Ministry of Science and Higher Education of the Republic of Kazakhstan

Suleyman Demirel University

Farukh Iskalinov

Implementing automatic word prediction and autocorrection for the Kazakh-language keyboard on the iOS platform

THESIS

Presented in Partial Fulfilment for the

Master of Technical Sciences Degree in Computer Science

(degree code: 7M06102)

Department of Computer Science

Faculty of Engineering and Natural Sciences

Supervisor: **PhD Birzhan Moldagaliyev**

Kaskelen 2023

Suleyman Demirel University
Faculty of Engineering and Natural Sciences
Department of Computer Science

Dean of Faculty

Associate Professor

PhD Zhamanov A.

06 2023

Topic of the thesis:

Implementing automatic word prediction and autocorrection for the Kazakh-language keyboard on the iOS platform

Thesis submitted as part of the requirements for the award of the MSc in "7M06102 - Computer Science" SDU, 2021-2023

Head of Department Assistant Professor, PhD Mukash Zh.

Academic Supervisor PhD Moldagaliyev B.

Master student Iskalinov F.

Kaskelen 2023

Ministry of Science and Higher Education of the Republic of Kazakhstan

Suleyman Demirel University

Farukh Iskalinov

Implementing automatic word prediction and autocorrection for the Kazakh-language keyboard on the iOS platform

THESIS

Presented in Partial Fulfilment for the

Master of Technical Sciences Degree in Computer Science

(degree code: 7M06102)

Department of Computer Science

Faculty of Engineering and Natural Sciences

Supervisor: **PhD Birzhan Moldagaliyev**

Kaskelen 2023

Suleyman Demirel University
Faculty of Engineering and Natural Sciences
Department of Computer Science

Dean of Faculty

Associate Professor

PhD Zhamanov A.

_____

« _____ » _____ 2023

Topic of the thesis:

Implementing automatic word prediction and autocorrection for the Kazakh-language keyboard on the iOS platform

Thesis submitted as part of the requirements for the award of the MSc in "7M06102 - Computer Science" SDU, 2021-2023

Head of Department _____ Assistant Professor, PhD Mukash Zh.

Academic Supervisor _____ PhD Moldagaliyev B.

Master student _____ Iskalinov F.

Kaskelen 2023

Declaration

I confirm that this is my own work and the use of all material from other sources has been properly and fully acknowledged.

Farukh Iskalinov

2023

Acknowledgements

I would like to express my deepest gratitude to my supervisor, PhD Birzhan Moldagaliyev, for his invaluable guidance, unwavering support, and immense knowledge throughout the entire research process. His expertise and insightful feedback have been instrumental in shaping this thesis.

I would also like to extend my heartfelt appreciation to my groupmates at the university who have provided valuable insights, discussions, and assistance during this research journey. Their contributions and collaborative spirit have greatly enriched my work.

I am grateful to the university for providing the necessary resources and facilities that have enabled me to conduct this research effectively. The academic environment and opportunities for growth have been truly inspiring.

Lastly, I would like to express my gratitude to my friends and family for their constant encouragement, understanding, and belief in my abilities. Their love and support have been a constant source of motivation throughout this endeavor.

This thesis would not have been possible without the support and contributions of all these individuals, and for that, I am truly grateful.

Dedication

I would like to express my heartfelt dedication of this thesis to my beloved family, whose consistent support and encouragement have served as the bedrock of my journey. Their unwavering faith in my abilities and the sacrifices they have made have consistently motivated me to pursue excellence.

Furthermore, this thesis is devoted to the individuals who have profoundly influenced and molded my intellectual and personal development. Their invaluable guidance, mentorship, and companionship have played a pivotal role in shaping both my research and my character.

Abstract

In the past few years, there has been a growing interest in natural language processing (NLP) and the development of language models for various applications, including text prediction and word correction in mobile device keyboards, aimed at improving user experience. However, iOS platforms lack special features adapted for Kazakh-speaking users. The main goal of this thesis is to develop and evaluate word prediction and autocorrection systems for the Kazakh language. Advanced methods such as LSTM and GRU are utilized to efficiently gather contextual information and predict the next word. Various error correction methods, including edit distance, N-gram based models, and a hybrid approach, are applied and evaluated. Experimental analysis using LSTM and GRU models allows for the optimization of hyperparameters and the improvement of word prediction accuracy. It was observed that the LSTM model achieved the highest accuracy. The hybrid approach achieves the highest accuracy of 91% among the evaluated error correction methods. Moreover, the integration of these models and methods with the iOS system enables the development of a fully-featured keyboard application specifically designed for the Kazakh language. This research contributes to the advancement of predictive and autocorrective technologies for next word prediction in Kazakh language processing, aiming to enhance the accessibility and usability of Kazakh language applications. It ultimately improves the overall quality of written communication and enhances user satisfaction with mobile keyboard applications.

Аңдатпа

Соңғы жылдары табиғи тілді өңдеуге (NLP) және мобильді құрылғы пернетақталарының пайдаланушы интерфейсін жақсарту үшін тілдік модельдерді әзірлеуге қызығушылық артып келеді. Алайда, қазақ тілді тұтынушыларға iOS платформасындағы пернетақтада қазақ тілін толық қамды пайдалану үшін бейімделген арнайы мүмкіндіктер жоқ. Бұл диссертацияның негізгі мақсаты сөздерді болжау және автокоррекциялау жүйелерін әзірлеу және бағалау. Контекстік ақпаратты тиімді жинау және келесі сөзді болжау үшін LSTM және GRU сияқты озық әдістер қолданылады Қателерді түзетудің әртүрлі әдістері, соның ішінде қашықтықты өңдеу, N-грамға негізделген модельдер және гибридті тәсіл қолданылады және бағаланады. LSTM және GRU модельдерін қолдана отырып, эксперименттік талдау гиперпараметрлерді оңтайландыруға және сөздерді болжау дәлдігін арттыруға мүмкіндік береді. LSTM моделі ең жақсы дәлдікке қол жеткізгені байқалды. Сөздегі қателерді түзету әдістері ішінде гибридті тәсіл ең жоғары дәлдікті, 91%, қамтамасыз етті. Сонымен қатар, осы модельдер мен әдістерді iOS жүйесімен біріктіру - арнайы әзірленген толық функционалды Қазақ тілді пернетақта қосымшасын жасауға мүмкіндік береді. Бұл нәтижелер қазақ тіліндегі сөздерді болжау және қателерді түзету әдістерінің тиімділігін түсінуге ықпал етеді, жазбаша коммуникацияның жалпы сапасын, тұтынушыоар арасындағы пернетақтаға қанағаттануын арттырады. Бұл зерттеу қазақ тілін өңдеуде, келесі сөзді болжау және автокоррекциялау технологияларын жетілдіруіне ықпал етеді, сондай-ақ қосымшаларда қазақ тілінде жазбаша пікір алмасу сапасын жақсартып, пернетақта тұтынушыларының ыңғайлылығын арттыруға өз септігін тигізеді.

Аннотация

В последние несколько лет наблюдается растущий интерес к обработке естественного языка (NLP) и разработке языковых моделей для различных приложений, включая предсказание текста и коррекцию слов в клавиатурах мобильных устройств, направленных на улучшение пользовательского опыта. Однако в платформах iOS отсутствуют специальные функции, адаптированные для казахскоязычных пользователей. Основная цель данной диссертации - разработка и оценка систем предсказания слов и автокоррекции для казахского языка. Применяются передовые методы, такие как LSTM и GRU, для эффективного сбора контекстуальной информации и предсказания следующего слова. Оцениваются различные методы коррекции ошибок, включая редактирование расстояния, модели на основе N-грамм и гибридный подход. Экспериментальный анализ с использованием моделей LSTM и GRU позволяет оптимизировать гиперпараметры и повысить точность предсказания слов. Было обнаружено, что модель LSTM достигает наивысшей точности. Среди оцененных методов коррекции ошибок, гибридный подход обеспечивает наивысшую точность - 91%. Более того, интеграция этих моделей и методов с платформой iOS позволяет разработать полнофункциональное приложение клавиатуры, специально разработанное для казахского языка. Данное исследование вносит вклад в развитие технологий предсказания и автокоррекции следующего слова в обработке казахского языка, с целью улучшения доступности и использования приложений на казахском языке. Оно также повышает общее качество письменной коммуникации и улучшает удовлетворенность пользователей мобильными клавиатурными приложениями.

Abbreviations

NLP - Natural Language Processing
DL - Deep Learning
ML - Machine Learning
LSTM - Long Short Term Memory
GRU - Gated Recurrent Unit
BiRNN - Bidirectional Recurrent Neural Network
BiLSTM - Bidirectional Long Short Term Memory
CONV1D - One-dimensional convolution
RNN - Recurrent Neural Network
CSR - Character Save Ratio
OCR - Optical Character Recognition
CNN - Convolutional Neural Network
DCIGN - Deconvolutional Neural Network
GPT - Generative Pre-trained Transformer
BERT - Bidirectional Encoder Representations from Transformer
ReLU - Rectified Linear Unit
NLTK - Natural Language Toolkit
KLC - Kazakh Language Corpus
URL - Uniform Resource Locator
RAM - Random Access Memory
CPU - Central Processing Unit
IDE - Integrated Development Environment
MVP - Minimum Viable Product

Table of Contents

Chapter 1

Introduction

1.1 Motivation and Background

The proliferation of the Internet and mobile devices has led to a significant expansion in textual communication. As users increasingly gravitate towards mobile devices, these portable and connected technologies have become integral to everyday communication practices [1]. Text-based information and its exchange, including SMS, email, chat rooms, and social networking communications, have become pervasive in modern life. The Figure 1.1 presents an overview of smartphone usage statistics spanning the past eight years.

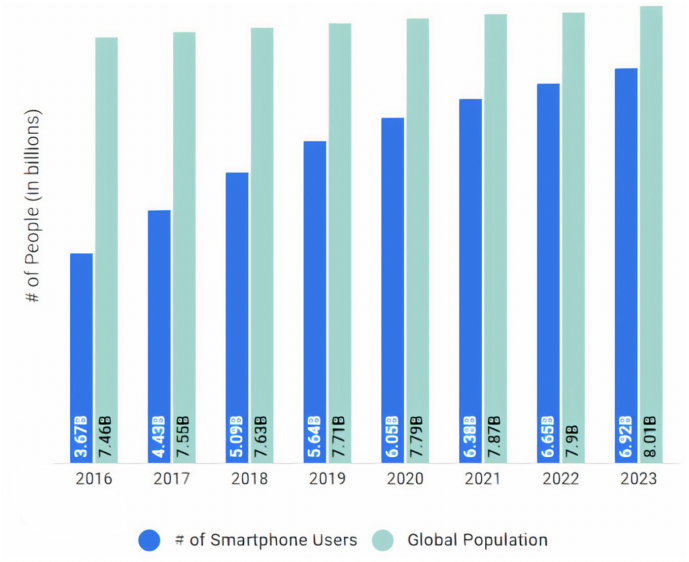


Figure 1.1: Growth of smartphone users

Consequently, intelligent input strategies have gained increasing importance. Typing on mobile devices can be time-consuming and error-prone, prompting the development of soft keyboards as a solution. Next-word predictions serve as a valuable resource in simplifying text entry, while auto-correct features rectify user errors. By analyzing a limited amount of previously entered text, language models can anticipate the most likely subsequent word or phrase.

Deep learning approaches have evolved into a potent tool in natural language processing (NLP). These methods have shown to be effective in a wide range of applications, including linguistic modeling, textual data classification, and automatic language translation. Complex patterns and representations may be retrieved from massive volumes of data using sophisticated algorithms and machine learning approaches. In a variety of natural language processing (NLP) applications, this can lead to enhanced performance and more accurate predictions.

Despite the encouraging outcomes of deep learning applications in multiple languages, low-resource languages like Kazakh continue to struggle with a lack of data and insufficient expertise in learning and applying these algorithms. It is a significant difficulty to develop excellent word prediction and autocorrection systems in Kazakh. The goal of this thesis is to look into the usage of deep learning technology to develop efficient and accurate word prediction and autocorrection systems with a focus on the Kazakh language.

The motivation behind this master's thesis is to develop an advanced model that improves the sentence writing process by offering predictive suggestions for the subsequent parts of a sentence, leveraging user input. Furthermore, this thesis aims to implement an algorithm for word auto-correction, with a specific emphasis on addressing the unique challenges posed by the Kazakh language.

The successful construction of a Kazakh keyboard with word prediction and correction capabilities has important practical consequences. It can facilitate faster and more accurate text input, improve writing efficiency, and enhance the overall user experience. Moreover, this study may contribute to the wider adoption and popularization of the Kazakh-language keyboard, encouraging its use among Kazakh-speaking individuals and communities. By promoting the effec-

tive utilization of the keyboard's features and functionalities, this research seeks to advance NLP skills specifically tailored to the Kazakh language, paving the way for future advancements and applications in this domain.

1.2 Problem Statement

Similar to other agglutinative languages, identifying misspelled words and making accurate word predictions or repairs poses significant challenges in the Kazakh language. This issue has become pervasive across various systems, including the pre-installed iOS keyboard. Since the Kazakh-language keyboard is still in its nascent stage on the iOS platform, developers have not yet incorporated auto-correction and word prediction capabilities.

Furthermore, it is important to highlight that there is a limited amount of scientific research specifically focused on developing word prediction and auto-correction for the Kazakh language. This makes this research particularly significant as it represents one of the pioneering efforts in exploring and developing such functionalities for the Kazakh language.

1.3 Aims and Objectives

The aim of this research is to implement next word prediction and auto-correction functionality for the Kazakh language within the iOS keyboard, utilizing Natural Language Processing (NLP) techniques. The primary objectives of this study include:

1. Create a detailed language model that is tailored to the Kazakh language. The proposed model incorporates sophisticated methodologies, specifically LSTM and GRU, to forecast the probable succeeding word by analyzing the contextual information of the input sentence..

2. Implement and assess the efficacy of deep learning models, specifically Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), under different hyperparameters. The present study aims to investigate the efficacy

of various model configurations in the context of word prediction and auto-correction tasks in the Kazakh language.

3. Implement and assess the efficacy of various error correction techniques, namely edit distance, n-gram based models, and a hybrid approach. The performance of each method will be evaluated and compared to determine their effectiveness in correcting errors. The efficacy of various techniques in identifying and rectifying spelling and typing errors in Kazakh language, along with their potential to enhance the precision and fluency of written text, will be evaluated.

4. Integrate next word prediction and auto-correction algorithms into the iOS keyboard interface. This would allow users to obtain real-time suggestions for sentence completion as well as automatically fix spelling and typing problems in Kazakh.

This thesis intends to contribute to the development of enhanced language processing capabilities for the Kazakh language, specifically inside the iOS environment, by attaining these aims and objectives.

1.4 Thesis Outline

The introduction and background of the research, as well as the research statement and objectives, are described in Chapter 1. Chapter 2 provides a review of relevant studies in the field. Chapter 3 delves into the theoretical foundations of machine learning, neural networks, language modeling techniques, and error detection/correction methods. In Chapter 5, the methodology utilized in the study is expounded upon, encompassing the techniques employed in data collection, preprocessing, algorithm development, and model evaluation. In Chapter 6, the experimental setups and comparative analysis of the auto-correction and word prediction systems are presented. In Chapter 7, the design and implementation of the Kazakh-language keyboard app is discussed, with a focus on the development environment and frameworks that were employed. The final Chapter 8 of the research provides a summary of the findings, implications, and contributions of the study, including discussions on potential future research directions.

Chapter 2

Literature Review

The following chapter provides a comprehensive examination of relevant studies regarding word prediction and auto-correction systems. It emphasizes significant discoveries and perspectives derived from these investigations.

2.1 Relevant Studies on Word Prediction

In a recent study [2], a deep learning-based language generation model was developed for medical recommendations in an Arabic language. The models they used for predicting the next word in medical recommendations included LSTM, BiLSTM, CONV1D, and LSTM-CONV1D, which are all deep learning architectures. The study had good results, and the CONV1D model got the highest matching score. The authors pointed out that there is room for more enhancements, like forecasting phrases or lengthier sequences of words, and integrating advanced models such as attention and transformers.

In this paper [3], the authors present a new method for predicting alarms in industrial environments by utilizing deep learning and natural language processing methods. In the case study, the proposed method was able to achieve an alarm prediction accuracy of around 80%, which shows that it is effective. The text shows why LSTM models are better than N-gram models and suggests ways to make them even better. This research adds to the existing knowledge on alarm management in industrial systems and demonstrates how deep learning and NLP

can be useful in this field.

The study conducted by the researchers aimed to investigate the effectiveness of different sequence-to-sequence deep learning models in generating novel dialogues between characters, as documented in [4]. The investigated models comprised of Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), and Bidirectional Recurrent Neural Network (BiRNN). Based on a thorough analysis and comparison of the models, the research findings indicate that the bi-directional RNN exhibited the least amount of loss, whereas the GRU exhibited the highest amount of loss. Additionally, it was observed that the GRU model demonstrated superior performance in terms of execution speed when compared to the bi-directional RNN model, which exhibited the lengthiest execution time.

In this study [5] a new methodology is presented for the prediction of optimal words and the subsequent generation of corresponding sentences in the Bangla language. The method proposed in this study employs a Gated Recurrent Unit (GRU)-based Recurrent Neural Network (RNN) that is trained on an n-gram dataset. The objective is to create language models that can accurately predict words. A comparative analysis was performed to evaluate the performance of LSTM-based RNN and Naive Bayes with Latent Semantic Analysis on an n-gram dataset.

The utilization of the Markov Chain Model for next word prediction based on user profiling is introduced in a scholarly article [6]. The paper proposes utilizing the Big Five Model to profile users' personalities and subsequently predicting the next word based on these traits. The utilization of the Markov Chain Model is implemented for the purpose of computing distances and proposing suitable vocabulary. The proposed research presents a new methodology for individualized next word forecasting.

Furthermore, researchers have explored the utilization of recurrent neural networks (RNNs) for next word prediction [7]. The study investigates the application of RNNs in predicting the next word in various contexts, including code completion, also describes how to increase prediction accuracy by using multi-window convolution techniques and residual-connected minimum gated units.

In these papers [8], [9], [10], the authors discuss the development and evaluation of a keyboard application for mobile devices that supports Swiss German, offering word prediction, completion, and correction mechanisms. The application demonstrates reasonable performance results and competes with similar applications. The prediction algorithm was evaluated using two different datasets, with a focus on the characters saved-to-total characters ratio (CSR). The effectiveness of the algorithm is evident, with a significant increase in CSR when adaptation is allowed. The findings align with the observations of SwiftKey, where users saved 33% of their characters.

Lastly, a paper presents a system that leverages TensorFlow for next word prediction and correction [11]. The system utilizes word vectors and RNNs to predict and correct misspelled words, enhancing human-computer interactions in social media and other platforms.

2.2 Relevant Studies on Auto-Correction

The study described in the paper[12] introduces a spell checker that is designed to identify and rectify real-word errors in Arabic text. The approach employs a language model based on word and stem n-grams, in conjunction with machine learning techniques, to achieve precise identification and rectification of errors. The results indicate that the system exhibits a high level of precision and recall, with an overall accuracy of 98% in the correction of single distance context errors. The employed methodology exhibits resilience and is well-suited for managing dyslexic text as well as post-OCR recognition of Arabic text.

In the study conducted by Amanjot Kaur[13], a hybrid approach was proposed for the implementation of a Spelling Checking and Correcting System, as documented in . The proposed methodology adopts a hybrid approach that amalgamates various techniques and incorporates language-specific linguistic features pertaining to the Punjabi language. The algorithm proposed by the researchers yielded an estimated accuracy of around 91% when applied to the input data.

In this research [14], the authors conducted a review of error detection and correction techniques for the Tamil language. The input words were subjected

to three different approaches for checking against a valid lexicon. These approaches include a tree-based algorithm, n-gram technique, and minimum edit distance technique. The authors evaluated the performance of these techniques using various sets of test words. The results of the evaluation demonstrate that the tree-based algorithm outperforms the other two techniques in terms of error detection. On the other hand, the n-gram technique, particularly when combined with stemmed words, provides the most suitable suggestions for correcting misspelt words. The testing results reveal that the developed system is highly effective in accurately detecting spelling errors and offering appropriate correction suggestions. The system achieves a minimum accuracy of 91%.

In this paper [15] proposed a Hindi spell-checker that utilized a word frequency dictionary as a language model. Error detection was performed using dictionary searches, while error correction involved the Damerau-Levenshtein edit distance and n-gram approach. The candidates for correction were ranked based on increasing edit distance.

In this study [16], a two-step DL model is proposed for detecting misspelled words in Turkish. The model includes a false positive reduction component to handle foreign words and abbreviations commonly found in online platforms. Various tokenization methods, such as character-based, syllable-based, and byte-pair encoding (BPE) approaches, are compared with LSTM and Bi-LSTM networks. The model demonstrates high accuracy and outperforms existing studies in spelling error detection. Future research directions include developing a correction model and exploring transformer-based architectures for sentence-level detection and correction.

Rakhimova [17] and Abdrazakh [18] conducted research on identifying and correcting incorrect words in the Kazakh language within social network data. They performed a comparative analysis of text correction systems, identified common errors, and developed a classification of errors in words. Their work contributes to enhancing language analysis and understanding in the Kazakh language within the context of social networks.

Chapter 3

Theoretical Background

This chapter provides a comprehensive overview of machine learning and deep learning algorithms used in the thesis project, focusing on neural networks. It covers theoretical foundations and practical applications of neural networks in the field of study.

3.1 Machine Learning

Machine learning is a discipline within the realm of computer science that employs mathematical models and computer algorithms to facilitate the learning process of machines, enabling them to improve their performance without the need for explicit programming. It is a subfield of Artificial Intelligence that employs algorithms to uncover hidden patterns in data. By utilizing historical data relevant to a particular problem, Machine Learning enables the prediction of desired outputs. There are three main categories within Machine Learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning, as illustrated in Figure 3.1.1.

3.1.1 Supervised learning

Supervised learning is a fundamental approach within the field of machine learning, wherein algorithms are trained using labeled data to predict outputs with a

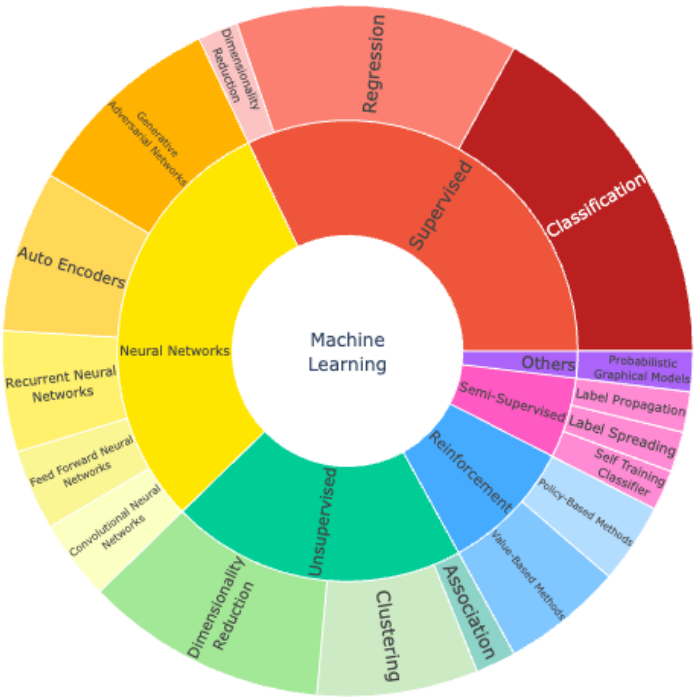


Figure 3.1.1: Classification of machine learning approach

high degree of accuracy. Mathematically, supervised learning can be represented by the equation:

$$y = f(x) \tag{3.1.1}$$

In Equation 3.1.1, the variable y denotes the predicted output, which is derived by applying a function to the input variable x. Through the process of supervised learning, the underlying function is automatically inferred based on the input values, enabling the generation of accurate predictions for the training data. Moreover, when new input values are introduced, the inferred function allows for making predictions accordingly.

Supervised learning encompasses two primary categories: regression and classification. Regression algorithms are employed when the objective is to predict real or continuous numerical values. For instance, these algorithms can be utilized to estimate housing prices, where the output is a continuous variable. Classification algorithms are deployed when the data needs to be classified into distinct categories. A classic example of classification is the prediction of tumor malignancy, wherein the algorithm categorizes tumors as either benign or malignant based on the provided data.

3.1.2 Unsupervised learning

Unsupervised learning represents another prominent approach within the realm of machine learning, wherein algorithms seek to uncover patterns and structures in unlabeled data. Unlike supervised learning, unsupervised learning does not rely on labeled examples for training. Instead, it focuses on discovering inherent relationships and organizing the data based on its intrinsic properties. Although unsupervised learning does not provide explicit output predictions, it plays a crucial role in exploratory data analysis and knowledge discovery.

Clustering is a widely used method in unsupervised learning that seeks to group similar data points together based on their intrinsic similarities or distances. Clustering algorithms, such as K-means or hierarchical clustering, can be applied to diverse domains ranging from customer segmentation in marketing to document clustering in natural language processing [19].

3.1.3 Reinforcement Learning

Reinforcement learning constitutes a distinct approach within the realm of machine learning, wherein an agent interacts with an environment and learns to make decisions and take actions based on feedback received. The agent, a key component of artificial intelligence, possesses the capability to acquire knowledge through its own experiences. Reinforcement learning differs from both supervised and unsupervised learning approaches in its fundamental principles and underlying mechanisms. While supervised and unsupervised learning methods rely on distinct learning paradigms, reinforcement learning introduces a unique framework centered around the concept of an agent interacting with an environment and learning through feedback [20].

The objective of this thesis is to utilize supervised learning algorithms for the prediction of the next word. The problem is posed as a regression problem that can be solved using deep neural networks.

3.2 Neural Networks

Neural networks, inspired by the functioning of the human brain, are computational models designed to uncover underlying patterns and trends within data. They utilize sophisticated algorithms to simulate the intricate processes of the brain, enabling them to excel in various tasks. The focus of this thesis is to leverage neural networks for the analysis of language sentences.

Neural networks have shown significant advancements in language modeling, outperforming traditional N-gram models that were previously proposed [21]. N-gram models, though widely used, face limitations in capturing long-range dependencies and contextual information. Neural networks, with their ability to capture complex patterns and relationships, have emerged as more effective models for language modeling tasks.

Moreover, neural networks demonstrate adaptability to handle continuously growing data, which is particularly relevant in the era of rapidly expanding datasets [22]. Their capacity to efficiently process large volumes of data enables continuous learning and improved performance over time.

In this thesis, the application of neural networks to language sentence analysis will be explored, harnessing their superior capabilities in capturing intricate linguistic structures and generating accurate predictions.

3.2.1 Architecture of Artificial Neural Networks

Artificial neural networks are structured with multiple layers, comprising input and output layers, in addition to one or more hidden layers. These layers consist of nodes, which are analogous to neurons in a biological neural network. The connections between nodes are referred to as arcs, and each arc is associated with a weight corresponding to a particular neuron. The neural network's output is computed through the application of an activation function to the weighted neurons. Activation functions that are frequently utilized comprise of sigmoid, hyperbolic, tangent, softmax, and various others [23].

The mathematical representation of the neural network illustrated in Figure

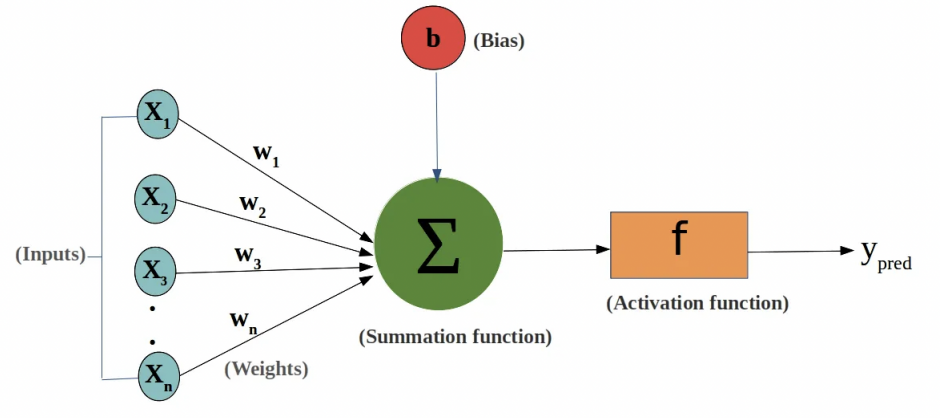


Figure 3.2.1: Basic Neural Network

3.2.1 corresponds precisely to Equation 3.2.1.

$$Y = h\left(\sum(W_i \cdot X_i + b_i)\right) \tag{3.2.1}$$

The equation represents a basic neural network model. In this equation, Y represents the output or prediction made by the neural network. The input variables are denoted by X, and they are multiplied by their corresponding weights, represented by W, where i is a single input-output combination. The weighted inputs are then summed up and combined with a bias term, denoted by b. The resulting value is passed through an activation function, f, which introduces non-linearity into the model and produces the final output, Y.

Components of the basic Artificial Neuron:

1. **Inputs:** Inputs represent a set of values used to forecast an output value. In the context of a dataset, inputs can be regarded as the attributes or features associated with the data.

2. **Weights:** Weights correspond to real values assigned to each input or feature, signifying their relative importance in predicting the final output.

3. **Bias:** The bias term in a neural network adjusts the activation function horizontally, similar to the y-intercept in a linear equation, enabling the network to shift and fit the data more effectively.

4. **Summation Function:** The summation function plays a crucial role in

combining the weights and inputs, calculating their cumulative sum.

5. **Activation Function:** The activation function is employed to introduce non-linearity into the model. By applying an activation function to the combined inputs, the neural network can capture complex relationships and patterns, enabling enhanced modeling capabilities.

The depth of a neural network refers to the number of hidden layers (Figure 3.2.2) it has between the input and output layers. Neural networks without hidden layers or with a single hidden layer are considered non-deep neural networks, while those with multiple hidden layers are classified as deep neural networks [24]. Deep neural networks are particularly effective at solving complex classification and regression problems. The addition of multiple layers provides the network with the ability to learn and model higher-level abstractions and features from the data. This makes the network more capable of handling diverse and multi-dimensional data inputs, leading to higher prediction accuracy and generalization across various use cases.

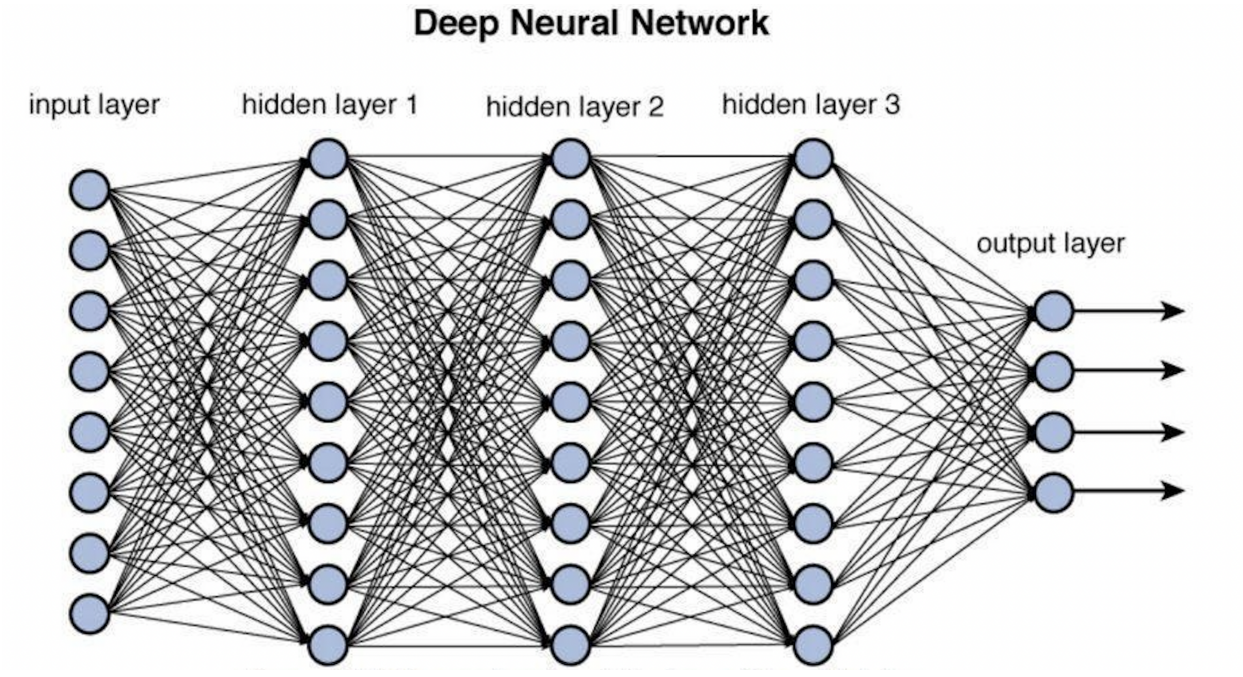


Figure 3.2.2: Deep Neural Network Architecture with Multiple Layers

The field of artificial intelligence has made tremendous strides in recent years, with significant advances in developing more accurate and robust artificial neural networks. These neural networks divided into three basic types: artificial neural

networks (ANNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs).

- Artificial Neural Networks (ANNs) are composed of layers of interconnected neurons that process input data in a forward manner. ANNs have gained significant popularity in various domains, particularly in tasks related to pattern recognition, including speech and image recognition. [25].
- Convolutional Neural Networks (CNNs) are specifically designed to handle image and video data. Unlike traditional Artificial Neural Networks (ANNs) with two-dimensional neuron positioning, CNNs employ a three-dimensional neuron positioning. The initial layer in a CNN is the convolutional layer, responsible for processing small sections of the visual field. CNNs are able to capture spatial patterns and characteristics in pictures, which has led to their effective application in a variety of fields, including object identification and recognition [26].
- Recurrent Neural Networks (RNNs) are specifically developed to handle sequential data, such as language modeling and time series data. They utilize recurrent connections within their layers to retain important information throughout the network and capture dependencies across time steps. This ability allows RNNs to provide insights into future events based on the input data. RNNs have proven effective in various applications, including automatic speech recognition and natural language processing.

The foundational neural network types, namely ANNs, CNNs, and RNNs, have paved the way for more advanced network architectures. Radial Basis Networks have been developed to enhance the performance of basic feed-forward networks, while CNNs have been extended with techniques like Deconvolutional Neural Networks and DCIGN. RNNs have also undergone significant advancements with LSTM and GRU, improving their ability to process sequential data. Despite RNNs' common use in language modeling, existing language models face challenges and limitations, highlighting the need for ongoing research.

Recent breakthroughs in language modeling have been achieved with models like the Markov Model, GPT, and BERT. These models, utilizing variations of

feed-forward and recurrent architectures, have demonstrated state-of-the-art performance in NLP tasks. As the field of neural networks continues to progress, it is anticipated that more sophisticated models will be developed by extending the foundational frameworks and integrating diverse architectures to tackle challenges in NLP and related domains.

3.2.2 Architecture of Traditional Recurrent Neural Networks

A recurrent neural network (RNN) refers to a network architecture wherein the neurons establish feedback connections among themselves. This characteristic affords the network a wide array of potential applications and configurations [27]. The versatility of recurrent neural network techniques is evident through their wide application to various problem domains. In the late 1980s, prominent researchers like Rumelhart, Hinton, and Williams introduced simple partially recurrent neural networks with the objective of learning character strings. Since then, numerous other applications have utilized RNNs to tackle challenges related to dynamical systems that involve time-sequenced events [28].

Recurrent Neural Networks (RNNs) have found application in various domains, including music, text, and dialogues, where they are utilized for sequence generation. However, RNNs suffer from a limitation wherein they struggle to retain information in memory over extended periods. Consequently, this drawback can induce instability in sequence generation processes. To mitigate this issue, the network can be designed to primarily focus on the most recent inputs and their corresponding predictions, enabling the model to learn from past errors and prevent instability. If the neural network possesses an extended memory, it can assimilate prior forecasts into its decision-making mechanism. Many researchers have endeavored to train Recurrent Neural Network (RNN) models to apprehend long-term dependencies. However, they have faced challenges related to vanishing gradients (mostly) or exploding gradients (occasionally), which significantly affect the model's performance.

Human beings have a tendency to access information stored in either short-

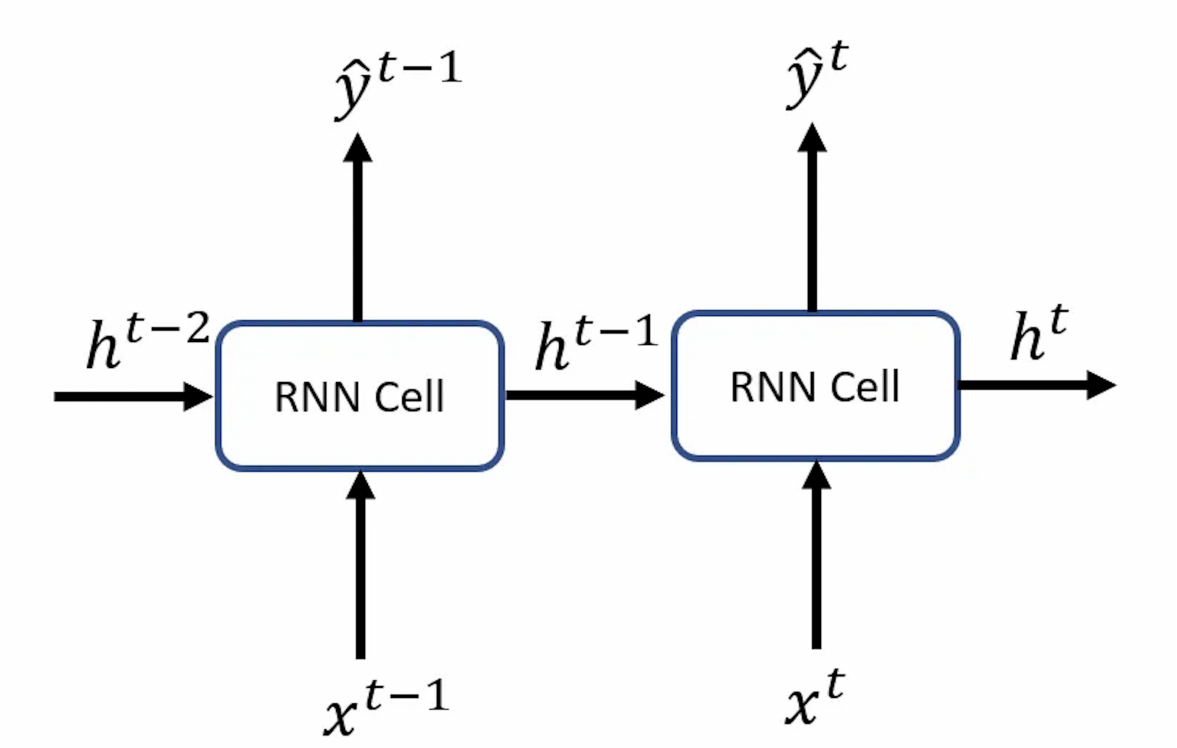


Figure 3.2.3: Simple RNN Architecture

term or long-term memory, combine it with current information, and employ logical reasoning to determine subsequent actions or responses based on past experiences. In a similar vein, the concept of enabling Recurrent Neural Networks (RNNs) to retain previous information or states aligns with this idea. Since the output of a recurrent neuron at a specific time step t relies on the previous input, which accumulates information up until time step $t-1$, this mechanism can be interpreted as a type of memory cell, the equation is shown in 3.2.2. Any component within a neural network that maintains or retains its state, even partially, across different time steps is commonly referred to as a memory cell. Every recurrent neuron possesses both an output and a hidden state, which is then transmitted to the subsequent neuron in the sequence. Figure 3.2.3 provides a visual representation of the structural components of a memory cell.

$$h_t = g(h_{t-1}, x_t) \tag{3.2.2}$$

3.2.2.1 The Problem of Long-Term Dependencies

Recurrent Neural Networks (RNNs) have gained considerable attention due to their ability to establish connections between preceding information and the current task, such as utilizing previous video frames to enhance comprehension of the present frame. The potential for RNNs to accomplish such connections highlights their value. However, the feasibility of this functionality varies depending on specific circumstances.

Certain scenarios allow for effective task performance by considering recent information alone. For example, in a language model aiming to predict the subsequent word based on prior words, having the context of *"the clouds are in the"* is sufficient to anticipate that the subsequent word will likely be *"sky"*. In such cases, where the gap between pertinent information and its required utilization is minimal, RNNs can effectively leverage past information.

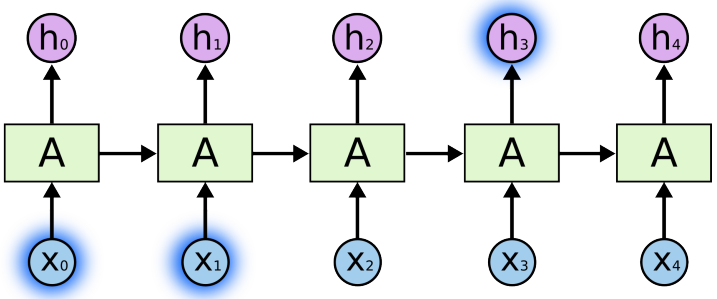


Figure 3.2.4: Short-Term Dependency

Figure 3.2.4 illustrates that calculating the output at h_3 involves a short-term dependency, necessitating X_0 and X_1, which is a simplistic undertaking for simple RNNs. Thus, it raises inquiries regarding whether RNNs are completely flawless and how they fare concerning extensive sequences and dependencies. In theory, RNNs possess the capability to accommodate long-term dependencies (Figure 3.2.5). However, practical limitations emerge as dependency gaps widen, and more context is mandatory for reliable learning. Such limitations result in potential model degradation and have been thoroughly studied in the research works [29, 30].

Certain complex scenarios necessitate a substantial augmentation of contextual

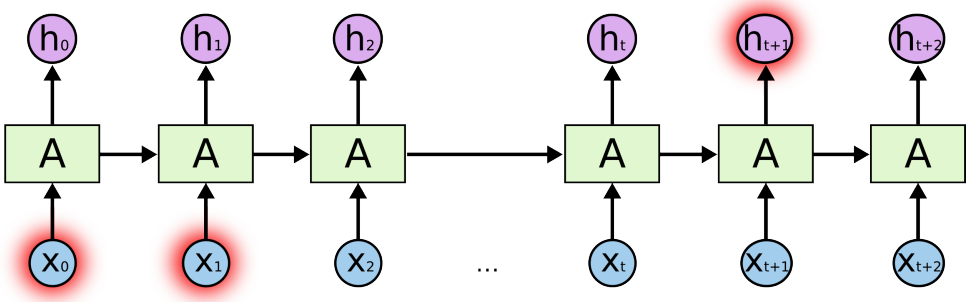


Figure 3.2.5: Long-Term Dependency

information to achieve accurate predictions. One such instance arises in the task of predicting the final word in a given sentence, exemplified by the statement "I was born in Turkey... I speak in *Turkish*". Recent studies indicate that the subsequent word is highly probable to represent the name of a language. However, in order to ascertain the specific language, it is imperative to consider the contextual information related to Turkey from an earlier period. It is plausible for the time gap between the pertinent information and its requirement to become significantly extensive.

According to theoretical understanding, Recurrent Neural Networks (RNNs) possess the inherent capability to effectively manage and model "long-term dependencies" [31].

3.3 Language Modeling Techniques for Word Prediction

These language modeling techniques have demonstrated their effectiveness in various natural language processing tasks, such as speech recognition, machine translation, sentiment analysis, and text generation. Their capability to capture long-term dependencies and handle sequences of different lengths makes them valuable for modeling sequential data. In the upcoming sections, we will provide a detailed exploration of the implementation and application of these models in the context of language modeling. Specifically, we will delve into two popular language modeling techniques: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models.

3.3.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks that incorporate a specialized gating mechanism, designed to regulate access to memory cells [32]. The history of LSTM networks dates back to the pioneering work of Sepp Hochreiter and Jürgen Schmidhuber in 1997 [31]. LSTM networks were developed as a solution to the vanishing gradient problem faced by traditional recurrent neural networks (RNNs), which made it difficult for RNNs to capture long-term dependencies in sequential data. The fundamental feature of LSTM networks lies in their ability to prevent the modification of memory cell contents for multiple time steps, facilitated by the gating mechanism. The LSTM model exhibits analogous chain structures to those of the RNN, albeit with divergent gated units.

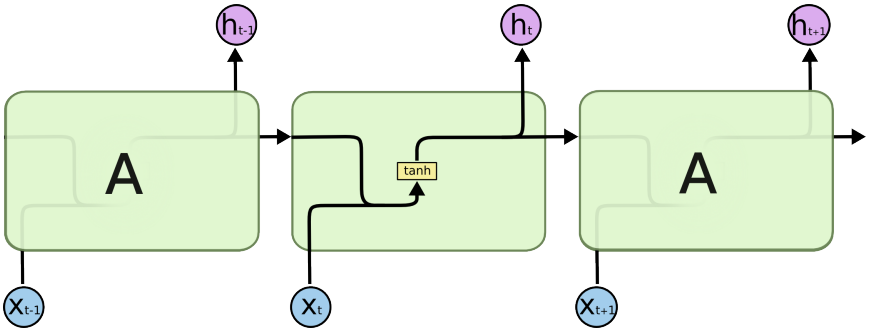


Figure 3.3.1: The Standard RNN with a Single Layer

Each instance of a recurrent neural network is comprised of a sequence of neural network modules that are repeated in a chain-like fashion. In conventional recurrent neural networks (depicted in Figure 3.3.1), the recurrent module is typically constructed with a simplistic architecture, such as a solitary hyperbolic tangent layer [32]. The hyperbolic tangent activation function restricts values to the interval [0, 1], thereby facilitating the model's ability to selectively forget or retain information. In the case where a sigmoid or tanh function yields a zero output, the corresponding value is deemed invalid. If the output is within the range of zero and one, the model will preserve the corresponding value.

LSTM networks also exhibit a similar chained structure. However, the recurring module within an LSTM network differs significantly. Rather than employing a single neural network layer, LSTM networks comprise four distinct layers that

interact in a highly specialized manner. This unique architectural design enables LSTMs to effectively capture and retain long-term dependencies in sequential data. The LSTM gated unit encompasses three distinct gates, which play a pivotal role in managing the storage and removal of preceding information based on the inputs provided to the model. These three gates are outlined below.

1. **Forget gate:** In the initial stage of our LSTM architecture, a crucial determination is made regarding the information to be discarded from the cell state. This decision-making process is facilitated by a sigmoid layer termed the "forget gate layer". By examining the prior hidden state h_{t-1} and the current input x_t, this layer generates a value between 0 and 1 for each element within the cell state C_{t-1} (Equation 3.3.1). A value of 1 signifies the decision to "completely retain" the corresponding information, whereas a value of 0 indicates the intention to "completely discard" it.

$$forgetgate_t = \sigma(W_f \cdot [h_{t-1}, x_t]) + b_f \tag{3.3.1}$$

2. **Input gate:** The subsequent stage involves determining the specific information to be incorporated into the cell state, which comprises two primary components. Initially, an "input gate layer" that utilizes a sigmoid function is responsible for deciding the values to be updated (Equation 3.3.2). Subsequently, a "tanh layer" generates a vector of prospective candidate values, denoted as $\hat{C}_t$ (Equation 3.3.3), that can potentially be appended to the current state, resulting in values confined within the range of -1 and 1. The subsequent step entails the integration of these two components to produce an updated version of the cell state.

$$inputgate_t = \sigma(W_i \cdot [h_{t-1}, x_t]) + b_i \tag{3.3.2}$$

$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3.3.3}$$

3. **Cell State:** The final gate within the LSTM cell is responsible for determining the subsequent hidden state, which encapsulates the information pertaining to prior inputs. This state pertains to the process of utilizing

the sigmoid function to evaluate the current input and the preceding hidden state. In other words, the transition from the previous cell state, C_{t-1}, to the updated cell state, C_t, takes place. Additionally, the cell state is subjected to the tanh function. To initiate the update process, the old cell state, C_{t-1}, is multiplied by the forget gate value, f_t, effectively discarding the information designated for forgetting. Subsequently, we add the multiplication of the input gate value, i_t, and the new candidate values, $\hat{C}_t$. These candidate values, $\hat{C}_t$, are appropriately scaled to reflect the extent to which each state value was decided to be updated. The equation C_t is shown below 3.3.4:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t \tag{3.3.4}$$

4. **Output Gate:** The final gate within the LSTM cell is responsible for determining the subsequent hidden state, which encapsulates the information pertaining to prior inputs. Initially, a sigmoid layer is employed to determine the specific segments of the cell state that will be outputted (Equation 3.3.5). Subsequently, the cell state is processed through a tanh function, which ensures that the values are confined within the range of -1 and 1. This transformed cell state is then multiplied by the output of the sigmoid gate. This selective multiplication allows for the output of only the designated segments of the cell state (Equation 3.3.5).

$$outputgate_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3.3.5}$$

The Equation 3.3.6 represents the output of the LSTM.

$$lstmoutput_t = outputgate_t \cdot \tanh(C_t) \tag{3.3.6}$$

An LSTM network with four interacting layers and all of the above equations and units are shown in the Figure 3.3.1:

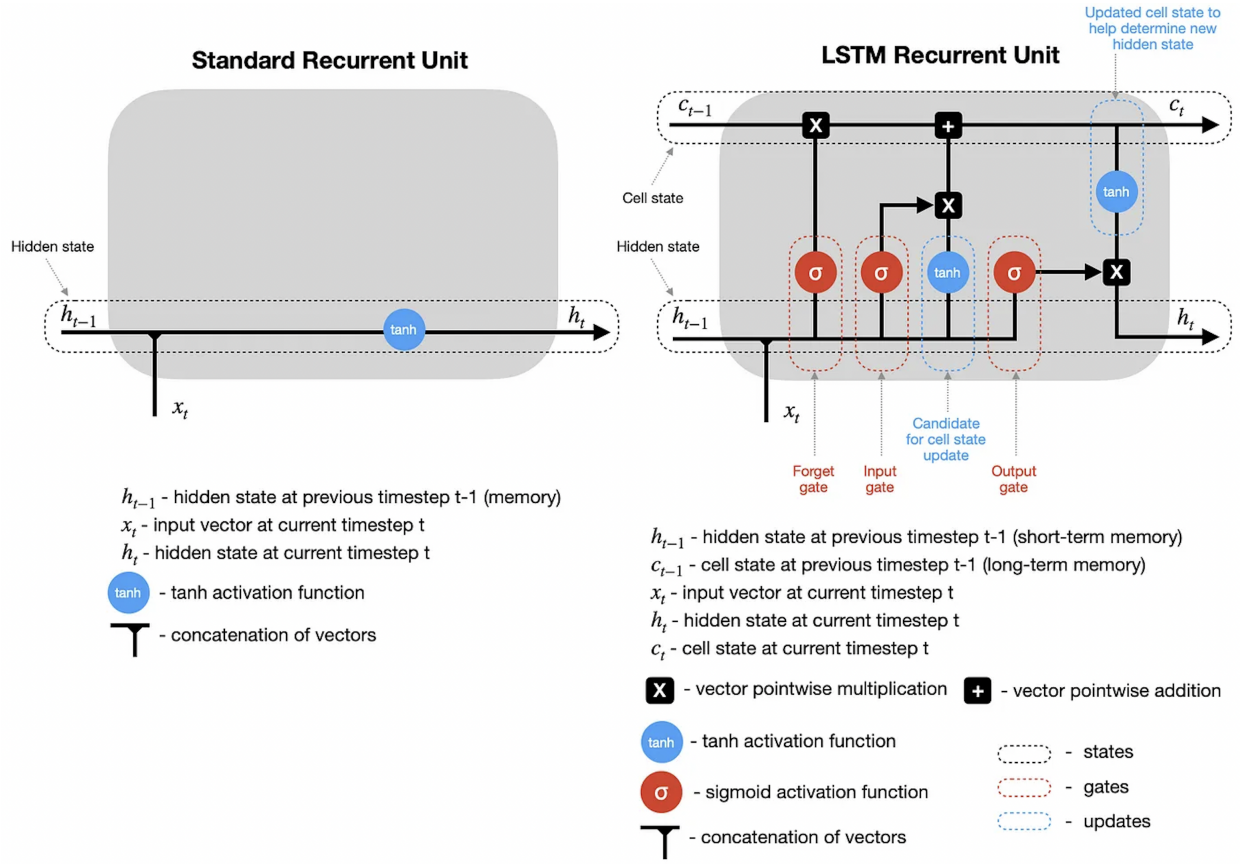


Figure 3.3.2: LSTM Network with Four Interacting Layers

3.3.2 Gated Recurrent Units (GRU)

Gated Recurrent Units (GRU) presents an alternative solution to address the issue of short-term memory in recurrent neural networks (RNNs) [33]. It belongs to the newer generation of RNNs and bears resemblance to Long Short-Term Memory (LSTM) networks. Unlike LSTM, GRU does not incorporate a separate cell state. Rather than that, it leverages the hidden state to enable the exchange of information among gates. Moreover, GRU features two gates, namely an update gate and a reset gate, as opposed to LSTM's three gates. These gates play a crucial role in controlling the flow of information within the network. While GRU has a simpler architecture compared to LSTM, it still exhibits the capability to capture long-term dependencies through the hidden state dynamics.

1. **Reset gate:** The reset gate within the GRU architecture performs a linear combination of the prior hidden state (h_{t-1}) and the present input (x_t) using a bias term and weighted sum. The resulting values are then passed through

a sigmoid function, which determines the extent to which the values should be discarded, remembered, or partially retained.

2. **Update gate:** The update gate in the GRU architecture plays a significant role in deciding which information should be preserved and which should be discarded. It is responsible for the long-term memory of the network and shares similarities with the combination of the input and forget gates in the LSTM architecture.

3. **Hidden state:** The hidden state candidate in the GRU model is formed by integrating the outputs with the new inputs (x_t) after resetting the previous hidden state in step two. This integration involves multiplying the outputs and inputs by their respective weights, adding biases, and passing the result through a hyperbolic tangent activation function in step six. Subsequently, the hidden state candidate is multiplied by the update gate's results in step seven. Finally, the modified hidden state candidate is added to the previously modified h_{t-1}, resulting in the generation of the new hidden state h_t.

$$z_t = \sigma(W_z X_t) + (U_z h_{t-1}) \tag{3.3.7}$$

$$r_t = \sigma(W_r X_t) + (U_r h_{t-1}) \tag{3.3.8}$$

$$h_t = \sigma(W X_t + U(r_t \otimes h_{t-1})) \tag{3.3.9}$$

The variables X_t , h_t, z_t, and r_t (as defined in equations 3.3.7, 3.3.8, 3.3.9) represent the input, hidden state, update gate, and reset gate vectors, respectively. The weight matrices W_z, W_r, and W are trainable parameters in the model. Additionally, the hidden state of the GRU structure produces output. The gates in the GRU structure are illustrated in Figure 3.3.3.

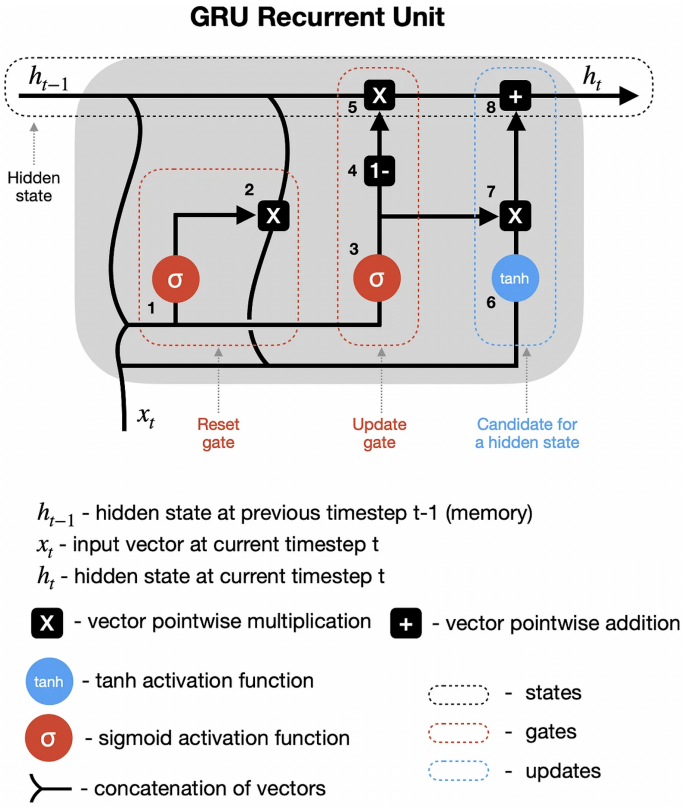


Figure 3.3.3: GRU recurrent unit

3.3.3 Activation Functions

Activation functions are mathematical functions employed in neural networks to enable neurons to learn and capture underlying patterns within a dataset. They play a crucial role in introducing non-linearity, allowing neural networks to model complex relationships. These functions determine whether a neuron should transmit a value or not, influencing the flow of information throughout the network.

The accuracy of a neural network's predictions relies on various factors, including the number of layers employed and, notably, the choice of activation function. Although there is no definitive guideline specifying the optimal number of layers for improved results and accuracy, a general rule of thumb suggests a minimum of two layers should be used. Similarly, literature does not prescribe a specific activation function for use. However, empirical evidence from studies and research demonstrates that incorporating one or more hidden layers in a neural network helps reduce prediction errors [34].

The primary role of an activation function is to transform the aggregated

weighted input from an individual neuron into an output value, which subsequently becomes the input for subsequent layers in a deep neural network. This process is repeated during training until the desired output is achieved. Figure 3.3.4 visually depicts an activation function's graphical representation for a single neuron.

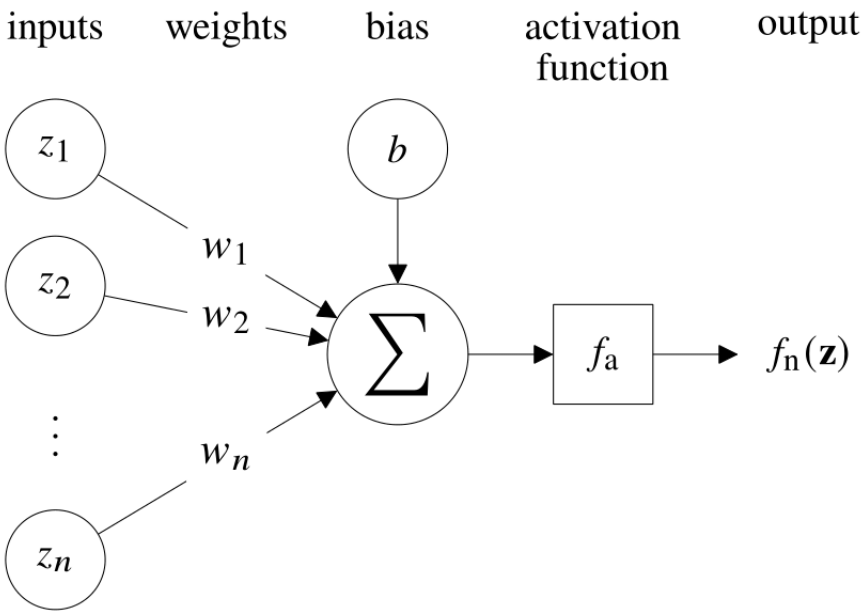


Figure 3.3.4: The activation function employed for a single neuron

3.3.3.1 Sigmoid Function

The Sigmoid Activation Function, also known as the logistic function, is a commonly used non-linear function in feedforward neural networks. It is bounded, differentiable, and defined for real input values. The equation 3.3.10 defining the sigmoid function is as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.3.10}$$

Neal [35] emphasized the notable benefits of sigmoid functions, which include their simplicity and ease of comprehension. These functions are commonly employed in shallow networks. However, Glorot and Bengio [36] recommended caution when initializing neural networks with small random weights, suggesting that the use of the Sigmoid AF in such cases should be avoided. It has been applied successfully in binary classification and logistic regression tasks. However, it has

certain limitations, including gradient saturation, slow convergence, and non-zero centered output. To address these drawbacks, alternative activation functions, such as the hyperbolic tangent function, have been proposed.

3.3.3.2 Hyperbolic Tangent Function

The tanh function, which is a hyperbolic tangent function, resolves the issue of non-zero centering that is present in the sigmoid function. It maps a real-valued number to the range [-1, 1] and is also a non-linear function. The derivative of the tanh function is similar to that of the sigmoid function.

Although the tanh function solves some of the drawbacks of the sigmoid function, it still does not completely eliminate the vanishing gradient problem. Its output is given by the following equation 3.3.11:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.3.11}$$

The tanh function has become the preferred choice compared to the sigmoid function in DL due to its better training performance for multi-layer neural networks. However, it is prone to the issue of vanishing gradient. The output generated is centered around zero, which facilitates the process of back-propagation. The limitation of the tanh function prompted researchers to explore alternative activation functions to address this issue, resulting in the development of the rectified linear unit (ReLU) activation function [37].

3.3.3.3 Rectified Linear Unit Function (ReLU)

ReLU or Rectified Linear Unit activation function, introduced by Nair and Hinton in 2010, is a popular choice in deep learning due to its superior performance and faster learning capabilities compared to other activation functions [38]. It effectively addresses the vanishing gradient problem but may encounter issues with "dead" neurons. Careful consideration should be given to selecting an appropriate activation function based on specific requirements. ReLU has a range of [0, +infinity] and offers benefits similar to the sigmoid function but with better performance and computational efficiency. The mathematical expression of the

equation is defined below 3.3.12:

$$f(x) = \max(0, x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{3.3.12}$$

3.3.3.4 Softmax Function

The Softmax function is commonly used in multi-class models to compute class probabilities, with each class assigned a probability value. It is frequently employed in the output layers of deep learning architectures.

The primary distinction between the Sigmoid and Softmax activation functions lies in their respective applications. The Sigmoid function is typically utilized in binary classification tasks, whereas the Softmax function is employed for multiclass classification tasks. The Softmax function introduces non-linearity into the network and is particularly beneficial when dealing with problems involving multiple classes. In contrast, the Sigmoid activation function is well-suited for solving binary classification problems. Mathematically, the Softmax function can be defined as (Equation 3.3.13):

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{3.3.13}$$

3.4 Classification of Methods for Error Detection and Correction

The procedure of spell checking encompasses two primary stages: detection and correction (Figure 3.4.1). Detection requires the inspection of an input string to validate its legitimacy as per a dictionary reference. In contrast, correction is aimed at identifying the optimal substitute for the detected error [39].

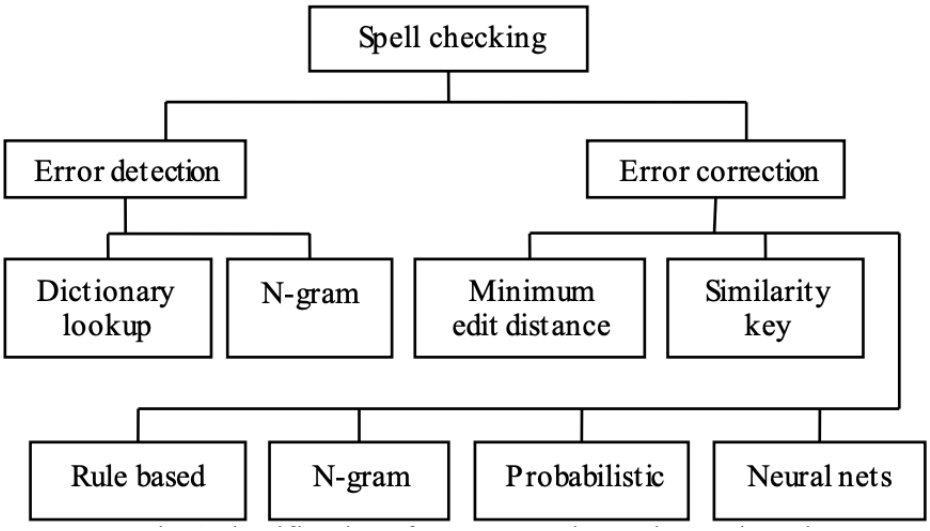


Figure 3.4.1: Classification of Methods for Error Identification and Rectification

3.4.1 Techniques for Error Detection

3.4.1.1 Dictionary Lookup Technique

The dictionary lookup method checks word input against a dictionary, ensuring correctness by having a word present. A hash table efficiently accesses the dictionary, evaluating input strings, calculating addresses, and extracting related words. If the calculated hash does not match the input string, an incorrect spelling error is detected.

3.4.1.2 N-Gram Analysis Technique

N-gram analysis uses n-grams, classified as unigrams, bigrams, or trigrams, as input strings. It uses a pre-compiled table of n-gram statistics to determine word frequency, recognize word occurrences, and make predictions using the collected statistics.

3.4.2 Techniques for Error Correction

3.4.2.1 Minimum Edit Distance Technique

The Minimum Edit Distance algorithm determines the smallest operations needed to convert a misspelled string into another. Three standard algorithms to calculate Minimum Edit Distance, which are Levenshtein, Hamming, and Longest

Common Subsequence. Levenshtein assigns a cost of 1 for each edit operation, while Hamming measures the distance between strings of the same length.

3.4.2.2 Similarity Key Technique

The Similarity Key technique involves mapping every string, both correctly spelled and misspelled, to a key that generates similar keys for correctly spelled strings. When a key is created for a misspelled string, it provides hints for similarly spelled words in the dictionary.

3.4.2.3 Rule-Based Technique

Rule-Based technique uses predefined rules to convert misspelled words into valid ones, using transition probabilities and confusion probabilities. Transition probabilities estimate n-gram frequency, while confusion probabilities measure letter mismatches.

3.4.2.4 Neural Networks

Neural networks offer a more advanced approach compared to the previous techniques. Back-propagation networks, which consist of output and input nodes, are commonly used in these methods. Each word in the dictionary corresponds to an output node, while potential n-grams at different positions in a word correspond to input nodes. The Backpropagation algorithm is employed to train the neural network.

3.4.3 Error Types

In morphologically rich languages like Kazakh, word forms are highly inflectional and agglutinative, resulting in extensive variation and flexibility. Spelling errors in these languages can be classified into typographic and cognitive errors. Cognitive errors involve difficulties in spelling a word due to lack of knowledge or familiarity with its correct form, while typographic errors arise from keyboard proximity and hand movements. Auto-correction algorithms focus on addressing misspelled words to improve text accuracy and readability.

3.4.4 Edit Distance Algorithm

3.4.4.1 Error Detection

The system starts by filtering out misspelled words. This process revolves around the comparison of the input word with a pre-established vocabulary in order to determine its correctness. A straightforward approach is employed through a dictionary lookup method to identify if the typed word exists within the dictionary. If the word that was typed cannot be located in the dictionary, then it is categorized as an error. Subsequently, an edit distance algorithm is employed to rectify the erroneous word and propose a suitable correction.

3.4.4.2 Word Suggestion Mechanism

Once a misspelled word is identified, the word suggestion mechanism generates a set of candidate words that have the potential to replace the misspelled word. While it is feasible to consider all correct words as candidates for correction, it is more practical to limit the search space and focus on words that bear resemblance to the identified spelling error. To rectify the error, we assess the structural similarity between the misspelled word and possible correct words using metrics such as the minimum edit distance or the Levenshtein edit distance.

The proposed simplification involves restricting the candidate list to words that differ from the misspelled word by just one edit operation, encompassing substitution, insertion, deletion, or replacement of consecutive letters.

Algorithm 1 Minimum Edit Distance Algorithm

Function `MinimumDistance`(*str1*, *str2*)**:**

```
m ← length(str1)  n ← length(str2)  Create a matrix E of size (m+1)×(n+1)
for i ← 0 to m do
    E[i][0] ← i
end
for j ← 0 to n do
    E[0][j] ← j
end
for i ← 1 to m do
    for j ← 1 to n do
        if str1[i − 1] = str2[j − 1] then
            cost ← 0
        end
        else
            cost ← 1
        end
        E[i][j] ← min(E[i − 1][j] + 1, E[i][j − 1] + 1, E[i − 1][j − 1] + cost)
    end
end
return E[m][n]
```

The minimum edit distance Algorithm 1 compares two strings, `str1` (the misspelled word) and `str2` (the possible correct word). It calculates the minimum number of edits required to transform `str1` into `str2`. A matrix of size $(m+1)\times(n+1)$ is created, where m and n represent the lengths of `str1` and `str2` respectively. The values in the matrix represent the minimum distance between corresponding substrings.

Using a cost of 1 for insertions or deletions, and a cost of 2 for substitutions, the algorithm determines the minimum cost needed to transform `str1` into `str2`. This cost corresponds to the minimum edit distance between the two strings.

The Levenshtein edit distance, based on this algorithm, helps identify the most likely correct word by minimizing the number of edit operations required to con-

vert the misspelled word (`str1`) to the possible correct word (`str2`).

3.4.4.3 Probability Distribution

To determine the most probable suggestion from the candidate words, a probability distribution mechanism is utilized. This mechanism calculates the probability of each word occurring in the language by analyzing a large text corpus. The candidate words are assigned probabilities based on their frequency of occurrence in the corpus. Frequency-based probabilities measure the likelihood of a word in a corpus, with higher probabilities assigned to more frequently encountered words. In automatic spelling correction, the probability $P(s|w)$ represents the likelihood of producing a string s instead of the word w, indicating similarity between the two strings. Brill and Moore (1997) introduced a framework that uses the maximum-likelihood principle to identify the best correction candidate w_{best} 3.4.1 from a list of possible candidates.

$$w_{\text{best}} = \arg \max_{w_i \in C(s)} P(s|w_i)P(w_i) \tag{3.4.1}$$

where $P(s|w_i)$ represents the probability of generating string s instead of word w_i, and $P(w_i)$ represents the probability of producing word w_i. The function $C(s)$ selects valid words from the dictionary W that serve as correction candidates for the erroneous string s.

This approach combines probability estimates with a candidate selection mechanism to determine the most suitable correction for a misspelled word, improving the accuracy of automatic spelling correction systems.

3.4.4.4 Replace Misspells

In the final step, the system replaces the misspelled word with the most probable suggestion obtained from the probability distribution mechanism. This replacement ensures the correction of the spelling error in the input text.

By incorporating these components, a statistical auto-correction system can effectively identify and correct misspelled words, enhancing the accuracy of typed

text. It is important to note that the simplified architecture presented here has its limitations compared to real-world auto-correction systems. In reality, advanced algorithms may incorporate more sophisticated techniques, such as natural language processing (NLP) and machine learning, to consider contextual information, grammatical patterns, and user-specific language models for improved accuracy. These advanced systems can handle a wider range of error types, including contextual errors and grammatical mistakes, leading to more accurate suggestions.

3.4.5 N-gram Model

Subsequent to the pioneering research, the utilization of n-gram models has found extensive application across a range of domains, encompassing speech recognition, machine translation, word correction, word prediction, and spelling correction [40]. In these models, the probability of the next word is assumed to depend solely on the previous N-gram, which represents a series of n preceding words.

The idea of utilizing n-gram language models has been applied to various applications, including speech recognition, machine translation, word correction, prediction, and spelling correction.

To estimate the conditional probability of a word at position t in a sentence, given the preceding words $w_{t-1}, w_{t-2}, \ldots, w_{t-N}$, the following formula is employed in Equation 3.4.2:

$$P(w_t|w_{t-1} \ldots w_{t-N}) \tag{3.4.2}$$

This probability estimation relies on counting the occurrences of these word sequences in the training data.The numerator of the estimation formula represents the number of times word t appears after the occurrence of words $t-1$ through $t-N$, while the denominator corresponds to the count of the N-gram sequence $t-1$ through $t-N$. The probability estimate equation 3.4.3 can be computed as:

$$\hat{P}(w_t|w_{t-1} \ldots w_{t-N}) = \frac{C(w_{t-1} \ldots w_{t-N}, w_t)}{C(w_{t-1} \ldots w_{t-N})} \tag{3.4.3}$$

However, a challenge arises when encountering N-grams with zero counts, which makes the estimation formula undefined. To address this issue, a k-smoothing technique is introduced. It involves adding a positive constant k to each numerator and k times the vocabulary size $|V|$ to the denominator. The modified probability estimation formula becomes 3.4.4:

$$\hat{P}(w_t|w_{t-1}\dots w_{t-N}) = \frac{C(w_{t-1}\dots w_{t-N}, w_t) + k}{C(w_{t-1}\dots w_{t-N}) + k|V|} \tag{3.4.4}$$

This adjustment ensures that even unseen N-grams have a non-zero probability estimate, by assigning a probability of $\frac{1}{|V|}$ to N-grams with zero counts.

In summary, the estimation of conditional probabilities in N-gram language models involves considering the preceding N words and utilizing counting techniques. The introduction of k-smoothing addresses the issue of zero counts, enabling a more robust estimation process.

Chapter 4

Implementation Tools and Frameworks

Python was chosen as the preferred language because of its diverse capabilities in machine learning and data processing. To solve certain problems, different tools and libraries were used. Short descriptions of the tools and sources are given in the Table 4.

Table 4.1: Implementation Tools and Frameworks

Tool/Library	**Description**
Pandas	It includes data structures and methods for manipulating numeric tables and time series.
NumPy	It provides multidimensional array-optimized implementation of computational methods.
NLTK	It offers a range of text processing libraries.
Scikit-learn	It provides a categorization, regression, and clustering techniques.
Keras with TensorFlow	It provides an API that facilitates the development of neural networks.
Matplotlib	It used for for building static, animated, and interactive visualizations.
(re) Library	It used to check for a pattern matching.
json Library	It used to parse JSON strings and files containing JSON object.
coremltools Library	It used to convert models into CoreML model.

Chapter 5

Methodology

In this chapter, we detail the data preprocessing process, which includes preparing the data to feed into the models, as well as all the experiments performed to evaluate the models. In addition, we will clarify the process of creating models, giving an idea of what is behind the development of models and their architecture.

5.1 Data

The Kazakh Language Corpus (KLC) was employed as the primary source of linguistic data for our thesis dataset. The corpus encompasses a diverse range of genres, comprising literary, publicistic, official, scientific, and informal texts. The compilation of a vast and all-encompassing corpus that accurately reflects the current state of the Kazakh language is considered a pioneering effort [41]. The KLC's "news" sub-corpus is predominantly composed of news articles written in Kazakh. The corpus follows a consistent document structure, with each document containing a title, source URL, metadata (including keywords), release date, author information, and the article text. The corpus encompasses over 400,000 categorized documents, providing a comprehensive depiction of the language across diverse domains and temporal epochs. The corpus comprises a total of 135 million words. The corpus was assembled from a diverse range of sources, encompassing websites, digitized books, dissertations, and articles from both public and private libraries.

Figure 5.1.1 presents a visualization of the top 10 words ranked by their frequency of occurrence. The figure provides an illustration of the frequency distribution of these terms. It is worth mentioning that a considerable proportion of these terms comprises of stop words, indicating their elevated occurrence rate within the given dataset. The aforementioned observation is consistent with the anticipated frequency of stopwords, which predominantly fulfill syntactic roles as opposed to communicating explicit content-related data.

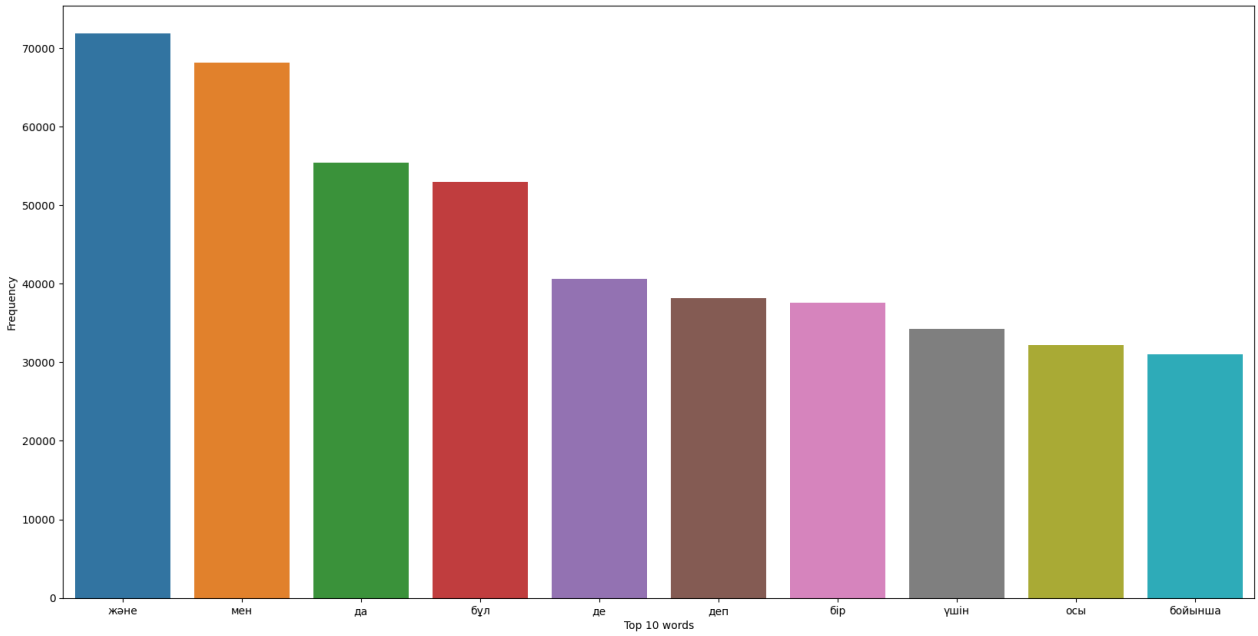


Figure 5.1.1: Frequency of occurrence of the top 10 words

Figure 5.1.2: Word Cloud of our corpus

5.2 Data Preprocessing

This section will provide a comprehensive explanation of the data preprocessing methodology. Text preprocessing plays a crucial role in extracting pertinent information from textual data by filtering out irrelevant or noisy elements. This preprocessing phase encompasses several key steps, including converting all words to a consistent case, removing punctuation marks and stop words, tokenization, word normalization, and other necessary procedures [42]. These steps are implemented to enhance the quality and usability of the data, ensuring that subsequent analyses and modeling efforts are based on a clean and standardized text corpus. To consolidate our data, which was originally stored in individual .txt files, it was imperative to merge them into a single file. This step was undertaken to facilitate data handling and enable seamless access and processing during subsequent stages of the analysis.

5.2.0.1 Data Cleaning

The text data in the datasets underwent several preprocessing steps. First, all text was converted to lowercase to ensure consistency in the representation of words. Next, special characters were removed using regular expressions to prevent the model from considering them as distinct entities. Another common technique employed in information retrieval is the elimination of stop words, which are highly frequent words that carry little or no relevant information for text categorization. In the case of the Kazakh language, there are numerous stop words that are extensively used and widely distributed. Examples of such stop words include 'осынау', 'өйткені', 'алайда', 'оған', 'сіздің', 'әркім', 'менде', 'әй', 'тарс-тұрс', 'қалт-құлт', 'тек', and others [43].

The process of removing stop words can be visualized in Listing 2.

5.2.0.2 Tokenization

Tokenization, a process of breaking down unstructured data and natural language text into discrete elements, was applied. This facilitated the representation of information as chunks, which could be directly utilized as vectors for machine

learning tasks. Tokenization transformed the unstructured text into a numerical data structure suitable for machine learning algorithms. Tokens also enabled computers to trigger useful actions and responses.

Upon completing the data preprocessing phase, which involved the removal of redundant spaces, special characters, and stop words, as well as the tokenization of the text into individual words or tokens, the dataset was effectively prepared for subsequent analysis and modeling. The cleaned dataset comprises approximately 70,000,000 words.

5.3 Data Splitting

In developing and evaluating our machine learning models, we utilize a common data splitting strategy. The dataset has been partitioned into three distinct subsets: the training set, validation set, and test set, with distributions of 80%, 10%, and 10% respectively. The training set (80%) is used for model learning, parameter adjustment, and pattern identification. The validation set (10%) provides an unbiased evaluation during training, aiding in parameter tuning and preventing overfitting. The test set (10%) serves as an unbiased evaluation of the final model's performance and represents its generalization capabilities.

5.4 Developing the Auto-correction Algorithm

In this section we discuss the development and evaluation of an autocorrection algorithm for detecting and correcting spelling errors. Our proposed method combines several approaches to improve the accuracy and efficiency of the autocorrection system. We provide a comprehensive review of each approach, illustrating their application through relevant examples.

5.4.1 Edit Distance Approach

In our research, we propose algorithms for word auto-correction, specifically targeting non-word errors, including typographical errors. To provide a detailed

methodology, we outline the following steps: Firstly, we employ a direct dictionary lookup method to detect error words. By searching for the typed word in a predefined dictionary, we determine if it is present. If a word is not present in the dictionary, it is deemed as an error.

To correct the identified error, we utilize the edit distance algorithm, which estimates the structural similarity between the misspelled word and possible correct words. The minimum edit distance, measured as the minimum number of operations required to transform one string into another, is calculated. In the context of spell checking, suppose we have two strings: S1 = [a1 a2 a3. . . an] and S2 = [b1 b2 b3. . . b m]. The aim is to determine the cheapest way to convert S1 to S2 using three fundamental operations: insertion, deletion, and replacement. In this scenario, each operation is assigned a unit cost, which is factored into the calculation of the edit distance. In our approach, we employ the Levenshtein algorithm to compute the edit distance. For the strings of different length, sequence or latent alignment technique is applied and then the above procedure is repeated.

If the first string is "мектеп" and the second string is "Мектеп", the distance is zero as no alterations are required since the strings are already identical. On the other hand, if str1 is "мектеп" and str2 is "мектап", then the edit distance is 1, since a single substitution is needed to convert 'a' into 'e' (Figure 5.4.1).

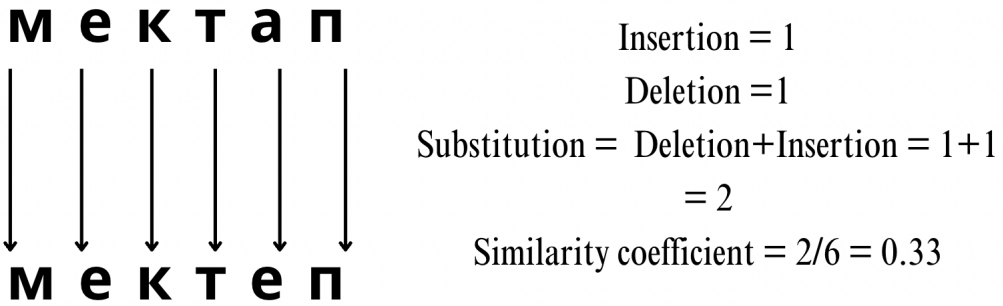


Figure 5.4.1: Edit Distance Example

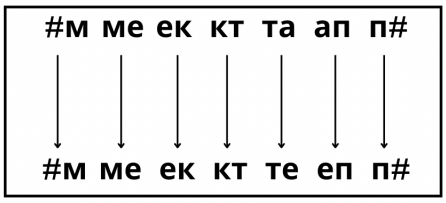
5.4.2 N-gram Based Approach

The concept of employing n-grams in language processing was initially introduced by Shannon [44]. This technique has been developed to handle a number of problems since its introduction, including word prediction, spelling correction, speech recognition, translation accuracy, and string searching. One significant advantage of employing n-grams is that they are language-independent.

Using n-gram statistics and word resources, this approach detects accurate recommendations by assigning weights to a variety of alternative repair possibilities. This is useful for determining potential adjustments and spotting erroneous wording. The n-gram model calculates similarities between two strings by counting the number of common n-grams. The bigger the number of shared n-grams, the more similar the two strings. The similarity coefficient is based on this idea.

$$\delta(a,b) = \frac{|\alpha \cap \beta|}{|\alpha \cup \beta|} \tag{5.4.1}$$

In this equation 5.4.1 , α and β represent the n-gram sets of two words a and b that are being compared. $|\alpha \cap \beta|$ signifies the count of shared n-grams in α and β, while $|\alpha \cup \beta|$ represents the total count of distinct n-grams in the combined set of α and β.



Bigram union

	ме	ек	кт	та	ап	те	еп
мектап	1	1	1	1	1	-	-
мектеп	1	1	1	-	-	1	1

Similarity coefficient
= 3/7= 0.4

Figure 5.4.2: N-gram Example

Once the n-gram method is used to calculate the similarity coefficient, it finds both the count of matching n-grams and unique n-grams in the two strings "мектап" and "мектеп". The Figure 5.4.2 demonstrates an example of how the similarity coefficient is calculated for the misspelled word "мектап" and the correct word "мектеп", using a 2-gram (bigram) approach. Following the computation of the similarity coefficient using the n-gram method, it extracts both the quantity of matching n-grams and the number of unique n-grams found in the two strings, namely "мектап" and "мектеп".

5.4.3 Hybrid Approach using Edit Distance and N-gram

Our suggested hybrid approach leverages the strengths of both n-gram and edit distance features. As illustrated in the Figure 5.4.3, the initial step involves segmenting the incorrect and correct words into bigrams.

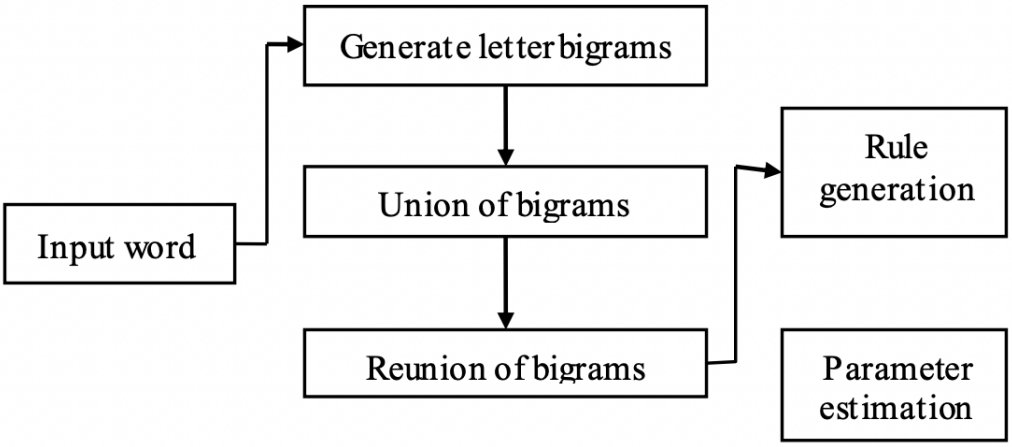


Figure 5.4.3: Operational Procedure of Hybrid Approach

Algorithm 2 Hybrid N-Gram Based Edit Distance Algorithm

Input: string pairs $(str1[i], str2[i])$

Output: transformation rule

Apply Bigram to the pairs $(str1[i], str2[i])$

Extract common bigrams from $(str1[i], str2[i])$

Determine the union of bigrams from $(str1[i], str2[i])$

Utilize n-gram based edit distance to construct transformation rule

Consider our example with the words (мектап, мектеп), the steps of the Hybrid N-gram and Edit Distance Algorithm 5.4.3 are:

1. Obtain the string pairs (str1[i], str2[i]) where str1[i] represents the incorrect word "мектап" and str2[i] represents the correct word "мектеп".
2. Apply the Bigram model on the string pairs (str1[i], str2[i]). The bigrams for the word "мектап" become {ме, ек, кт, та, ап} and for the word "мектеп", they become {ме, ек, кт, те, еп}.
3. Identify the common bigrams between str1[i] and str2[i]. The common bigrams between "мектап" and "мектеп" are {ме, ек, кт}, indicating segments of the string that don't require transformation.
4. Determine the union of bigrams from str1[i] and str2[i]. The union of the bigrams, in this case, is {ме, ек, кт, та, ап, те, еп}, which represents the complete set of unique bigrams present in both the words.
5. Apply the n-gram based edit distance method and formulate a transformation rule. For the unique bigrams that aren't common to both words, "та" in "мектап" and "еп" in "мектеп", we generate a transformation rule: replace "та" in "мектап" with "еп". This rule serves to effectively transform "мектап" into "мектеп", minimizing the edit distance by harnessing the N-gram model.

A representation of the applied example can be discerned from the subsequent Figure 5.4.4.

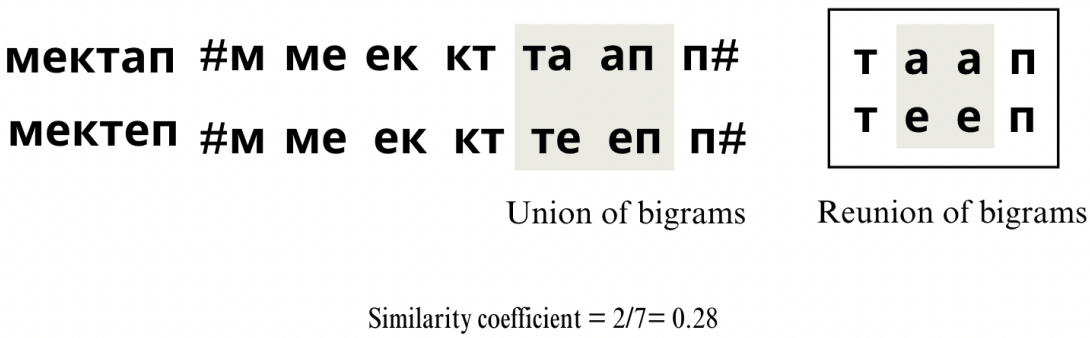


Figure 5.4.4: Hybrid Approach using Edit Distance and N-gram

5.5 Building the Word Prediction Model

This section discusses developing language models for prediction of the next word. Two models were created and trained on a common dataset, serving as the initial starting point for implementation. As mentioned in Chapter 3, language modeling is treated as a regression problem, where tokenization is employed to convert text into individual tokens or words. To achieve this, the Keras tokenizer class is utilized to generate integer sequences for model training. The process of tokenization is executed as a regression model, whereby the anticipated outcomes are integer sequences that can subsequently be transformed into corresponding words.

5.5.1 LSTM

The LSTM model is a type of deep learning model that utilizes artificial "cells" to handle memory, making it more effective for text prediction compared to traditional neural networks and other models. Our model consists of an embedding layer, followed by bidirectional LSTM (Bi-LSTM) and LSTM layers (Figure 5.5.1).

The Embedding layer is taught to embed all words in the training dataset using random weights. The Embedding layer is the network's initial hidden layer. Word embedding is a method of encoding words and texts using dense vector representations. Deep Learning also includes sentence embedding, which is used in Unified Sentence Encoder for more complicated language modeling applications [45]. Sentence embedding attempts to capture the semantic meaning of complete phrases or documents, allowing the model to comprehend and reason about the contextual information in natural language text.

It also includes a dropout layer with 0.2 value of dropout for regularization and two dense layers for classification. The first dense layer has 50 units, while the second dense layer serves as the output (softmax) layer and has a number of units equal to the size of the vocabulary. The model is compiled with categorical cross-entropy loss and the Adam optimizer. A learning rate scheduler is implemented to adjust the learning rate during training based on the epoch. Model checkpoints and learning rate scheduler callbacks are used to save the best model and adjust

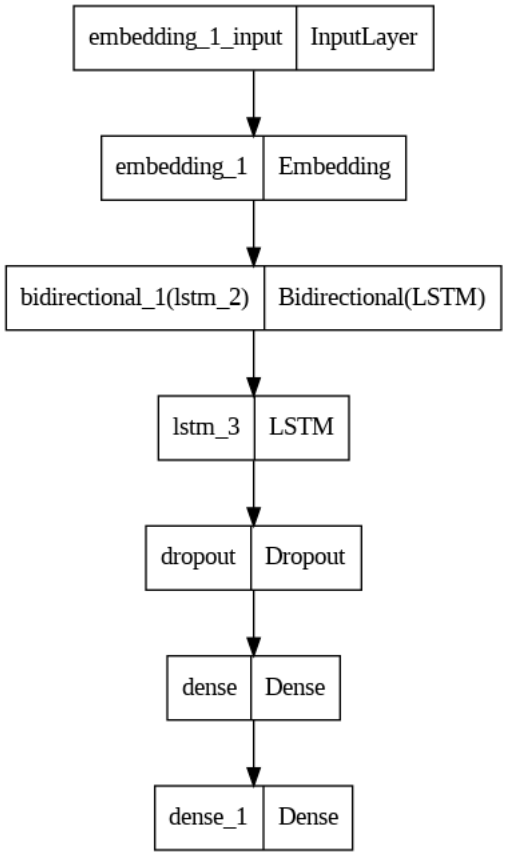


Figure 5.5.1: Layers of the LSTM Model

the learning rate (see Listing 8).

5.5.2 GRU

In this section, we'll be exploring a language model based on Gated Recurrent Units (GRU). As pointed out in Chapter 3, GRU models, much like LSTMs, are frequently used for generating sequences. Our GRU-based language model follows the same architectural approach as the previously discussed model. The architecture of this model comprises of a 1000-word embedding layer and GRU layers with 300 cells each, interspersed with two dropout layers set at a rate of 0.2% (see Listing 9). The rate denotes the proportion of neurons present in the layers that will be excluded during every iteration, commonly referred to as an epoch. The model predictions are compared with actual words, and the difference (the error) is used to update the model parameters through backpropagation and gradient descent optimization process. This iterative process continues until the model's predictions align well with the actual outcomes, or the predefined

maximum number of epochs is reached.

5.5.3 Optimization of Model Parameters

Optimization of model parameters, also known as hyper-parameter tuning, is an essential step in the construction of machine learning models. The goal of this process is to fine-tune the parameters of the model to improve performance and increase predictive accuracy [46].

Hyper-parameters are the configuration variables of the model, which influence the learning process and control the behavior of the model. They are set prior to the commencement of the learning process and remain constant throughout. The parameters under consideration include the size of the batch, types of activation functions, inclusion of dropout layers, count of hidden layers, choice between LSTM or GRU layers, rate of learning, number of training iterations or epochs, selection of optimizer, and the neuron count in each layer (Table 5.1).

Table 5.1: Parameter setting to train the proposed models

Adjustable Parameters	**Assigned Values**
Learning rate	0.0001, 0.001, 0.01
Epochs	30, 60, 100
Batch size	32, 64
Optimisation Algorithm	Adam
LSTM, GRU (layers)	3
Dropout Layer	0.2

Fine-tuning these parameters is essential as it can greatly enhance the performance of the model. While it can be computationally intensive and time-consuming, the outcome often leads to models that perform better on unseen data, providing more accurate and reliable results.

The hyper-parameter tuning process is critical in creating efficient and effective models, and understanding the role each hyper-parameter plays in the model's operation is key to achieving optimal performance.

- **Learning rate:** The learning rate, a critical hyperparameter, guides the adjustment of weights during model training, balancing speed and accuracy.

If set too high, it may lead to overshooting and unnecessary oscillations, prolonging the training process. Conversely, a smaller learning rate can enhance precision and improve the model's overall accuracy [47]. Recognizing the complexity involved in managing learning rates, most deep learning frameworks provide automated tools, like a learning rate scheduler, to handle this aspect effectively.

- **Epoch:** The number of epochs, a vital hyperparameter, denotes the total iterations over the complete training dataset. Each epoch gives each sample in the dataset a chance to update the model parameters. Usually set to a high value, running the algorithm for numerous epochs allows for thorough minimization of the model's error [48]. Creating line plots, or "learning curves," with epochs on the x-axis and the model's error or skill on the y-axis, can provide insights into the model's learning progress, revealing if it's overfit, underfit, or appropriately trained 5.5.2.

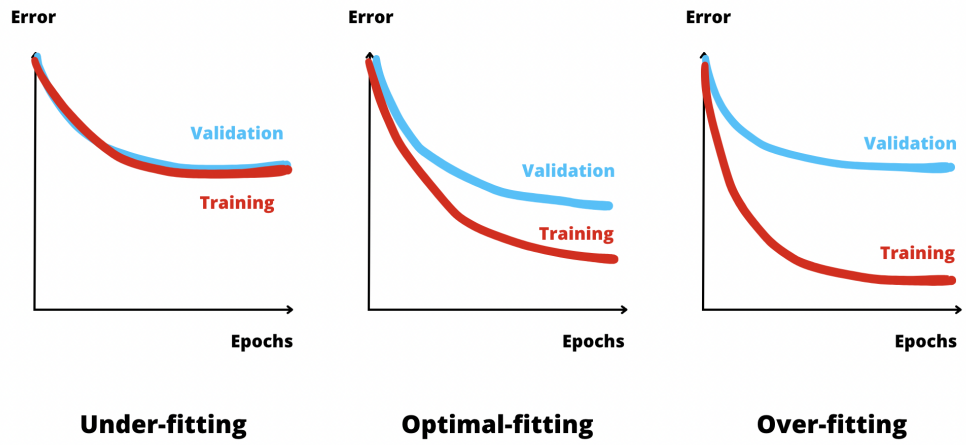


Figure 5.5.2: Over-fitting, Optimal-fitting, and Under-fitting in Deep Learning

- **Batch size:** The batch size is a hyperparameter that sets the number of data samples that the model sees before it updates its internal parameters. Essentially, a batch is like a loop iterating over some samples, making predictions, and then comparing those predictions to the expected outputs. The resulting error is used to improve the model, primarily by moving along the error gradient. The batch size should ideally be at least one but not more than the total number of samples in the training dataset [48].

- **Optimizer:** Optimization algorithms, often referred to as optimizers, are crucial for enhancing both the accuracy and the efficiency of the training process. However, choosing the right optimizer can be a complex task. It's worth noting that Adaptive Momentum Estimation (Adam) often yields favorable outcomes rapidly across diverse neural network architectures. The Adam optimizer is an optimization algorithm that is used in deep learning models to manage the learning rate, effectively enhancing the speed and performance of model training. It accomplishes this by leveraging both the first and second moments (mean and variance) of the gradients, which aids in adjusting the learning rate based on the recent steps, ensuring swift convergence and reducing the training time [49].

- **Dropout:** Dropout is a training technique in neural networks where a random selection of neurons is ignored during each training cycle [50]. These "dropped-out" neurons do not contribute to the activation of downstream neurons during the forward pass, and their weights are not updated during the backward pass. Dropout is implemented by randomly selecting nodes to be dropped-out with a specified probability, typically around 20%, during each weight update cycle. It is important to note that dropout is only applied during the training phase and is not used when evaluating the performance of the model.

5.6 Evaluation Methods

The present thesis employs various evaluation techniques to evaluate the predictive model for the subsequent word and auto-correction methods.

5.6.1 Accuracy

The assessment of model performance relies heavily on the fundamental evaluation metric of accuracy. The metric evaluates the accuracy of predicted responses by computing the ratio of correct predictions to the overall number of instances in the test dataset. The accuracy metric is determined through the division of the summation of true positives (TP) and true negatives (TN) by the summation

of true positives, false positives (FP), true negatives, and false negatives (FN). This metric offers significant insights into the models' capacity to produce precise responses to the provided inquiries.

When the aforementioned text is expressed in a mathematical equation, it can be represented as follows 5.6.1:

$$\text{Accuracy} = \frac{TH + TN}{TP + FP + TN + FN} \tag{5.6.1}$$

5.6.2 Perplexity

Perplexity is a widely employed assessment metric for language models, which gauges the efficacy of a probability distribution or model in predicting a given sample. Perplexity is computed through the process of exponentiating entropy. A reduced perplexity value is indicative of superior predictive capabilities. Perplexity is commonly computed in logarithmic space and subsequently converted to its initial format, as stated by the source [48].

Perplexity can be mathematically defined in statistical theory through the utilization of the formula denoted as 5.6.2.

$$PP = 2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 p(s_i)} \tag{5.6.2}$$

where N represents the number of words in a test dataset.

5.6.3 Cross-Entropy

The notio cross entropy is employed in the fields of information theory and machine learning to quantify the dissimilarity between two probability distributions. A prevalent practice in assessing the efficacy of a predictive model is to contrast the projected distribution with the actual distribution.

The cross entropy between two probability distributions A and B can be mathematically computed using the equation 5.6.3.

$$H(A, B) = -\sum_{x} A(x) \log B(x) \tag{5.6.3}$$

Where:

- $H(A, B)$ represents the cross entropy of distribution B to distribution A.
- $A(x)$ represents the probability of event x according to distribution A.
- $B(x)$ represents the probability of event x according to distribution B.

This formula allows us to quantify how well the distribution B matches the true distribution A. A lower cross entropy indicates a closer match between the two distributions.

5.6.3.1 Categorical cross entropy

Categorical cross entropy is a specific form of cross entropy used when dealing with categorical data or multi-class classification problems. It is commonly used as a loss function in machine learning models to measure the dissimilarity between predicted class probabilities and the true class labels.

Mathematically, the categorical cross entropy between a true probability distribution Y and a predicted probability distribution P is calculated as follows 5.6.4:

$$H(Y, P) = -\sum_{i=1}^{C} Y_i \log P_i \tag{5.6.4}$$

In this formula, $H(Y, P)$ represents the categorical cross entropy, Y_i denotes the true probability of class i, and P_i represents the predicted probability of class i. C represents the total number of classes. The categorical cross entropy loss penalizes larger discrepancies between the predicted and true probabilities. It encourages the model to improve its predictions to better match the true class probabilities.

Chapter 6

Experiments and Results

6.1 Experimental setups

All conducted tests take place in a setting featuring a MacOS M1 Pro platform, powered by a Core i7 processor and supplemented with 32 GB of RAM. A detailed outline of the experimental environment is presented in Table 6.1 below.

Table 6.1: System equipment

Component	**Description**
CPU	Core i7
Language	Python
Operating System	MacOS M1 Pro
System Type	64-bit
Memory	32 GB

6.2 Comparative Analysis

6.2.1 Auto-correction

In this section, we present the results obtained from experimental analysis conducted on our pre-processed dataset. The dataset initially comprised accurately spelled words. However, for evaluation purposes, deliberate modifications were introduced, resulting in both non-word errors and real-word errors within the text.

To introduce intentional errors, we categorized them into eight distinct categories.

1. Insertion of a single letter within a word
2. Deletion of a single letter from a word
3. Substitution of a single letter within a word
4. Transposition of two adjacent letters within a word
5. Presence of a single character different from the original word
6. Addition or removal of one letter along with one different letter
7. Addition or removal of repeated characters and one different character
8. Exchange of two consecutive letters and one different character

A test set consisting of 3000 misspelled words was created. Our proposed algorithm successfully corrected 2850 (95%) of the misspelled words, while encountering difficulties in rectifying the remaining 150 words. Furthermore, the Edit Distance approach exhibited a 91% accuracy in correcting misspelled words, whereas the N-gram approach achieved an 88% accuracy in word correction. The Table 6.2 below presents a comparative analysis of the accuracy for the three approaches.

Table 6.2: Correction Accuracy Comparison

Approach	Correction Accuracy (%)
Proposed Hybrid Algorithm	95
Edit Distance	91
N-gram	88

This evaluation demonstrates the effectiveness of our Hybrid Algorithm in rectifying misspelled words, outperforming both the Edit Distance and N-gram approaches in terms of correction accuracy.

We also conducted another experiment where we deliberately introduced a higher number of intentional errors in the words. These errors involved inserting, deleting, substituting, or transposing more than two letters within the words.

The objective was to assess the performance and efficiency of all three approaches under such conditions.

The results indicated a decrease in efficiency and performance for all three approaches. The proposed algorithm achieved an accuracy of 87%, while the N-gram approach achieved 80% accuracy. The Edit Distance approach showed the lowest performance with an accuracy of 75%. The Table 6.3 below presents a comparative analysis of the accuracy with higher intentional errors.

Table 6.3: Accuracy Comparison with Higher Intentional Errors

Approach	Correction Accuracy (%)
Proposed Hybrid Algorithm	87
N-gram	80
Edit Distance	75

Additionally, we conducted experiments involving words with varying lengths, and the results revealed the comparative execution times of the Edit Distance, N-gram, and Hybrid algorithms. Figure 6.2.1 presents the comparison of execution times for different word lengths using these algorithms.

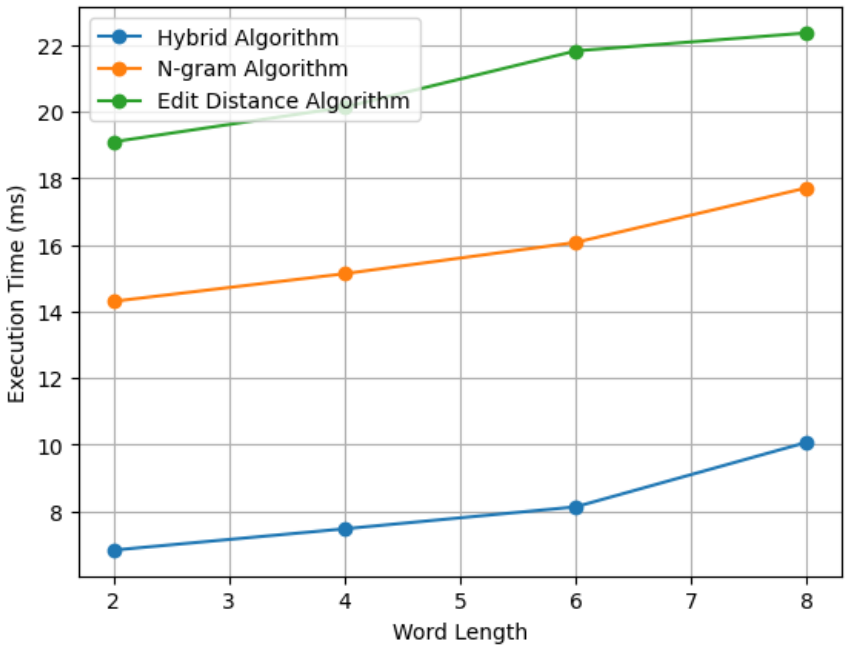


Figure 6.2.1: Execution Time Comparison for Different Word Lengths

6.2.2 Word Prediction

The two models proposed in the study were trained using different parameter configurations, as described in the previous Chapter 5. Within this thesis, training was conducted over 30, 60, and 100 epochs, resulting in respective training times of 6 and 4 hours for the LSTM and GRU models, using a learning rate of 0.001. The discrepancy in training duration between the two models is substantial. In terms of training accuracy, the LSTM model an accuracy of 0.8769, while the GRU model attained a slightly lower accuracy of 0.860.

Table 6.4 demonstrates the varying performance of networks with different parameters. The LSTM model, trained with a batch size of 32, took approximately 6 hours to complete. The proposed model achieved a good perplexity score of 0.931 on the validation dataset, indicating strong prediction performance. Additionally, the validation loss was significantly low for this model with a batch size of 64. Perplexity, being the inverse of instances where predictions match actual labels, decreases as the model's probabilities, including correct and incorrect ones, increase. Although the results were slightly less accurate than the batch size of 32, they were still promising. The training process resulted in a perplexity value of 1.132 and took around 5.4 hours to complete. The validation loss obtained was 0.783.

Table 6.4: Results Obtained Using Various Hyper-parameters

Model	**Perplexity(val)**	**Trained Time (hours)**	**Loss(val)**
LSTM-32	0.931	6	0.747
LSTM-64	1.132	5.4	0.783
GRU-32	1.11	4	0.723
GRU-64	1.341	3	0.739

Table 6.4 reveals that the perplexity for the GRU model with a batch size of 32 was 1.11, which is not significantly different from the LSTM model with the same batch size. Comparing the training times of both models is crucial, as users will train models on their own devices. With a batch size of 64, the GRU model achieved a perplexity of 1.341 on the validation dataset, with a training time of 3 hours, which was shorter than the LSTM model's training time for the same batch size. The validation loss for the GRU model was 0.739.

Table 6.5: Results with Different Parameters for LSTM

Learning Rate	Batch Size	Accuracy (val)	Epochs
0.001	**32**	**86.40**	**30**
0.0005	32	87.37	60
0.0001	32	85.78	100
0.001	**64**	**87.69**	**30**
0.0005	64	83.76	60
0.0001	64	82.11	100

Table 6.6: Results with Different Parameters for GRU

Learning Rate	Batch Size	Accuracy (val)	Epochs
0.001	**32**	**86.00**	**30**
0.0005	32	85.45	60
0.0001	32	86.24	100
0.001	**64**	**85.77**	**30**
0.0005	64	79.23	60
0.0001	64	81.01	100

The outcomes yielded by LSTM and GRU models under varying parameters are displayed in tables 6.5 and 6.6. The Figure 6.2.2 depicts a visual representation of the outcomes. The study was carried out to assess the influence of learning rate, batch size, and number of epochs on the precision of the models. The tabulated data offers valuable insights regarding the efficacy of the models across various parameter configurations. The values that are highlighted in the results indicate the highest level of accuracy that was attained by each model, with regards to particular batch sizes (32 and 64), over a range of epochs. The utilization of tables is imperative in comprehending the impacts of parameter selections on model efficacy and determining the most advantageous configuration.

Taking into account the timing of the predicted word or output within the user's input is also of great significance. Specifically, the LSTM model achieved a prediction time of 17 ms for a given input, while the GRU model exhibited a prediction time of 14 ms for inputs of the same length input.

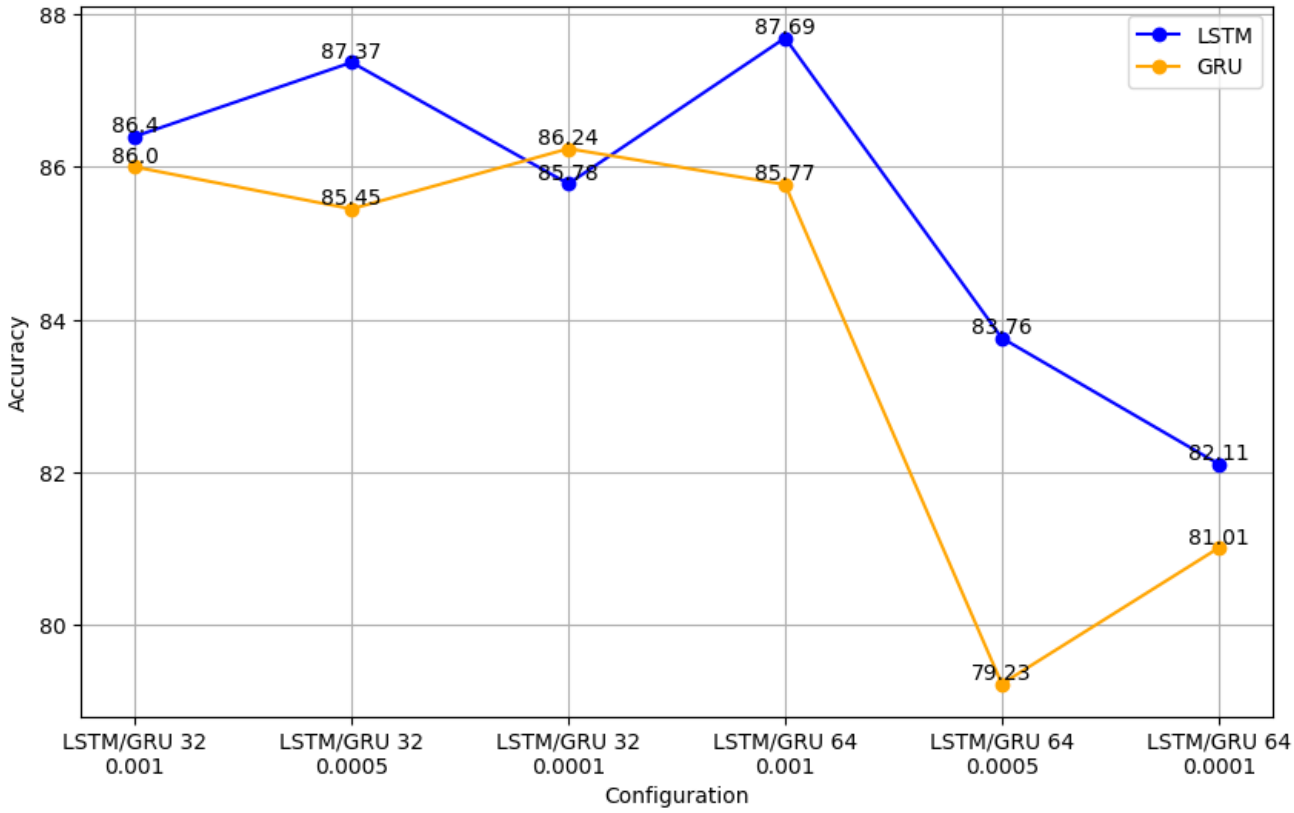


Figure 6.2.2: Results with different parameter settings

Chapter 7

System Design and Implementation

7.1 Development Environment and Frameworks

7.1.1 iOS Operating System

The development of the Kazakh-language keyboard system was carried out on the iOS operating system. The iOS platform provides a wide range of tools, application programming interfaces (APIs), and frameworks that facilitate the development of superior and intuitive applications. It ensures compatibility with various iOS devices, including iPhones and iPads, allowing for seamless deployment and distribution of the keyboard application.

7.1.2 Swift Programming Language

Swift is a modern, powerful, and intuitive programming language developed by Apple for various software development purposes. It combines the best features of multiple programming languages, making it easier to read, write, and maintain code [51]. Swift is designed to be safe, efficient, and expressive, enabling developers to create reliable and high-performance applications. We chose Swift as the programming language for developing the keyboard application.

7.1.3 Integration with CoreML and Xcode

Integrating trained language models and the auto-correction algorithm into an iOS application can be achieved using CoreML, Apple's machine learning framework. CoreML allows seamless integration of trained models into iOS apps, enabling on-device inference and efficient utilization. The models must be in the Core ML model format with the **.mlmodel** file extension for compatibility. To integrate our LSTM model for next word prediction, we follow steps such as training and optimizing the model, converting it to Core ML format (see Listing 10), and seamlessly integrating it into an iOS application using Xcode. This integration enhances the user's typing experience through Core ML's machine learning capabilities. Xcode, developed by Apple Inc., is a feature-rich integrated development environment (IDE) specifically designed for macOS and iOS application development. Table 7.1.3 below shows the main files used to implement the autocorrect and word prediction functions in XCode.

Table 7.1: Main files for auto-correction and word prediction

File	Description
Prediction.swift	This view controller handles user input and displays predicted next words. It communicates with the Core ML model trained for next word prediction.
Autocorrection.swift	The autocorrection.py file contains Python code that implements the auto-correction functionality using hybrid edit distance and n-gram algorithms. It is integrated into the Xcode project through PythonKit, enabling the execution of Python code within the iOS environment.

7.2 Kazakh-language Keyboard App

After completing the necessary steps for app development in Xcode and preparing the prediction model, we proceeded to create an application for testing purposes, focusing on the minimum viable product (MVP). The application's design at this stage is simple and intuitive, featuring a text entry field and a suggestion tool

positioned below it. This tool offers users word completion, prediction, and correction functionalities, significantly enhancing their typing experience. Users can seamlessly enter text and receive real-time suggestions based on our implemented algorithms and trained model, ensuring a smooth and efficient typing process. Figure 7.2.1 showcases an example of word completion, prediction, and correction features in our developed application.

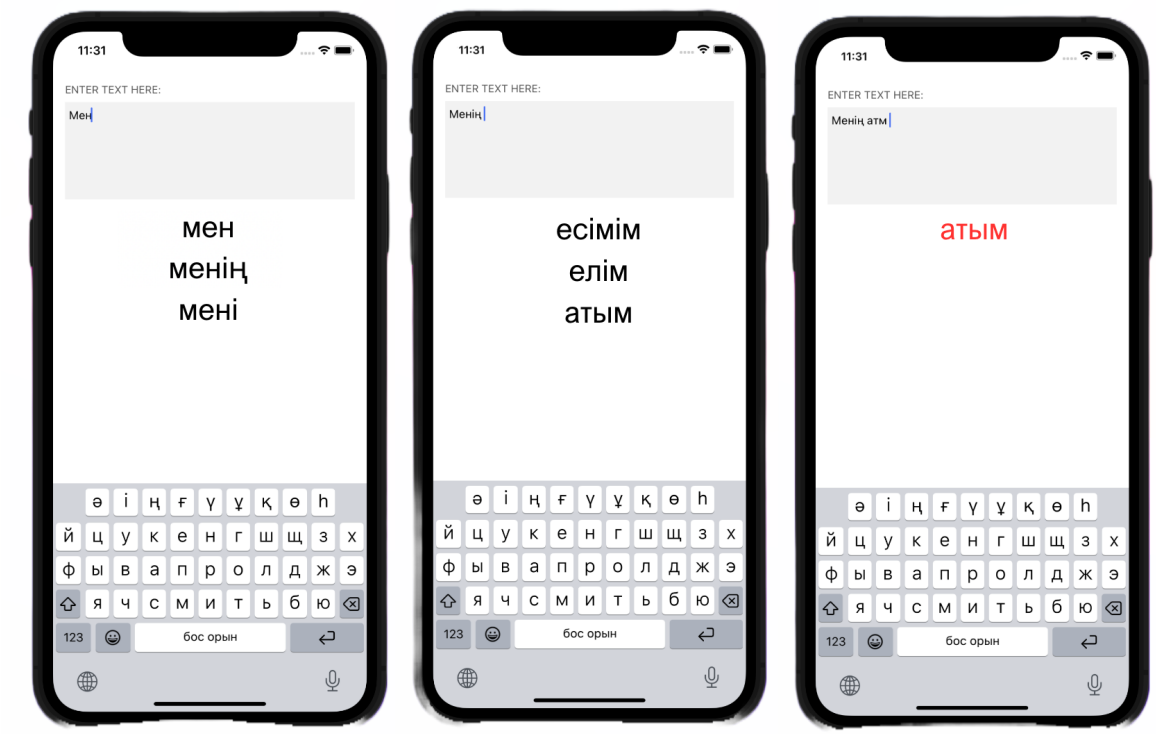


Figure 7.2.1: Completion, Prediction and Correction Example

Chapter 8

Discussion and Conclusion

This chapter presents a comprehensive summary of the research conducted, including the underlying methodologies employed, future directions for work, and potential avenues for improving the findings of this thesis. The focus lies on exploring ways to enhance the results of the experiments and build upon the work presented in this thesis.

In recent years, remarkable advancements have been witnessed in the fields of machine learning (ML) and natural language processing (NLP). The rapid progress can be attributed to the increased availability of computational power, enabling significant developments in deep learning and machine learning techniques. Language models have particularly benefited from the emergence of transformer models trained on massive amounts of data, leading to notable advancements in their capabilities. The advancements in ML and NLP, along with the utilization of advanced recurrent neural network models, have paved the way for substantial improvements in word prediction tasks.

The primary objective of this thesis is to develop a keyboard system equipped with auto-correction and word prediction functionalities. The focus of the study is centered around the implementation of a predictive model for suggesting the next word and the integration of an error correction mechanism. To achieve this, in-depth exploration and analysis of deep learning and machine learning approaches have been conducted. We investigated two sophisticated recurrent

neural network models for word prediction: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) and algorithms such as Edit Distance, N-gram based modeling and Hybrid approach have been extensively studied and examined to enhance the performance of the proposed error correction system.

To determine the best-performing model for our specific task, word prediction experiments were conducted using various combinations of parameters. The goal was to evaluate the performance of each model by employing the perplexity score as a performance metric, with Softmax as the activation function. When selecting the optimal model, factors such as accuracy, learning rate, and response time were taken into consideration.

For the auto-correction experiments, a comparative accuracy assessment was employed to evaluate the effectiveness of the three approaches. To conduct more comprehensive experiments, test data was generated with variations ranging from real words to non-words, including instances with one, two, or multiple letters changed. The accuracy metric was used to assess the performance of the correction mechanism.

The comparison between the two word prediction models revealed that their results were quite similar, except for one notable difference: the learning time. It was observed that LSTM required significantly more time compared to GRU. Additionally, it was noted that the model's performance also depended on the selection of hyperparameters, particularly certain values. Regarding the auto-correction mechanism, the experiments demonstrated that the hybrid approach achieved an accuracy of 95% for single-letter erroneous words, and an accuracy of 87% for words with two or more errors. Furthermore, it should be highlighted that the hybrid approach exhibited faster response times in providing correct word suggestions. These findings highlight the trade-off between learning time and model performance in word prediction tasks, as well as the effectiveness and efficiency of the hybrid approach in autocorrection. The results pave the way for further improvements and optimizations in the development of advanced language modeling techniques.

Future Research Directions

There are several potential avenues for future research and improvement in this area:

1. Dataset Augmentation: To enhance the program's performance, it could be beneficial to augment the user-specific dataset. This could involve incorporating more frequently used sentences from daily correspondence to improve the prediction of co-referencing words.

2. Federated Learning: Consider exploring the use of federated learning as an alternative to centralized machine learning. This approach eliminates the need for constant internet connectivity for predictions since the model resides on the device. Additionally, federated learning allows for more personalized predictions by leveraging the user's own model.

3. Handling Unknown Words: Currently, the model is limited to a specific dataset. To enhance its versatility, mechanisms can be developed to incorporate new words that are not part of the existing vocabulary. This adaptation process could involve adding new words to the vocabulary when encountered, making the model more generalized and adaptable.

4. Personalized Prediction: Consider incorporating personalized prediction capabilities based on the user's history. This could involve leveraging user-specific data to tailor the word predictions to individual preferences and writing styles.

These potential research directions provide opportunities for further advancements and enhancements in the field of word prediction and auto-correction systems.

Bibliography

[1] Zippia. 20 vital smartphone usage statistics [2023]: Facts, data, and trends on mobile use in the u.s., 2023. Accessed on Apr. 3, 2023.

[2] Maria Habib, Mohammad Faris, Raneem Qaddoura, Alaa Alomari, and Hossam Faris. A predictive text system for medical recommendations in telemedicine: a deep learning approach in the arabic context. IEEE Access, 9:85690–85708, 2021.

[3] Shuang Cai, Ahmet Palazoglu, Laibin Zhang, and Jinqiu Hu. Process alarm prediction using deep learning and word embedding methods. ISA transactions, 85:274–283, 2019.

[4] Sanidhya Mangal, Poorva Joshi, and Rahul Modak. Lstm vs. gru vs. bidirectional rnn for script generation. arXiv preprint arXiv:1908.04332, 2019.

[5] Omor Faruk Rakib, Shahinur Akter, Md Azim Khan, Amit Kumar Das, and Khan Mohammad Habibullah. Bangla word prediction and sentence completion using gru: an extended version of rnn on n-gram language model. In 2019 International Conference on Sustainable Technologies for Industry 4.0 (STI), pages 1–6. IEEE, 2019.

[6] Hozan K Hamarashid, Soran A Saeed, and Tarik A Rashid. Next word prediction based on the n-gram model for kurdish sorani and kurmanji. Neural Computing and Applications, 33:4547–4566, 2021.

[7] Jingyun Yang, Hengjun Wang, and Kexiang Guo. Natural language word prediction model based on multi-window convolution and residual network. IEEE Access, 8:188036–188043, 2020.

[8] Melanie Hüsser. Kännsch-swiss german keyboard for ios. PhD thesis, Bachelor's thesis, ETH Zurich (September 2015), 2015.

[9] Valentin Trifonov. *Kä*nnsch-updates and improvements to the swiss german keyboard. 2016.

[10] Andres Konrad. Kännsch-Adaptive Keyboard for iOS. PhD thesis, Bachelor's thesis, ETH Zurich (July 2016), 2016.

[11] R Prasad. Next word prediction and correction system using tensorflow, 2016.

[12] Aqil M. Azmi, Manal N. Almutery, and Hatim A. Aboalsamh. Real-word errors in arabic texts: A better algorithm for detection and correction. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 27 (8):1308–1320, 2019. doi: 10.1109/TASLP.2019.2918404.

[13] Amanjot Kaur, Paramjeet Singh, and Shaveta Rani. Spell checking and error correcting system for text paragraphs written in punjabi language using hybrid approach. International Journal Of Engineering And Computer Science, 3(9):8030–8032, 2014.

[14] Ratnasingam Sakuntharaj and Sinnathamby Mahesan. A novel hybrid approach to detect and correct spelling in tamil text. In 2016 IEEE international conference on information and automation for sustainability (ICIAfS), pages 1–6. IEEE, 2016.

[15] Amit Sharma and Pulkit Jain. Hindi spell checker. Indian Institute of Technology Kanpur, 2013.

[16] BURAK AYTAN and CEMAL OKAN ŞAKAR. Deep learning-based turkish spelling error detection with a multi-class false positive reduction model. Turkish Journal of Electrical Engineering and Computer Sciences, 31(3):581–595, 2023.

[17] Diana Rakhimova and Yntymak Abdrazakh. The task of identifying morphological errors of words in the kazakh language in social networks. In 2022 7th

International Conference on Computer Science and Engineering (UBMK), pages 344–349. IEEE, 2022.

[18] Yntymak Abdrazakh, Aliya Turganbayeva, and Diana Rakhimova. Development and study of an approach for determining incorrect words of the kazakh language in semi-structured data. In Advances in Computational Collective Intelligence: 13th International Conference, ICCCI 2021, Kallithea, Rhodes, Greece, September 29–October 1, 2021, Proceedings 13, pages 535–545. Springer, 2021.

[19] Anil K Jain. Data clustering: 50 years beyond k-means. Pattern recognition letters, 31(8):651–666, 2010.

[20] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT press, 2018.

[21] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. Advances in neural information processing systems, 13, 2000.

[22] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In Interspeech, volume 2, pages 1045–1048. Makuhari, 2010.

[23] Kiran Sharma, Ankit Naik, and Purushottam Patel. Study of artificial neural networks. International Journal of Advanced Research Trends in Engineering and Technology (IJARTET), 2(4):46–48, 2015.

[24] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Müller. Explaining deep neural networks and beyond: A review of methods and applications. Proceedings of the IEEE, 109(3):247–278, 2021.

[25] Jayanta Kumar Basu, Debnath Bhattacharyya, and Tai-hoon Kim. Use of artificial neural network in pattern recognition. International journal of software engineering and its applications, 4(2), 2010.

[26] Mohammad Galety, Firas Husham Al Mukthar, Rebaz Jamal Maaroof, and

Fanar Rofoo. Deep neural network concepts for classification using convolutional neural network: A systematic review and evaluation. 2021.

[27] Larry Medsker and Lakhmi C Jain. Recurrent neural networks: design and applications. CRC press, 1999.

[28] Larry R Medsker and LC Jain. Recurrent neural networks. Design and Applications, 5:64–67, 2001.

[29] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. Diploma, Technische Universität München, 91(1), 1991.

[30] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2):157–166, 1994.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[32] Alex Graves. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

[33] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555, 2014.

[34] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. Towards Data Sci, 6(12):310–316, 2017.

[35] Radford M Neal. Connectionist learning of belief networks. Artificial intelligence, 56(1):71–113, 1992.

[36] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[37] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Mar-

shall. Activation functions: Comparison of trends in practice and research for deep learning. arXiv preprint arXiv:1811.03378, 2018.

[38] Xiaojie Jin, Chunyan Xu, Jiashi Feng, Yunchao Wei, Junjun Xiong, and Shuicheng Yan. Deep learning with s-shaped rectified linear activation units. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 30, 2016.

[39] M Priya, R Kalpana, and T Srisupriya. Hybrid optimization algorithm using n gram based edit distance. In 2017 International Conference on Communication and Signal Processing (ICCSP), pages 0216–0221. IEEE, 2017.

[40] SM El Atawy and A Abd ElGhany. Automatic spelling correction based on n-gram model. International Journal of Computer Applications, 182(11): 0975–8887, 2018.

[41] Olzhas Makhambetov, Aibek Makazhanov, Zhandos Yessenbayev, Bakhyt Matkarimov, Islam Sabyrgaliyev, and Anuar Sharafudinov. Assembling the kazakh language corpus. In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 1022–1031, 2013.

[42] S Vijayarani, Ms J Ilamathi, Ms Nithya, et al. Preprocessing techniques for text mining-an overview. International Journal of Computer Science & Communication Networks, 5(1):7–16, 2015.

[43] Aditya Wiha Pradana and Mardhiya Hayaty. The effect of stemming and removal of stopwords on the accuracy of sentiment analysis on indonesian-language texts. Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control, pages 375–380, 2019.

[44] Claude E Shannon. Prediction and entropy of printed english. Bell system technical journal, 30(1):50–64, 1951.

[45] Farukh Iskalinov. A profession recommender system based on deep learning and machine learning approaches. Suleyman Demirel University Bulletin: Natural and Technical Sciences, 62(1):15–33, 2023. ISSN 2709-2631. doi: 10.47344/sdubnts.v62i1.946.

[46] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing, 415:295–316, 2020.

[47] D Randall Wilson and Tony R Martinez. The need for small learning rates on large problems. In IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222), volume 1, pages 115–119. IEEE, 2001.

[48] Jason Brownlee. What is the difference between a batch and an epoch in a neural network. Machine Learning Mastery, 20, 2018.

[49] Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. Compressing gradient optimizers via count-sketches. In International Conference on Machine Learning, pages 5946–5955. PMLR, 2019.

[50] Aejaz Farooq Ganai and Farida Khursheed. Predicting next word using rnn and lstm cells: Stastical language modeling. In *2*019 Fifth International Conference on Image Information Processing (ICIIP), pages 469–474. IEEE, 2019.

[51] Mikias Berhanu Gebre. Developing and ios application for value stream mapping with swift. 2016.

Appendix A

Listing 1: Data Cleainig example

```
import re
def extra_space(text):
    new_text= re.sub("\s+","␣",text)
    return new_text
def sp_charac(text):
    new_text=re.sub("[^rus_kaz_letters]", "" , text)
    return new_text
```

Listing 2: Removing stopwords example

```
import nltk
nltk.download('stopwords')
from wordcloud import WordCloud
from nltk.corpus import stopwords

stop_words = set(stopwords.words('kazakh'))
counter={}
for i in word_count.keys():
    if i not in list(stop_words):
        counter[i]=word_count[i]
print(len(counter.keys()))
```

Listing 3: Making errors for the given word

```
def find_wrong_word(sent, vocab):
    wrong_words = []
    sent = sent.strip().lower().split("␣")
    for word in sent:
        if word not in vocab:
            wrong_words.append(word)
    return wrong_words
```

Listing 4: Filtering misspleling words

```
import random
def make_errors(word):
    # Convert the word to a list for easier manipulation
    word_list = list(word)
    word_len = len(word_list)

    # Randomly choose an error to introduce
    error_type = random.randint(1, 8)

    if error_type == 1:  # Single letter insertion
        random_index = random.randint(0, word_len)
        random_letter = random.choice(rus_kaz_chars)
        word_list.insert(random_index, random_letter)

    elif error_type == 2:  # Single letter deletion
        if word_len > 1:
            random_index = random.randint(0, word_len - 1)
            del word_list[random_index]

    elif error_type == 3:  # Single letter substitution
        random_index = random.randint(0, word_len - 1)
        random_letter = random.choice(rus_kaz_chars)
        word_list[random_index] = random_letter

    elif error_type == 4:
        if word_len > 1:
            random_index = random.randint(0, word_len - 2)
            word_list[random_index], word_list[random_index + 1]
            = word_list[random_index + 1], word_list[random_index]

    elif error_type == 5:  # One character different
        random_index = random.randint(0, word_len - 1)
        random_letter = random.choice(rus_kaz_chars)
        while random_letter == word_list[random_index]:
            random_letter = random.choice(rus_kaz_chars)
        word_list[random_index] = random_letter

    elif error_type == 6:
        if random.random() < 0.5:
```

```
            random_index = random.randint(0, word_len)
            random_letter = random.choice(rus_kaz_chars)
            word_list.insert(random_index, random_letter)
        else:  # 50% chance to remove a letter
            if word_len > 1:
                random_index = random.randint(0, word_len - 1)
                del word_list[random_index]
        # Plus one letter different
        random_index = random.randint(0, len(word_list) - 1)
        random_letter = random.choice(rus_kaz_chars)
        while random_letter == word_list[random_index]:
            random_letter = random.choice(rus_kaz_chars)
        word_list[random_index] = random_letter

    elif error_type == 7:
        if word_len > 1:
            random_index = random.randint(0, word_len - 1)
            if random.random() < 0.5:
             if word_list[random_index] == word_list[random_index - 1]:
                del word_list[random_index]
            else:  # 50% chance to add a repeated character
        word_list.insert(random_index, word_list[random_index - 1])
        # Plus one character different
        random_index = random.randint(0, len(word_list) - 1)
        random_letter = random.choice(rus_kaz_chars)
        while random_letter == word_list[random_index]:
            random_letter = random.choice(rus_kaz_chars)
        word_list[random_index] = random_letter

    elif error_type == 8:
        if word_len > 2:
            random_index = random.randint(0, word_len - 3)
            word_list[random_index], word_list[random_index + 2]
            = word_list[random_index + 2], word_list[random_index]
        # Plus one character different
        random_index = random.randint(0, len(word_list) - 1)
        random_letter = random.choice(rus_kaz_chars)
        while random_letter == word_list[random_index]:
            random_letter = random.choice(rus_kaz_chars)
        word_list[random_index] = random_letter
```

```
    return ''.join(word_list)
```

Listing 5: Testing Edit Distance

```
def test_autocorrect(utdata, vocab, probs, string):
    tcount = 0
    fcount = 0
    rcount = 0
    print("Running "+string+" : Basic Auto-correct system")
    for k, v in utdata.items():
        incorrect_list = v.strip().split()
        #print(incorrect_list)
        for w in incorrect_list:
            tcount = tcount + 1
            cw = get_correct_word(w, vocab, probs, 25)
            if cw==k:
                #print('correct')
                rcount = rcount + 1
            else:
                #print('wrong')
                fcount = fcount + 1
    print("Accuracy : {} %".format((rcount/tcount)*100))
```

Listing 6: N-gram based Appoach

```
def test_autocorrect_bigram_min_edit(
                    utdata,
                    vocab,
                    probs,
                    string,
                    bigram_count):
    tcount = 0
    fcount = 0
    rcount = 0
    print("Running "+string+" : Bi-gram Auto-correct system")
    for k, v in utdata.items():
        incorrect_list = v.strip().split()
        #print(incorrect_list)
        for w in incorrect_list:
            tcount = tcount + 1
            cw = get_correct_word_bigram(w,
```

```
            '<s>', probs, vocab, bigram_counts, 0.3, 0.7, 25)
            if cw==k:
                #print('correct')
                rcount = rcount + 1
            else:
                #print('wrong')
                fcount = fcount + 1
    print("Accuracy : {} %".format((rcount/tcount)*100))
```

Listing 7: Hybrid Approach

```
def test_autocorrect_bigram_min_edit(
                    utdata,
                    vocab,
                    probs,
                    string,
                    bigram_count):
    tcount = 0
    fcount = 0
    rcount = 0
    print("Running "+string+" :
    Bi-gram (min-edit) Auto-correct system"
    )
    for k, v in utdata.items():
        incorrect_list = v.strip().split()
        #print(incorrect_list)
        for w in incorrect_list:
            tcount = tcount + 1
            cw = get_correct_word_bigram_min_edit(w, '<s>',
            probs, vocab, bigram_counts, 0.3, 0.7, 25, 0.000001)
            if cw==k:
                #print('correct')
                rcount = rcount + 1
            else:
                #print('wrong')
                fcount = fcount + 1
    print("Accuracy : {} %".format((rcount/tcount)*100))
```

Appendix B

Listing 8: Creating LSTM Model

```
def lstm_model(length,unit1,unit2,n):
    import matplotlib.pyplot as plt
    from tensorflow.keras.regularizers import l2
    from tensorflow.keras.layers import LSTM, Activation,
    Dropout, Dense, Input,Embedding,Bidirectional
    from tensorflow.keras.callbacks import ModelCheckpoint,
    ReduceLROnPlateau, LearningRateScheduler
    from tensorflow.keras.models import Model,Sequential
    from tensorflow.keras.optimizers import Adam

    # Calling the encoding function to get
    # the data of specified length and the vocabulary
    data_x,data_y,v,wti=encoding_data(length)
    print("Data Encoded")
    print("Data_x",data_x[:5])
    print("Data_y",data_y[:5])
    print("Vocab_Size",v)

    # Preparing the model based on
    # the inputs of unit1 ,unit2 and vocab values
    model = Sequential()
    model.add(Embedding(v, length-1, input_length=length-1))
    model.add(Bidirectional(LSTM(unit1, return_sequences=True)))
    model.add(LSTM(unit2))
    model.add(Dropout(0.2))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(v, activation='softmax'))
    print(model.summary())
    model.compile(
```

```
    loss='categorical_crossentropy',
    optimizer=Adam(lr=0.001),
    metrics=['accuracy', 'perplexity'])
filepath="news_lstm_len"+str(length)+".hdf5"
checkpoint = ModelCheckpoint(
        filepath, monitor='loss',
        verbose=1,
        save_best_only=True,
        mode='min')

def scheduler(epoch):

    if epoch < 60:
        return 0.001
    elif epoch < 100:
        return 0.0005
    else:
        return 0.0001
lr = LearningRateScheduler(scheduler)
callbacks_list = [checkpoint,lr]
history=model.fit(
        data_x, data_y,
        batch_size=64,
        epochs=n,
        callbacks=callbacks_list)
del data_x,data_y,v,wti
```

Listing 9: Creating GRU Model

```
def gru_model(length,unit1,unit2,n):
import matplotlib.pyplot as plt
from tensorflow.keras.regularizers import l2
from tensorflow.keras.layers import LSTM, Activation,
Dropout, Dense, Input,Embedding,Bidirectional
from tensorflow.keras.callbacks import ModelCheckpoint,
ReduceLROnPlateau, LearningRateScheduler
from tensorflow.keras.models import Model,Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import GRU
from tensorflow import keras
```

```
# Calling the encoding function to get the data
# of specified length and the vocabulary
data_x,data_y,v,wti=encoding_data(length)
print("Data Encoded")
print("Data_x",data_x[:5])
print("Data_y",data_y[:5])
print("Vocab_Size",v)

model = Sequential()
model.add(Embedding(v, length-1, input_length=length-1))
model.add(GRU(unit1, return_sequences=True))
model.add(GRU(unit2))
model.add(Dropout(0.2))
model.add(Dense(50, activation='relu'))
model.add(Dense(v, activation='softmax'))
print(model.summary())
model.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(lr=0.001),
    metrics=['accuracy', 'perplexity'])
filepath="gru_news_len"+str(length)+".hdf5"
checkpoint = ModelCheckpoint(
    filepath, monitor='loss',
    verbose=1, save_best_only=True,
    mode='min')

def scheduler(epoch):
    if epoch < 60:
        return 0.001
    elif epoch < 100:
        return 0.0005
    else:
        return 0.0001
lr = LearningRateScheduler(scheduler)
callbacks_list = [checkpoint,lr]
history=model.fit(data_x, data_y,
batch_size=64, epochs=n,
callbacks=callbacks_list)
```

Listing 10: Exporting .mLModel

```
import coremltools import tensorflow as tf
# Load and compile the trained LSTM model
Istm model = tf.keras.models.
load model( 'trained_Istm_model.h5')
1stm_model.compile (
loss= 'categorical_ _crossentropy',
optimizer='adam')
# Set the desired number of predictions to 3
num predictions = 3
Istm model.layers [-1].output shape = (None,
num predictions)
# Convert the model to Core ML format
corem1_ model =
coremItools.converters.keras.convert(lstmmodel)
# Save the Core ML model
coreml _model.save ('Istm_next_ _word_prediction.mImodel')
```