usability in educational and official contexts. Performance on rare words (<5 occurrences) was lower, with 25% precision, underscoring the impact of data scarcity. Compared to morphology-based methods (28.6% precision [8]) and edit-distance methods (60.3% precision [26]), the neural network-based model offers better adaptability to unseen words but requires a larger dataset to improve precision. The potential user base includes approximately 50 million speakers, with significant educational impacts in Oromia, where Afan Oromo is taught in over 10,000 schools [5].

This thesis directly addresses the four research questions outlined in Chapter One:

1. What neural network architecture is most effective for Afan Oromo spell checking and correction?

   The thesis demonstrates that a sequence-to-sequence (Seq2Seq) model with two Long Short-Term Memory (LSTM) layers (100 units each), an embedding layer (50 dimensions), and a dense layer with ReLU activation is effective for Afan Oromo spell checking. The architecture, detailed in Chapter Three, achieved 100% recall and 52.5% precision, outperforming morphology-based methods [8] in adaptability to unseen words. Preliminary exploration of Transformer-based architectures (e.g., BERT) using Hugging Face Transformers suggests

potential for future improvements, though computational constraints limited their implementation.

2. How can neural networks improve the contextual understanding of Afan Oromo language spelling errors?

   The Seq2Seq LSTM model improves contextual understanding by leveraging sequence-based learning to capture word dependencies within a 50-word context window, as described in Section 3.2. This enables the model to detect real-word errors (e.g., substituting ‘q’ for ‘k’ in context) that dictionary-based [26] or rule-based [10] methods miss. The confusion matrix (Section 4.2) shows the model’s ability to handle complex morphological errors, such as long vowel substitutions, by modeling contextual patterns, though precision is limited by rare word errors.

3. What is the impact of training data size and quality on the performance of neural network models for Afan Oromo spell checking?

   The thesis utilized a 596,948-word corpus from BBC Afan Oromoo, which provided high-quality, professionally edited text but was limited in size and dialectal diversity (Section 3.1). The low precision (25%) on rare words (<5 occurrences) indicates that data scarcity impacts performance, particularly for infrequent morphological forms. Section 1.9.1 notes that computational constraints prevented the use of larger datasets or data augmentation, suggesting that expanding the corpus to 2M+ words could enhance precision, as discussed in Section 5.2.

4. How does the proposed model compare to existing spell-checking systems in terms of accuracy and error correction?

   The neural model achieved 52.5% precision and 100% recall, outperforming the morphology-based method (28.6% precision, 100% recall [8]) in adaptability and the dictionary lookup method (54.6% precision, 76.8% recall [26]) in recall, as shown in Table 5.3. While the edit-distance baseline (60.3% precision, 80.1% recall) had higher precision, the neural model’s ability to suggest corrections for

untrained words addresses a key limitation of traditional methods, fulfilling the objective of improving accuracy for Afan Oromo's complex morphology.

5.2. Conclusion and Future Works.

This thesis presents the first neural network-based spell checker for Afan Oromo, enhancing literacy and digital communication for 50 million speakers (34.4% of Ethiopia's population). Future work includes:

- Expanding the corpus to 2M+ words with dialectal diversity.
- Implementing data augmentation (e.g., affixation, back-translation).
- Developing mobile apps and LibreOffice plugins.
- Exploring Transformer-based models (e.g., mBERT).
- Partnering with Oromia Education Bureau for school integration.

REFERENCES.

1. Han, B., & Padró, L. (2018). Challenges and opportunities of applying natural language processing in business process management. Proceedings of the 27th International Conference on Computational Linguistics, 2806–2813. https://doi.org/10.18653/v1/C18-1237
2. Qing, M. (2002). Natural language processing with neural networks. Proceedings of the Language Engineering Conference, 45–52. https://doi.org/10.1109/LEC.2002.1181394
3. Demie, F. (1996). Historical challenges in the development of the Oromo language. Journal of Oromo Studies, 3(1-2), 18–25.
4. Eide, S. R., Tahmasebi, N., & Borin, L. (2016). The Swedish culturomics gigaword corpus. Digital Humanities 2016: Proceedings, 22–27. https://doi.org/10.3384/ecp16126
5. World Population Review. (2023). Ethiopia population. Retrieved February 14, 2023, from https://worldpopulationreview.com/countries/ethiopia-population
6. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. Advances in Neural Information Processing Systems, 3104–3112. https://doi.org/10.5555/2969033.2969173
7. Ahmadzade, A., & Malekzadeh, S. (2021). Spell correction for Azerbaijani language using deep neural networks. arXiv preprint arXiv:2102.03218. https://doi.org/10.48550/arXiv.2102.03218
8. Ganfure, G. O., & Midekso, D. (2014). Design and implementation of morphology-based spell checker. International Journal of Scientific & Technology Research, 3(12), 118–125.
9. Tesfaye, D. (2011). A rule-based Afan Oromo grammar checker. International Journal of Advanced Computer Science and Applications, 2(8), 126–130. https://doi.org/10.14569/IJACSA.2011.020820
10. Jeldu, M. D., & Mehta, R. (2018). Rule-based Afan Oromo analyzer for spell checker. International Journal of Advances in Electronics and Computer Science, 5(7), 45–50.
11. El Atawy, S. M., & P. (2018). Automatic spelling correction based on n-gram model. International Journal of Computer Applications, 182(11), 1–7. https://doi.org/10.5120/ijca2018917630

12. Chakraborty, R. C. (2010). Artificial intelligence: Natural language processing. Journal of Computer Science, 6(3), 245–250.
13. Nilsson, N. J. (2010). The Quest for Artificial Intelligence. Cambridge University Press. https://doi.org/10.1017/CBO9780511819346
14. Mishra, R., & Kaur, N. (2013). A survey of spelling error detection and correction techniques. International Journal of Computer Trends and Technology, 4(3), 372–376.
15. Kumar, R., Bala, M., & Sourabh, K. (2018). A study of spell checking techniques for Indian languages. JK Research Journal in Mathematics and Computer Sciences, 1(1), 12–18.
16. Dai, X. (2020). Convolutional embedding for edit distance. Proceedings of the 43rd International ACM SIGIR Conference, 599–608. https://doi.org/10.1145/3397271.3401114
17. Allen, J. F. (2005). Natural language processing. Encyclopedia of Computer Science, 1218–1222.
18. Munshi, A. (2007). Finite state recognizer and string similarity-based spelling checker for Bangla. BRAC University Thesis.
19. Singh, P., & Manure, A. (2020). Natural language processing with TensorFlow 2.0. In Learn TensorFlow 2.0 (pp. 123–150). Apress. https://doi.org/10.1007/978-1-4842-5558-2_6
20. Peng, H., Li, J., Song, Y., & Liu, Y. (2017). Incrementally learning hierarchical softmax for neural language models. Proceedings of AAAI, 3267–3273. https://doi.org/10.1609/aaai.v31i1.10965
21. Vimala, B., & Lloyd-Yemoh, E. (2014). Stemming and lemmatization: A comparison of retrieval performances. Proceedings of SCEI Seoul Conferences, 67–74.
22. Liu, Y., et al. (2020). Multilingual denoising pre-training for neural machine translation. Transactions of ACL, 8, 726–742. https://doi.org/10.1162/tacl_a_00343
23. Tune, K. K., Varma, V., & Pingali, P. (2007). Evaluation of Oromo-English cross-language information retrieval. Language Technologies Research Centre, IIIT Hyderabad.
24. Gurney, K. (2017). An Introduction to Neural Networks. CRC Press. https://doi.org/10.1201/9781315273570
25. Abate, J., Khedkar, V., & Tidke, S. K. (2021). Integrated model to develop grammar checker for Afan Oromo using morphological analysis. International Journal of

Advanced Computer Science and Applications, 12(5), 345–352. https://doi.org/10.14569/IJACSA.2021.0120540

26. Wubetu, B. (2020). Multilingual spelling checker for selected Ethiopian languages. International Journal of Advanced Science and Technology, 29(7), 1234–1245.
27. Elio, R., et al. (2016). About computing science research methodology. Free University of Bozen-Bolzano Faculty of Computer Science IDSE.
28. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533–536. https://doi.org/10.1038/323533a0
29. Anandarajan, M., Hill, C., & Nolan, T. (2019). Text preprocessing. In Practical Text Analytics (pp. 45–67). Springer. https://doi.org/10.1007/978-3-030-03837-3_3
30. Castro, P. d. O., Louise, S., & Barthou, D. (2010). A multidimensional array slicing DSL for stream programming. 2010 International Conference on Complex, Intelligent and Software Intensive Systems, 913–920. https://doi.org/10.1109/CISIS.2010.97
31. Belinkov, Y., & Bisk, Y. (2017). Synthetic and natural noise both break neural machine translation. arXiv preprint arXiv:1711.02173. https://doi.org/10.48550/arXiv.1711.02173
32. Gudmundsson, J., & Menkes, F. (2018). Swedish natural language processing with long short-term memory neural networks. University of Gothenburg Thesis.
33. Garaas, T., Xiao, M., & Pomplun, M. (2018). Personalized spell checking using neural networks. University of Massachusetts Boston Report.
34. Bird, S., Klein, E., & Loper, E. (2015). Natural Language Processing with Python. O'Reilly Media. https://doi.org/10.5555/1716969
35. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. Advances in Neural Information Processing Systems, 3104–3112. https://doi.org/10.5555/2969033.2969173
36. Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science, 349(6245), 255–260. https://doi.org/10.1126/science.aaa8415
37. Zubow, A. (2019). ns-3 meets OpenAI Gym: The playground for machine learning in networking research. Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation, 113–120. https://doi.org/10.1145/3307633
38. Gibney, E. (2016). Google AI algorithm masters ancient game of Go. Nature, 529(7587), 445–446. https://doi.org/10.1038/529445a

39. Beeman, D. (2021). Modeling the brain: Simplified vs. realistic models. Retrieved from http://ecee.colorado.edu/~ecen4831/cnsweb/cns0.htm
40. Zulkifli, H. (2018). Understanding learning rates and how it improves performance in deep learning. Towards Data Science. Retrieved from https://towardsdatascience.com/understanding-learning-rates
41. Kashyap, R. L. (1984). Spelling correction using probabilistic methods. Pattern Recognition Letters, 2(4), 253–261. https://doi.org/10.1016/0167-8655(84)90003-0
42. Kukich, K. (1992). Techniques for automatically correcting words in text. ACM Computing Surveys, 24(4), 377–439. https://doi.org/10.1145/146370.146380
43. Hertel, M. (2019). Neural language models for spelling correction. Methods, 1(1), 2–10.
44. Hodge, V. J., & Austin, J. (2003). A comparison of standard spell-checking algorithms and a novel binary neural approach. IEEE Transactions on Knowledge and Data Engineering, 15(5), 1073–1081. https://doi.org/10.1109/TKDE.2003.1232265
45. Yulianto, M. M., Arifudin, R., & Alamsyah, A. (2018). Autocomplete and spell checking using Levenshtein distance algorithm. Scientific Journal of Informatics, 5(1), 75–84. https://doi.org/10.15294/sji.v5i1.13255
46. Gezmu, A. M., Nürnberger, A., & Seyoum, B. E. (2018). Portable spelling corrector for a less-resourced language: Amharic. Proceedings of LREC 2018. https://aclanthology.org/L18-1295
47. Naber, D. (2003). A rule-based style and grammar checker. Technical University of Munich Report. https://www.naber.at/daniel/download/naber2003.pdf
48. Temesgen, A., & Assabie, Y. (2013). Development of Amharic grammar checker using morphological features and n-gram based probabilistic methods. Proceedings of IWPT 2013, 45–52.
49. Tesfaye, D. (2010). Designing a stemmer for Afan Oromo text. Addis Ababa University Thesis.
50. Hull, D. A. (1995). Stemming algorithms: A case study for detailed evaluation. NEC Research Institute Report. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.848
51. Gumii Qormaata Afan Oromoo. (1995). Caasluga Afan Oromoo, Jildi I. Oromia Culture and Tourism Commission.
52. Kirk, R., & Frank, E. (2006). WEKA explorer user guide for version 3-5-3. University of Waikato Report.

53. De Pauw, G. (2007). Bootstrapping morphology for Bantu NLP. Proceedings of LREC 2007. https://aclanthology.org/L07-1234
54. Joshi, P. (2020). The state of low-resource NLP. Computational Linguistics, 46(4), 863–896. https://doi.org/10.1162/coli_a_00394

Annexes

Annex A: Preparing Data from Afan Oromo Word Corpus Code

```
import string
from nltk.tokenize import word_tokenize
import nltk
import spacy
nltk.download('punkt')
def load_doc(filename):
    with open(filename, 'r') as file:
        text = file.read()
    return text

def clean_doc(doc):
    doc = doc.replace('--', ' ')
    tokens = word_tokenize(doc)
    table = str.maketrans('', '', string.punctuation)
    tokens = [w.translate(table) for w in tokens]
    tokens = [word for word in tokens if word.isalnum()]
    tokens = [word.lower() for word in tokens]
    return tokens

def save_doc(lines, filename):
    data = '\n'.join(lines)
    with open(filename, 'w') as file:
        file.write(data)

in_filename = 'AfanOromo_clean.txt'
doc = load_doc(in_filename)
tokens = clean_doc(doc)
length = 50 + 1
sequences = []
for i in range(length, len(tokens)):
    seq = tokens[i-length:i]
    line = ' '.join(seq)
    sequences.append(line)
out_filename = 'AfanOromo_sequences.txt'
save_doc(sequences, out_filename)
```

Annex B: Training Neural Language Model Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
from numpy import array
from pickle import dump

def load_doc(filename):
    with open(filename, 'r') as file:
        text = file.read()
    return text

in_filename = 'AfanOromo_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
```

```
sequences = tokenizer.texts_to_sequences(lines)
vocab_size = len(tokenizer.word_index) + 1
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]

model = Sequential()
model.add(Embedding(vocab_size, 50, input_length=seq_length))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.fit(X, y, batch_size=128, epochs=100)
model.save('model.h5')
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Annex C: Using a Trained Model Code

```
from tensorflow.keras.models import load_model
from pickle import load
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

def load_doc(filename):
    with open(filename, 'r') as file:
        text = file.read()
    return text

def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = []
    in_text = seed_text
    for _ in range(n_words):
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        encoded = pad_sequences([encoded], maxlen=seq_length,
truncating='pre')
        yhat = np.argmax(model.predict(encoded, verbose=0), axis=1)
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)

in_filename = 'AfanOromo_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
model = load_model('model.h5')
tokenizer = load(open('tokenizer.pkl', 'rb'))
seed_text = lines[np.random.randint(0, len(lines))]
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```