Comparative Analysis of Transformer-Based Models for Text-To-Speech Normalization

Pankti Dholakia
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
DOI: https://doi.org/10.31979/etd.5dd6-k38w
https://scholarworks.sjsu.edu/etd_projects/1242
Comparative Analysis of Transformer-Based Models for Text-To-Speech Normalization

A Project Report
Presented to
The Faculty of Department of Computer Science
San José State University

In Partial Fulfillment
Of the Requirements for Degree
Master of Computer Science

by
Pankti Dholakia
May 2023
The Designated Project Committee Approves the Master’s Project Titled

COMPARATIVE ANALYSIS OF TRANSFORMER-BASED MODELS FOR TEXT-TO-SPEECH NORMALIZATION

by

Pankti Dholakia

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2023

Dr. Ching-Seh Wu, Ph.D 
Department of Computer Science

Dr. Genya Ishigaki, Ph.D 
Department of Computer Science

Dr. Fabio Di Troia, Ph.D 
Department of Computer Science
ABSTRACT

COMPARATIVE ANALYSIS OF TRANSFORMER-BASED MODELS FOR TEXT-TO-SPEECH NORMALIZATION

by Pankti Dholakia

Text-to-Speech (TTS) normalization is an essential component of natural language processing (NLP) that plays a crucial role in the production of natural-sounding synthesized speech. However, there are limitations to the TTS normalization procedure. Lengthy input sequences and variations in spoken language can present difficulties. The motivation behind this research is to address the challenges associated with TTS normalization by evaluating and comparing the performance of various models. The aim is to determine their effectiveness in handling language variations. The models include LSTM-GRU, Transformer, GCN-Transformer, GCNN-Transformer, Reformer, and a BERT language model that has been pre-trained. The research evaluates the performance of these models using a variety of metrics, including accuracy, loss, word error rate, and sentence error rate. For the primary experiments, Google's TTS Wikipedia dataset was used. In order to evaluate the efficacy of TTS normalization models on inconsistent language, such as slang, this research paper produces a relatively small Twitter dataset. The dataset was manually annotated to provide the models with additional evaluation metrics. The inclusion of this dataset offers further insights into the models' effectiveness in handling variations in language. The results of this study demonstrated that the Reformer model with BERT tokenizer achieved the highest accuracy on both datasets, while the Reformer model with BPE tokenizer had low word and sentence error rates and performed better on longer input sequences. The GCN-Transformer and GCNN-Transformer models also performed well, with the GCNN-Transformer outperforming its counterpart and RNN implementations. We observed that although the BERT model had the advantage of pre-training, the Reformer model could compete with an accuracy of 96% without pre-trained data. These findings highlight the significance of precise TTS normalization models for natural language generation and human-computer interaction. Our study contributes to the ongoing effort to enhance TTS normalization models.

Keywords – Natural Language Processing, Text-to-Speech, Normalization, Machine Learning, Transformer, Language Models, Pre-trained BERT, Human-Computer Interaction
ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my thesis advisor, Dr. Mike Wu, for his guidance, support, and expertise throughout my research. His valuable insights and feedback have been instrumental in shaping this study. I would also like to thank my committee members, Dr. Genya Ishigaki and Dr. Fabio Di Troia, for their valuable suggestions and critical evaluation of my work. Their expertise in machine learning and database technologies have been invaluable in improving the quality of my research.

I would also like to express my appreciation to my friends and family for their encouragement and support throughout this journey. Their unwavering support has been a source of motivation and inspiration.

Lastly, I would like to acknowledge the invaluable contributions of all the researchers and experts in the field of natural language processing. Their groundbreaking research has paved the way for this study, and I hope that my work can contribute to advancing the development of TTS normalization models and improving human-computer interaction.
TABLE OF CONTENTS

1. INTRODUCTION ................................................................................................................................................... 12
  1.1 Natural Language Generation ................................................................................................................................. 12
  1.2 Text-to-Speech Normalization ................................................................................................................................ 12
  1.3 Problem Statement .................................................................................................................................................. 13
  1.4 Research Objective ................................................................................................................................................. 13

2. BACKGROUND STUDY ....................................................................................................................................... 15
  2.1 Challenges in text normalization for text-to-speech ....................................................................................... 16
    2.1.1 Proposed Solutions for Text Normalization ............................................................................................... 16
    2.1.2 Neural Probabilistic Language Model ........................................................................................................... 16
  2.2 Neural Network Architectures ........................................................................................................................ 17
    2.2.1 Recurrent Neural Network and Long Short-Term Memory ....................................................................... 17
    2.2.2 Convolutional Neural Networks and Gated CNN ...................................................................................... 17
    2.2.3 Hybrid Bidirectional LSTM ....................................................................................................................... 19
    2.2.4 Reformer: The Efficient Transformer ........................................................................................................ 20
  2.3 Transfer Learning ........................................................................................................................................... 21
    2.3.1 BERT Pre-Trained ..................................................................................................................................... 21
    2.3.2 Parameter Efficient Transfer Learning ...................................................................................................... 21
    2.3.3 Transfer Learning with a Unified Test-to-Text Transformer ..................................................................... 22

3. METHODOLOGY ................................................................................................................................................... 23
  3.1 Technical Approach ........................................................................................................................................ 23
    3.1.1 Model Implementation and Training ............................................................................................................... 23
    3.1.2 Convert Predicted Output to Speech ............................................................................................................... 23
  3.2 Development Platform .................................................................................................................................... 24
  3.3 Python Modules .............................................................................................................................................. 24

4. DATASET ............................................................................................................................................................... 25
  4.1 Exploratory Data Analysis ...................................................................................................................................... 26
  4.2 Data Preparation ................................................................................................................................................. 29
    4.2.1 Data Preparation for LSTM-GRU Model ........................................................................................................ 29
    4.2.2 Data Preparation for Transformer and Reformer Models ............................................................................... 30

5. IMPLEMENTATION .............................................................................................................................................. 32
COMPARATIVE ANALYSIS OF TRANSFORMER-BASED MODELS FOR TEXT-TO-SPEECH NORMALIZATION

Pankti Dholakia

5.1 LSTM-GRU Model ................................................................................................................................................. 32
  5.1.1 Long Short-Term Memory (LSTM) ................................................................................................................ 32
  5.1.2 Gated Recurrent Unit (GRU) ....................................................................................................................... 33
  5.1.3 Model Architecture ......................................................................................................................................... 34

5.2 Transformer – GCN and GCNN ............................................................................................................................. 34
  5.2.1 Transformer ..................................................................................................................................................... 34
  5.2.2 Graph Convolutional Network (GCN) ......................................................................................................... 38
  5.2.3 Gated Convolutional Neural Network (GCNN) .......................................................................................... 39
  5.2.4 Model Architectures ......................................................................................................................................... 40

5.3 Reformer Language Model ................................................................................................................................... 42
  5.3.1 Locality Sensitive Hashing (LSH) ................................................................................................................ 42
  5.3.2 Model Architecture ......................................................................................................................................... 43
  5.3.3 Language Model Head .................................................................................................................................. 44

5.4 Transfer Learning with BERT ............................................................................................................................. 45
  5.4.1 BERT – Bidirectional Encoder Representations ......................................................................................... 45
  5.4.2 GPT-2 ............................................................................................................................................................. 46
  5.4.3 Model Architecture ....................................................................................................................................... 46

6. METRICS AND EXPERIMENTS ............................................................................................................................ 48
  6.1 Metrics ................................................................................................................................................................. 48
  6.2 Experiments ......................................................................................................................................................... 48
    6.2.1 LSTM-GRU for Classification .................................................................................................................... 48
    6.2.2 LSTM-GRU for seq2seq model .................................................................................................................. 50
    6.2.3 GCN-Transformer ....................................................................................................................................... 51
    6.2.4 GCNN-Transformer .................................................................................................................................. 52
    6.2.5 Reformer ....................................................................................................................................................... 53
    6.2.6 Transfer Learning with BERT .................................................................................................................... 56
    6.2.7 Output to Audio Conversion ...................................................................................................................... 57

7. RESULTS AND ANALYSIS .................................................................................................................................... 59
  7.1 Model Comparison .............................................................................................................................................. 59
    7.1.1 Result with Wikipedia Dataset .................................................................................................................. 59
    7.1.2 Result with Twitter Dataset ....................................................................................................................... 59

7.2 Model Analysis ...................................................................................................................................................... 60
  7.2.1 Advantages of Gated CNN .......................................................................................................................... 60
  7.2.2 Advantages of Reformer .................................................................................................................................. 61
8. CONCLUSION AND FUTURE WORK ................................................................................................................. 62

8.1 Challenges and future works .................................................................................................................. 62
LIST OF TABLES

Table 1: Challenges in TTS Summary................................................................................................................... 17
Table 2: Existing TTS Approaches........................................................................................................................ 21
Table 3: Number of Tokens [12] ........................................................................................................................... 26
Table 4: GCN-Transformer Architecture .............................................................................................................. 40
Table 5: GCNN -Transformer Architecture ........................................................................................................... 41
Table 6: BERT Model Architecture ...................................................................................................................... 47
Table 7: GCNN vs GCN-Transformer Results ...................................................................................................... 53
Table 8: Reformer Results ..................................................................................................................................... 56
Table 9: BERT Model Results ............................................................................................................................... 57
Table 10: Results with Wikipedia Dataset ............................................................................................................. 59
Table 11: Results with Twitter Dataset .................................................................................................................. 60
LIST OF FIGURES

Figure 1: Background Study Organization ............................................................................................................ 15
Figure 2: GCNN Architecture [15] ....................................................................................................................... 18
Figure 3: Neural Network Architecture using GCNN for NLP [16] ........................................................................... 19
Figure 4: LSH in Reformer [18] ............................................................................................................................. 20
Figure 5: Reversible Residual Layers [7] .................................................................................................................. 20
Figure 6: Deep Learning Model Training ............................................................................................................... 23
Figure 7: Convert Output to Audio ........................................................................................................................ 23
Figure 8: Original Data File ................................................................................................................................... 25
Figure 9: Number of tokens per class ..................................................................................................................... 26
Figure 10: Class v/s Frequency graph in the dataset ............................................................................................. 27
Figure 11: Maximum Change Tokens ................................................................................................................... 27
Figure 12: Maximum Change Classes ................................................................................................................... 28
Figure 13: Input Sequences (Sentences) .................................................................................................................. 28
Figure 14: Tokenized Twitter Data File ................................................................................................................ 29
Figure 15: Label Encoding .................................................................................................................................... 29
Figure 16: Padding Sequences ............................................................................................................................... 30
Figure 17: Embedding Matrix ............................................................................................................................... 30
Figure 18: Fill NaN Values .................................................................................................................................... 31
Figure 19: Pad Sequences ...................................................................................................................................... 31
Figure 20: Encode Train and Validation Dataset ................................................................................................... 31
Figure 21: LSTM Architecture [25] ...................................................................................................................... 32
Figure 22: GRU Architecture [26] ........................................................................................................................ 33
Figure 23: LSTM-GRU Model Architecture ......................................................................................................... 34
Figure 24: Computing Angle Rates ....................................................................................................................... 35
Figure 25: Apply Positional Encoding to Input ..................................................................................................... 36
Figure 26: Positional Encoding Visualization [26] .............................................................................................. 36
Figure 27: Self-Attention Formula [2] .................................................................................................................... 37
Figure 28: Compute Multiple Heads [2] ................................................................................................................ 37
Figure 29: Multi-Headed Attention [2] ................................................................................................................... 37
Figure 30: Encoder [2] ........................................................................................................................................... 38
Figure 31: Convolution v/s Graph Convolution [28] ............................................................................................. 39
Figure 32: Adjacency Matrix [28] ........................................................................................................................ 39
Figure 33: Local-Sensitivity Hashing [28] .............................................................................................................. 43
Figure 34: Reformer Model Configuration ............................................................................................................ 44
Figure 35: BERT Architecture [30] ....................................................................................................................... 46
Figure 36: Word Error Rate ................................................................................................................................... 48
Figure 37: LSTM-GRU Classification Input ......................................................................................................... 49
Figure 38: LSTM-GRU Accuracy and Loss .......................................................................................................... 49
Figure 39: LSTM-GRU WER and SER ................................................................................................................ 50
Figure 40: Convert Sequences to Graph ................................................................................................................ 51
Figure 41: Sequence Graph ................................................................................................................................... 51
Figure 42: GCN-Transformer Accuracy and Loss ................................................................................................ 52
Figure 43: GCN-Transformer WER and SER ....................................................................................................... 52
Figure 44: GCNN-Transformer Accuracy and Loss .............................................................................................. 53
Figure 45: GCNN-Transformer WER and SER .................................................................................................... 53
Figure 46: BPE-Tokenizer ..................................................................................................................................... 54
Figure 47: BERT Pre-trained Tokenizer ................................................................................................................ 55
Figure 48: Reformer Accuracy and Loss ............................................................................................................... 56
Figure 49: BERT Accuracy, WER and SER ......................................................................................................... 57
Figure 50: Audio Conversion Process ................................................................................................................... 58
Figure 51: Sample Audio File ................................................................................................................................ 58
1. INTRODUCTION

1.1 Natural Language Generation

Natural Language Processing (NLP) is a field of computer science that focuses on the interaction between computers and human language. One branch of NLP, Natural Language Generation (NLG), involves the creation of natural-sounding language by machines. NLG uses computer algorithms to generate text, speech, or other forms of natural language output based on a set of rules or data inputs. The goal of NLG is to create language output that is indistinguishable from human-generated language in terms of style, tone, and content. NLG has various applications across multiple industries, including e-commerce, healthcare, finance, and entertainment.

NLG generates language that sounds natural using language models such as LSTM, RNN, and transformers. To understand the patterns and structure of natural language, these models are trained on massive datasets of human-generated text. It frequently employs LSTM (Long Short-Term Memory) and RNN (Recurrent Neural Network) models to generate text and speech by predicting the next word or phoneme in a sequence. These models are especially effective at generating text or speech with a predictable structure, such as product descriptions or weather forecasts. Transformers, on the other hand, are more effective at encoding long-range dependencies in language and, as a result, are better suited to generating more complex and nuanced language output. It has been demonstrated that transformers are especially effective at generating text and speech that is indistinguishable from language generated by humans.

1.2 Text-to-Speech Normalization

Normalization of Text-to-Speech (TTS) is a crucial challenge in natural language processing (NLP). It entails converting written text into synthesized, natural-sounding speech. The goal of TTS normalization is to create high-quality synthesized speech that is indistinguishable from human speech in terms of tone, intonation, and pronunciation. Normalization of TTS is essential because it has multiple practical applications, including accessibility tools for the visually impaired, language learning, and entertainment. It is also a crucial component of human-computer interaction, allowing users to interact with machines via spoken language. However, TTS normalization presents several obstacles that must be overcome in order to produce high-quality synthesized speech. Dealing with variations in language, such as dates, addresses, and vernacular, is a significant obstacle. As an example, $123 must be normalized to "one hundred and twenty-three dollars" [1]. Another difficulty is dealing with lengthy input sequences, which can lead to information loss and an unnatural-sounding speech synthesizer.
1.3 Problem Statement

Text-to-Speech (TTS) normalization is a critical challenge in natural language processing (NLP) that involves transforming written text into speech that sounds natural. The normalization process of TTS aims to produce high-quality synthesized speech that is natural in terms of tone, intonation, and pronunciation. TTS normalization faces several limitations as previously mentioned like - lengthy input sequences, the handling of variations in spoken language, and the accurate prediction of the pronunciation of homophones and words with multiple possible pronunciations. In this study, we propose to evaluate sophisticated machine learning models for TTS normalization in order to address these issues. This research specifically proposes using state-of-the-art models such as the Transformer, Reformer, and BERT models, which have demonstrated promising results in handling complex language variants and lengthy input sequences.

This research uses the TTS Wikipedia dataset to evaluate the performance of these models, as well as a Twitter dataset generated with manual annotation to evaluate the models' performance on inconsistent speech, such as slang. This study evaluates the efficacy of the model’s using metrics like but not limited to accuracy, loss, word error rate, and sentence error rate. The metrics chosen to evaluate for language models differ from the popular metrics of precision, recall and F1 scores as they reflect the efficacy of model on classification tasks, for natural language generation model word and sentence error rates provide a better assessment.

1.4 Research Objective

The primary goal of this study is to compare the natural language generation capability of different language models for text-to-speech normalization and the effect of replacing feed-forward layers of Encoder architectures with GCN and GCNN to evaluate the efficiency. It also aims to assess the difference in the output quality depending on varying lengths of input sequences and compare more sophisticated language models like Reformers and pre-trained BERT.

This study focuses on the following objectives:
• Perform a comparative analysis to evaluate the performance of existing approaches for TTS normalization using multiple datasets.
• Asses the difference of self-attention and multi-headed attention layers within Transformer-based models.
• Examining LSTM-GRU, GCN-Transformer, GCNN-Transformer, Reformer, and pre-trained BERT language models.
• Analyze the performance of LSTM-GRU models for classification versus seq2seq application like TTS normalization.
• Investigate the impact of various hyperparameters, including self-attention, the number of stacked layers in encoders, dropout rates, and activation functions, on model performance through a series of tests.
This research project provides a comprehensive evaluation of text-to-speech (TTS) normalization models and their related applications. This report will be organized as follows. Chapter 2 provides an overview of existing work and models utilized in TTS normalization, while Chapter 3 covers the development platform, libraries, and technical approach adopted in this study. Data selection, preprocessing, and details about the dataset will be discussed in Chapter 4. Chapter 5 covers the approach, the experimental model designs, and their implementation. The experiments with different models, their results and discussion will be covered in Chapters 6 and 7 with the Conclusion and the scope of future work in Chapter 8.
2. BACKGROUND STUDY

The Natural Language Processing (NLP) is a discipline with extensive research and numerous challenges, such as text normalization for text-to-speech synthesis using deep learning models. Due to their superior performance relative to other models, Recurrent Neural Networks (RNNs), Long Short-Term Memory, and, more recently, Gated Convolutional Neural Networks (CNNs) are widely used in model design for NLP applications. Recently, the study by A. Vasvani et al. [2] titled "Attention is All you Need" introduced Transformers as a new architecture, which has been investigated as a potentially more effective replacement for existing neural networks. As a result, performance enhancements have been observed in the field of text normalization. This research will examine models such as LSTMs, Gated CNNs, and Transformers, among others to provide an exhaustive overview of the models and architectures used for text-to-speech normalization. The articles and papers included in this literature review are recent NLP techniques that are used for the TTS normalization as well as studies the problems existing in text normalization. Section I of this review discusses the current challenges. Section II discusses existing deep learning approaches for TTS used to address these challenges. Section III discusses the existing transfer learning techniques that have been employed in tandem with deep learning models. The Figure 1. below illustrates the organization of this background study.

Figure 1: Background Study Organization
2.1 Challenges in text normalization for text-to-speech

One of the most significant challenges in text normalization or TTS normalization is to determine the appropriate approaches for handling non-standard English language [3]. This encompasses data from social media and messaging platforms, such as Twitter, which utilize a range of acronyms, slang, and emoticons [4]. It is crucial to identify models that can generalize well on both standard and non-standard forms of English language. Numerous approaches and architectures have been proposed to address these challenges.

2.1.1 Proposed Solutions for Text Normalization

Many studies have addressed the difficulty of normalizing text for text-to-speech synthesis using deep learning models. Max Kaufman proposed a two-step method combining statistical machine translation with a neural network-based pre-processor to eliminate noise from Twitter posts and convert them to standard English [5]. Ranjan Satapathy et al. compared four different deep learning encoder-decoder frameworks for microtext normalization, a crucial task for natural language processing (NLP) and data mining, and discovered that LSTM, CNN, and GRU layers with the CMUdict dataset produced the best results [6]. Deane Pennel and Yang Liu created a data dictionary to convert standard English to textual slang and vice versa [7]. Using a hybrid bi-directional implementation of an LSTM with both word and character level training for known and unknown words, respectively, Sunil Kumar et al. achieved effective results [8]. Due to the lack of data for text-to-speech normalization experimentation, researchers must manually generate training rules and dictionaries. To enhance the existing model architectures, it is necessary to generate a large amount of data from a variety of sources.

2.1.2 Neural Probabilistic Language Model

The challenge of the curse of dimensionality makes training models for word sequence assessment inherently difficult. Traditional n-gram-based methods have been effective in achieving generalization by concatenating extremely brief overlapping sequences from the training set [9]. However, to combat this challenge, a method has been proposed that learns a distributed representation for words, allowing each training phrase to educate the model about an exponential number of semantically surrounding sentences. This approach achieves better performance than state-of-the-art n-gram models by simultaneously learning the probability function for word sequences and the distributed representation for each word [10]. The joint probability function of word sequences is expressed in terms of distributed word feature vectors assigned to each word in the vocabulary. Experiments have shown that this method significantly outperforms state-of-the-art methods in terms of perplexity, with differences between 10 and 20%. This paper provides a valuable reference for reducing dimensionality in the training space while designing an optimal model structure [10].
Table 1 summarizes the results achieved by various researchers in identifying the key challenges in TTS normalization and provides suggestions for different approaches to overcome them.

<table>
<thead>
<tr>
<th>Study</th>
<th>Approach</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Kaufman [5]</td>
<td>Two-step approach combining statistical machine translation with a pre-processor.</td>
<td>Significantly increased readability of Twitter posts converted into English</td>
</tr>
<tr>
<td>Ranjan Satapathy et al. [6]</td>
<td>Compared four different deep learning encoder-decoder frameworks.</td>
<td>Combination of LSTM, CNN, and GRU yielded 80% accuracy.</td>
</tr>
<tr>
<td>Sunil Kuma et al. [8]</td>
<td>Hybrid bi-directional implementation of an LSTM</td>
<td>Achieved efficient results</td>
</tr>
<tr>
<td>Overall</td>
<td>Lack of data for experimentation in text-to-speech normalization, leading to manual generation of rules and dictionaries for training</td>
<td>Generating a large corpus of data from varied sources is necessary to improve existing model architectures</td>
</tr>
</tbody>
</table>

2.2 Neural Network Architectures

2.2.1 Recurrent Neural Network and Long Short-Term Memory

Recurrent neural networks (RNNs) with attention layers have been widely used in language models, speech recognition, and prediction tasks like Named-entity recognition (NER) or Part of Speech (POS) tagging. Sproat and Jaitly in their research [11] suggest that RNNs are suitable for text normalization, which is further validated by Maryam Zare and Shourya Rohatgi in their development of the DeepNorm framework [12]. The framework uses a classification algorithm for semiotic classes, followed by a Seq2Seq RNN model to predict normalized text using regular expression functions. However, the study suggests that RNNs and LSTMs can be replaced by better encoder-decoder architectures like transformers for improved performance. In [11], the researchers present a challenge to solve text normalization using RNN-based architectures, along with a dataset created by passing Wikipedia data through Google's Kestrel for text normalization, and manual annotation of about 1000 English tokens. The paper implements an RNN model with attention layers, which overfits the data, leading to high accuracy. However, the main purpose of the paper is to understand the dataset and generate a similar one using Twitter data.

2.2.2 Convolutional Neural Networks and Gated CNN

A convolution is a window that slides over a larger set of input data with a focus on a subset of the input matrix, which serves as the foundation of the CNN layer. A convolution sweeps the window through images and then calculates the pixel values of the input and filter dot product. This enables convolution to highlight the pertinent features [13, 14]. While CNNs are commonly used for image classification and recognition, different ConvNets such as the Gated Convolutional Neural Network (GCN) [13] have demonstrated promise in the field of natural language processing. This is since RNNs are extremely sluggish to train, and as a result, much of the focus over the past few years has been on
developing new architectures to speed them up. RNNs process the input sequentially, with each word arriving one at a time; in contrast, all elements in a CNN are processed simultaneously, which can significantly accelerate processing.

Research indicates that a hierarchical CNN framework is significantly more effective than the conventional RNN seq2seq models used in earlier works [13, 14]. RNNs have low-efficiency issues because their training relies on previous steps. Therefore, the authors propose a seq2seq model based on CNN to generate source text representations [14]. CNNs with multiple layers produce hierarchical representations of the input. In contrast to the chain structure of RNN, this multi-layer CNN provides a shortcut for expressing long-range sequences in parallel. The implemented model employs Convolutional architecture with a copying mechanism. Copying mechanism assigns a weight to each word in the original document to evaluate the word's importance using positional attention score. The proposed model can represent a sequence with a hierarchical structure, which greatly improves its effectiveness. A copying mechanism is also implemented to further enhance the performance of the model, as it can handle rare words [14]. While [13] only addresses the problem of context summarization, their approach is relevant to our research problem of text summarization, where we can implement hierarchical CNN instead of RNN prior to introducing transformer layers.

In their ground-breaking research, Dauphin et al. [15] introduced a convolutional-based language model that produces results comparable to RNNs, but with minimal training costs due to the parallelization strategy of stacked GCNN layers. The authors develop a finite context approach using stacked convolutions, which can be more efficient than sequential tokens because they permit parallelization. They develop and apply new gated convolutional networks to language modeling. Stacking convolutional networks allows for the representation of large context sizes and the extraction of hierarchical features. Analyzing the input hierarchically resembles classical grammar formalisms that construct increasingly granular syntactic tree structures. Figure 2. Is the GCNN architecture introduced in [15].

![Figure 2.: GCNN Architecture [15]](image)

In the above architecture multiple layers of gated convolutions are utilized to process sequential data.
Each layer includes a convolutional block that generates two distinct convolutional outputs and a gating block that uses one output to gate the other. The convolutional block applies "causal convolutions" to the input using a window that only overlaps the current and preceding timesteps. The two convolutional outputs are then sent to the gating block, which element-wise multiplies one output by the sigmoid of the other output to select the pertinent information for predicting the next word. The model employs a residual skip connection and, on occasion, a bottleneck structure within a layer to reduce computational expense. Multiple GCNN model layers can be stacked, and adaptive SoftMax is used to determine which word to select from a language model's vocabulary.

GCNN performs better than LSTM on the Google billion words dataset. In order to accurately compare these methods, they each manage the same number of GPUs. This research facilitates the design of convolutional layers for our implementation of GCNN as opposed to CNN. Residual connections mitigate the problem of vanishing gradients that arises as models with a significant number of layers are built deeper. The substantial reduction of computational costs while producing comparable results with RNNs and LSTM networks enables this new model to be applied to a variety of NLP tasks, including text normalization [16]. Figure 3 illustrates an overview of the architecture. In the first layer, characteristics of each word are extracted. The second layer does not view the phrase as a collection of words, but as a sequence with local and global structure from which characteristics can be extracted. The subsequent layers are conventional Gated CNN layers.

![Figure 3.: Neural Network Architecture using GCNN for NLP [16]](image)

2.2.3 Hybrid Bidirectional LSTM

Text normalization is a necessary stage for speech and language applications, such as Automatic Speech Recognition (ASR), because it facilitates the separation of concerns during data storage and preprocessing. To address this issue, a hybrid encoder-decoder model employing bidirectional LSTM for normalization has been proposed [17]. For known words, the model is trained at the word level, and for unknown words, at the character level. The dataset contains five columns, including sentence and token IDs, the class to which the word belongs, and the before and after normalized representations of the word.

Using character-level encoders for indeterminate words and word-level encoders for specified words,
the proposed hybrid model has generated promising results. Unlike RNNs and CNNs, Sunil Kumar and others [17] in their model uses a bidirectional LSTM encoder. Nevertheless, there is still space for advancement, such as training the entire dataset on a high-computing system and implementing adversarial training and hyperparameter tuning.

2.2.4 Reformer: The Efficient Transformer

N. Kitaev and A. Levskya [18] enhance the architecture of the original transformer. After analyzing the performance of different types of attention layers, they implement a transformer with improved attention. In place of feedforward architecture, the study introduces reversible transformer architecture. In this paper, Reversible Transformer is referred to as a Reformer. The authors replace dot-product attention with locality-sensitive hashing to reduce its complexity to NlogN. Figure 4 shows how the proposed LSH algorithm works to reduce design complexity of a transformer –

![Figure 4: LSH in Reformer [18]](#)

Furthermore, they use reversible residual layers instead of the standard residuals. The resulting model, performs on par with Transformer models but is much more memory-efficient and much faster on long sequences. We aim to make use of reformers in place of transformers in our model design in order to reduce space complexity. Figure 5 illustrates how reversible residual layers work in practice –

![Figure 5: Reversible Residual Layers [18]](#)

This study highlights the importance of text normalization and presents a practical implementation strategy for its use in speech and language applications. Table 2 summarizes the results and findings of the existing text-to-speech approaches discussed in this section.
Table 2: Existing TTS Approaches

<table>
<thead>
<tr>
<th>Model</th>
<th>Architecture</th>
<th>Techniques</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN with attention</td>
<td>RNN + attention layers</td>
<td>Text classification, Seq2Seq</td>
<td>High</td>
</tr>
<tr>
<td>LSTM</td>
<td>LSTM</td>
<td>Text classification, Seq2Seq</td>
<td>High</td>
</tr>
<tr>
<td>Hybrid Bi-directional LSTM</td>
<td>Bi-directional LSTM</td>
<td>Character-level and word-level encoding, Seq2Seq</td>
<td>High</td>
</tr>
<tr>
<td>CNN</td>
<td>Multi-layer CNN</td>
<td>Hierarchical representations, copying mechanism</td>
<td>Promising</td>
</tr>
<tr>
<td>Gated Convolutional Neural Network (GCNN)</td>
<td>GCNN + Residual skip connection</td>
<td>Stacked convolutions, hierarchical features, gated temporal convolutions</td>
<td>Competitive with LSTM, reduced computational costs</td>
</tr>
</tbody>
</table>

2.3 Transfer Learning

2.3.1 BERT Pre-Trained

Transfer learning via pre-trained models like BERT has shown promising results in language modeling and other NLP tasks. BERT's bidirectional design uses transformers to learn contextual relationships between words in a text, allowing it to understand the context of a word based on its surroundings. BERT's pre-training and fine-tuning steps make it highly adaptable to a wide range of tasks [19]. It has also been identified that text normalization can improve the performance of BERT models for other NLP applications [20]. Jae Hun Ro et al [1] found fine-tuning BERT to be the most effective approach for text normalization in TTS, outperforming RNN models.

2.3.2 Parameter Efficient Transfer Learning

An efficient transfer mechanism in NLP involves fine-tuning big pre-trained models. However, fine-tuning is parameter inefficient when there are several downstream activities since each action requires a whole new model. In their study, N. Houlsby et al [21] instead transfer via adaptor modules. By adding only a few trainable parameters per job and allowing for the addition of new tasks without having to go back and complete the old ones, adapter modules produce a compact and flexible model. Since the original network's parameters are fixed, there is a high level of parameter sharing.

Keeping the original network's settings constant results in a high degree of sharing of parameters they use the recently suggested BERT Transformer model to 26 different text categorization tasks, including the GLUE benchmark [21, 22]. By adding only 3.6% parameters per job, they were able to match the performance of complete fine-tuning to within 0.4%. On the other hand, fine-tuning trains all the variables for each challenge.

The basic model is the open, pre-trained BERT Transformer network. They use BERT to do classification. Each sequence starts with a unique "classification token" as the first token. To anticipate the class label, they add a linear layer to the token's embedding. With Adam whose learning rate increases linearly throughout the first 10% of steps before gradually decaying to zero, we optimize. With a batch
size of 32, all runs are trained on 4 Google Cloud TPUs.

2.3.3 Transfer Learning with a Unified Test-to-Text Transformer

In their systematic study [23], the authors examine various elements such as pre-training objectives, architectures, transfer methodologies, and unlabeled data sets to achieve cutting-edge results on language understanding tasks. They fuse this knowledge with scale and offer the "Colossal Clean Crawled Corpus" (C4), which is a data collection of hundreds of terabytes of clean English text collected from the web. This corpus is used to conduct experiments on a large scale, and the authors share their pre-trained models, code, and data collection to encourage further research on transfer learning for natural language processing.

The authors also use relative position embeddings in their work, which provide a distinct learnt embedding based on the offset between the "key" and "query" in the self-attention method. They employ a streamlined version of position embeddings, where each embedding is an extra scalar added to the associated logit used to calculate attention weights. The position embedding parameters are shared among all layers of the model for efficiency, while each attention head in each layer employs a distinct learnt position embedding. The authors use the GLUE and SuperGLUE benchmarks [22], which comprise text classification tasks to test general language understanding abilities. Additionally, they offer the C4 corpus as a comparative dataset to their main dataset to test model performance in custom datasets.
3. METHODOLOGY

3.1 Technical Approach

3.1.1 Model Implementation and Training

The implementation of the proposed models in this project involves a pre-processing step where the dataset is tokenized and the textual data is encoded into vectors (tensors). The dataset and pre-processing steps undertaken to prepare the input will be discussed in detail in the following sections. After data preparation, the input data is provided to various deep learning models depending on whether the model will be used as a classifier or a language model to predict output sequences. Figure 6 depicts the process of training the models for comparative analysis.

Figure 6: Deep Learning Model Training

3.1.2 Convert Predicted Output to Speech

Once the models have been trained, the model with the greatest performance is saved in HDF5 format. HDF5 is a file format and library designed to store scientific data. HDF5 can store two categories of objects: datasets and groups. A dataset is fundamentally a multidimensional array of data elements, and a group is an HDF5 file structure for organizing objects. It can be used to store models including the trained weights and model architecture. Sample input sentences from the test data are processed by the saved model, which generates a sequence of the input's predicted speech form. The output is passes through the library; pyttsx3 is a Python library for converting text to speech. In contrast to other libraries, it operates offline and is compatible with Python 2 and 3. Following this, we save each audio file as a .wav file. Figure 7 explains this process of converting the model’s predicted output to audio for analysis.

Figure 7: Convert Output to Audio
3.2 Development Platform

Google Collab Pro is a cloud service that accelerates deep learning on Jupyter notebooks using GPUs. The cloud platform is suitable for deep learning applications due to its Tensorflow, Pandas, Keras, and Pytorch libraries and CUDA GPU integration. Collab's Pro version provides a more consistent development environment with the following additional computational resources -

- 100 GPU units
- 25 GB of RAM
- 170 GB Hard Disk

However, there are limitations on the duration of the session and the size of the uploaded file. Collab Pro is required in order to resolve the difficulty of developing with a dataset of our size.

3.3 Python Modules

Python3 is the language of choice for developing machine and deep learning models due to its versatility and extensive library of scientific computing and data analysis libraries. For this study the following python and open-source libraries are used –

- **Pandas** - used for data manipulation and analysis.
- **Numpy** – used for scientific computing with support for multi-dimensional arrays and matrices, like encoding text to vectors.
- **Matplotlib** – used for data visualization.
- **Tensorflow** - Open-source machine learning framework developed by Google, used for building machine learning models.
- **Pytorch** - Open-source machine learning framework developed by Facebook, used to run machine learning models with CUDA (GPU).
- **Keras** - High-level neural networks API written in Python, used for creating model layers like GCN and GCNN on top of TensorFlow.
- **HuggingFace** - Open-source natural language processing (NLP) library, providing state-of-the-art transformers for NLP tasks.
- **Transformers** - Python library built on top of HuggingFace, providing pre-trained models and utilities for working with transformers, used for BERT pre-trained.
- **NetworkX** – used for converting sequential input to networks and graphs.
- **jiwer** - Python library for evaluating the performance of automatic speech recognition (ASR) systems using word error rate (WER).
4. DATASET

In order to improve the efficiency and effectiveness of natural language generation models, it is necessary to resolve the lack of a substantial number of datasets for text normalization for text-to-speech, as previously discussed. However, one dataset has become the standard-bearer for research in this field. Richard Sproat and Navdeep Jaitly [11] of Google created a dataset from UTF8-encoded Wikipedia articles for their research. The dataset is based on the Wikipedia corpus, specifically 2016 English Wikipedia articles. Each pair in the dataset consists of an original sentence and a normalized rendition of that sentence. By applying a set of rules to rectify common errors and inconsistencies in the text, normalization involves transforming the original sentence into a more standard form. The text sentences were then transmitted through Kestrel – Google's TTS text normalization system to produce the resulting data. For example, "123" in written text must be changed to "one hundred and twenty-three" for spoken language. The resulting dataset is a collection of these sentences that are tokenized into words, where each word has a sentence_id, before, after and a class. This dataset is available on Kaggle in both English and Russian in the form of a .CSV file [25]. Each row represents a record, and each column represents a field, within a CSV (Comma Separated Values) file. CSV files are a common format for exchanging data that can be easily imported into spreadsheet programs and utilized in machine learning operations. The image shown below depicts the data file before pre-processing. The first column represents the class, the next represents textual form of the word, and the last column indicates the form of the word after speech normalization. The Figure 8 represents the structure of the data file and how it looks prior to processing.

![Original Data File](image-url)
The total number of words/tokens in the dataset are eleven million seven thousand seven, divided into train and test. Table 3 represents the number of tokens in train and test dataset, respectively.

Table 3: Number of Tokens [12]

<table>
<thead>
<tr>
<th>Data</th>
<th>No. of tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td>9,918,442</td>
</tr>
<tr>
<td>Test</td>
<td>1,088,565</td>
</tr>
</tbody>
</table>

The CSV file is imported into a panda DataFrame in order to comprehend its structure. The columns listed below are added to the DataFrame. ADDRESS, CARDINAL, DATE, DECIMAL, DIGIT, ELECTRONIC, FRACTION, LETTER, MEASURE, MONEY, ORDINAL, PLAIN, TELEPHONE, TIME, and VERBATIM are the distinct classes assigned to each token. Each of these categories identifies the token class. This data unquestionably serves as the foundation for exploratory research in this field and is a standard dataset for the TTS text normalization challenge.

4.1 Exploratory Data Analysis

Exploratory data analysis (EDA) is a crucial stage in the process of data analysis, which entails exploring and summarizing data to gain insights into its characteristics and distribution. This section examines the tokens per class within a dataset. This entails analyzing the distribution of word sequences across distinct data classes or categories. Figure 9 lists the number of tokens per each of the distinct classes in our dataset.

![Figure 9: Number of tokens per class](image)

Then, we construct a frequency versus class diagram to visualize the distribution of tokens per class. This entails comparing the frequency of each token to the class to which it belongs. Figure 10 below shows the frequency of the occurrence of the tokens of each class.
It is evident from Figures 9 and 10 that the "PLAIN" class dominates the dataset due to the high proportion of common words in Wikipedia articles. Dealing with textual input that contains a combination of date, letters, and cardinality presents a significant challenge due to the significant difference between the written and spoken forms of such information. The process of normalization involves converting the text from its written to its spoken form. By identifying the classes and tokens that exhibit the greatest degree of variation between these two forms, we can gain insight into the distinctive linguistic characteristics of these classes and develop more efficient methods for dealing with them. In order to visualize which classes and tokens exhibit the most significant distinctions between their written and spoken forms after normalization, we generate the following graphs.

The graphs in Figure 11 and Figure 12 illustrate the tokens that show maximum degree of change and the classes that these tokens belong to.
To conduct additional analysis, we extracted the relevant sentences from the dataset and used them input sequences. In the Figure 13 below, the sentences in the input are clearly shown.

In addition to analyzing the textual input that includes date, letters, and cardinality, we also generated another dataset using Twitter data. To achieve this, we tokenized tweets and then passed them through an open-source framework similar to Google's Kestrel - Sparrowhawk. This process resulted in a new dataset consisting of 1000 tokens that are more closely aligned with the slang and acronym language used in tweets.

Due to a lack of resources, we had to manually annotate the errors in "class" and "after" columns. The sample Twitter dataset is saved in a CSV format like the previously discussed Wikipedia dataset. Each input sequence in this dataset is a tweet, and the original dataset used for this purpose is the sentiment140 dataset. This dataset contains 1,600,000 tweets extracted using the Twitter API, which are available on Kaggle. The dataset is comprised of several columns, including the "target" column which represents the polarity of the tweet, the "ids" column which represents the ID of the tweet (2087), the "date" column which represents the date of the tweet, the "flag" column which represents the query (lyx), and the "user" column which represents the user that tweeted (robotickilldozr). We only use the "text" column of the dataset, which contains the actual text of the tweet (Lx is cool), to generate our resultant CSV file. Figure 14 below describes the structure of the .CSV file generated after pre-processing the twitter dataset for our implementation approach.
4.2 Data Preparation

4.2.1 Data Preparation for LSTM-GRU Model

The pre-processing of the input data involves several steps. Firstly, the output labels (i.e., classes) are encoded using the LabelEncoder function from the scikit-learn library, which fits the encoder to the output labels and transforms the labels into integers. The resulting encoded labels are stored in a separate variable. Next, the Keras library function is used to convert the encoded labels to one-hot encoded vectors, which are stored in another variable. Figure 15 demonstrates the code snippet used to encode the data into vectors.

The input data is expected to be a list of lists of strings, where each inner list represents a single sentence and each string represents a token in that instance. Thus, each instance is appended to the list as a list of one string element. The Keras Tokenizer is used to build the vocabulary of the tokenizer based on
the unique words in the input data. The vocabulary size is computed, which will be the input shape for the model's first layer. The maximum length of each sequence of integers is set to 2 words. Sequences shorter than 2 words are padded with zeros, and sequences longer than 2 words are truncated to 2 words. In order to provide a uniform input to our models we use the code described in the Figure 16 below to appropriately pad the sequences.

```
# pad documents to a max length of 2 words
max_length = 2
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

```
[ [  6734  1709]  
  [  12   10]  
  ...  
  [  2031   290]  
  [    3    0]  
  [ 20589   290] ]
```

Figure 16: Padding Sequences

Pre-trained word embeddings from a Global Vectors for Word Representation (GloVe) file are used to create an embedding matrix. An empty embedding matrix is created with dimensions (vocab_size, 100), where vocab_size is the number of unique words in the input data, and 100 is the size of the word vectors in the GloVe file. For each word in the word index created by the Tokenizer, the corresponding word vector from the embeddings_index dictionary is fetched, if it exists. If the word is not in the embeddings_index dictionary, the vector components are generated by converting each character of the word to its ASCII code. Figure 17 is a sample of an embedding matrix generated for a given input.

```
array([[ 1.0790000e-01,  5.0030999e-02,  1.0840000e-01,  4.9830983e-02,  
        8.9735000e-02,  2.1860000e-01,  5.0050000e-01,  9.8760000e-02,  
       -2.1350000e-01,  3.4200000e-01,  9.2670000e-02,  1.0590000e-01,  
       -1.3200000e-01,  8.1259999e-01,  1.8730000e-01,  4.2509999e-01,  
        9.0039999e-01,  1.3971000e-01,  1.8700099e-01,  4.0510000e-01,  
       -5.0950000e-01,  5.0950000e-01,  4.8360099e-01,  3.0030999e-01,  
       -2.9050000e-01,  3.4942001e-01,  9.9590000e-02,  7.8360000e-02,  
       -2.3159999e-01,  4.7210000e-01,  7.8125000e-01,  1.3450000e-01,  
       -2.1350000e-01,  3.5950000e-01,  4.8360099e-01,  1.0870000e-01,  
       -2.8530000e-01,  2.1470000e-01,  3.9240000e-02,  7.9370000e-02,  
       -7.6340000e-01,  3.2499996e-01,  7.5840997e-01,  1.0853000e+00,  
       -4.1381000e-01,  4.5210000e-01,  1.2111000e-01,  5.1367000e-01,  
       -1.3349996e-01,  1.1377999e-01,  2.8760000e-01,  1.0774000e-01,  
        5.5040000e-02,  1.5386000e-01,  1.8550000e-02,  2.9721000e-02,  
        2.4160000e-01,  9.2490000e-01,  2.3911000e-01,  2.8233999e-01,  
        3.4770000e-01,  5.1621000e-01,  4.3386000e-01,  3.6851000e-01,  
        7.6529998e-01,  7.2102000e-02,  2.7930099e-01,  9.2560099e-01,  
       -5.0330999e-02,  8.5850000e-01,  1.2509999e-01,  9.2560099e-01,  
       -2.9051000e-01,  1.0939999e-01,  6.7200099e-02,  2.1378999e-01,  
        4.6900000e-01,  2.1377000e-01,  8.4800099e-01,  5.2529999e-02,  
        5.0368000e-01,  2.0830099e-01,  6.7660099e-01,  1.3160000e-01,  
       -1.5580000e-02,  2.8765000e-01,  7.2210997e-01,  5.2060025e-01,  
       -7.2289997e-02,  1.5194000e-01,  1.3139996e-01,  8.9109995e-02,  
       -3.1869000e-01,  6.1418998e-01,  6.2593997e-01,  4.1567998e-01,  
       -3.8179001e-02,  3.9803999e-01,  4.7660099e-01,  3.9983999e-04]}]
```

Figure 17: Embedding Matrix

4.2.2 Data Preparation for Transformer and Reformer Models

Firstly, any missing values in the dataset are replaced with an empty string using the fillna method of a pandas DataFrame. Next, a lambda function is applied to each row of the DataFrame using the apply
method. This function tokenizes the sentence using a tokenizer object and stores the resulting tokens in two new columns named before_tokens and after_tokens. The code snippet presented in Figure 18 depicts the pre-processing step of filling any null values in the dataset with an empty string.

```python
# fill missing values with an empty string
df['before'].fillna('', inplace=True)
df['after'].fillna('', inplace=True)
```

Figure 18: Fill NaN Values

The tokenized sequences are then grouped by sentence and concatenated into a single long sequence using the groupby method of a pandas DataFrame and a lambda function that flattens the resulting list of lists. The input and output sequences are encoded using a map function that applies the encode function to each example in the dataset. The encode function takes an example and two lists of sequences as arguments and returns the example with the input and output sequences encoded as dictionaries. Figure 19 shows the padding of sequences to match the maximum input length of 2 as it is one of the input dimensions for our models.

```python
df[‘before_tokens’] = df[‘before’].apply(lambda x: tokenizer(str(x), max_length=required_seq_len, padding=’max_length’, truncation=True, return_tensors=’pt’))
df[‘after_tokens’] = df[‘after’].apply(lambda x: tokenizer(str(x), max_length=required_seq_len, padding=’max_length’, truncation=True, return_tensors=’pt’))

# Create input and output sequences
input_sequences = df.groupby(“sentence_id”)[“before_tokens”].apply(lambda x: x.iloc[0]).tolist()
output_sequences = df.groupby(“sentence_id”)[“after_tokens”].apply(lambda x: x.iloc[0]).tolist()
```

Figure 19: Pad Sequences

The Dataset class from the datasets package is then employed to generate a training and validation dataset. Using the train_test_split method, the dataset is divided into a training set and a validation set. The resulting datasets are then encoded using the map and encode methods. We use the map function in order to perform the encoding of each train and validation dataset as shown in Figure 20.

```python
# Encode the input and output sequences
train_dataset = train_dataset.map(encode, batched=False, remove_columns=[‘sentence_id’, ‘token_id’, ‘class’, ‘before’, ‘after’])
val_dataset = val_dataset.map(encode, batched=False, remove_columns=[‘sentence_id’, ‘token_id’, ‘class’, ‘before’, ‘after’])
```

Figure 20: Encode Train and Validation Dataset

The pre-processing pipeline consists of the replacement of missing values, tokenization of sentences, concatenation of sequences, encoding of input and output sequences, and construction of training and validation datasets. These stages are essential for preparing the data for use in sequence-to-sequence model training. These datasets can be used to train a neural network model capable of producing target sequences from input sequences.
5. IMPLEMENTATION

5.1 LSTM-GRU Model

5.1.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that has found extensive use in natural language processing (NLP) and other sequence-related tasks. The development of LSTMs was motivated by the desire to solve the problem of vanishing gradients in conventional RNNs, which occurs when the gradients used to update the model's parameters become too small. This problem causes these models to learn slowly and perform poorly. LSTMs solve this problem by integrating a gated memory cell that can learn and forget information selectively over time. Multiple gates within the memory cell regulate the passage of information into and out of the cell. These gates include the input gate, which determines how much new information should be stored in the cell, the disregard gate, which determines which information should be discarded, and the output gate, which determines how much information should be passed on to the subsequent time step. Figure 21 below illustrates the components of a long short-term memory model architecture.

The LSTM architecture is defined by a collection of symbols representing its various components and functions. The symbol "X" represents the scaling of data, whereas the symbol "+" represents the addition of data. The symbol "" represents the sigmoid layer, which regulates the passage of data into and out of memory cells. The "tanh" symbol denotes the hyperbolic tangent function, which is applied to the candidate values prior to passing them to the memory cell. It employs two memory states: the output of the previous LSTM unit, represented by "h(t-1)", and the memory from the previous LSTM unit, represented by "c(t-1)". "X(t)" represents the current input, while "c(t)" represents the newly updated memory. "h(t)" represents the LSTM unit's current output.

The forget gate, which is controlled by a sigmoid layer, determines which portions of old output
should be discarded based on the current input. A new input is then evaluated using a sigmoid layer and a tanh layer to decide which information should be updated in the memory cell. The updated cell state is obtained by adding the new memory to the old memory, resulting in a cell state that retains relevant information and discards irrelevant information. Finally, a sigmoid layer decides which portions of the cell state should be outputted as it can output either 0 or 1, and the cell state is passed through a tanh layer to generate possible output values. The tanh function is utilized, whose second derivative can persist over a wide range prior to zeroing out. LSTM enables the model to learn and predict the answer to a question based on long-term dependencies rather than immediate dependencies.

### 5.1.2 Gated Recurrent Unit (GRU)

Similar to Long Short-Term Memory (LSTM) are GRUs. GRU, like LSTM, employs gates to control the information flow. They are comparatively more recent than LSTM. Because of this, they are superior to LSTM and have a simplified architecture. In contrast to the Long Short-Term Memory (LSTM) architecture, the Gated Recurrent Unit (GRU) does not have a distinct cell state (Ct). Instead, the GRU contains only one concealed state (Ht). This simplified architecture provides benefits for computational efficiency and training performance. Figure 22 illustrates this simplified architecture of a GRU with its components.

![GRU Architecture](image)

Figure 22: GRU Architecture [26]

At each timestamp t, it receives an input Xt and the preceding timestamp's hidden state Ht-1. Later, it outputs a new hidden state Ht, which is subsequently transmitted to the subsequent timestamp. Currently, a GRU cell consists predominantly of two gates as opposed to three gates in an LSTM cell. The initial gate is the Reset gate, while the second is the update gate. The Reset Gate handles the network's short-term memory, which is the hidden state (Ht). Due to the sigmoid function, the value of rt will vary between 0 and 1. Here, Ur and Wr are the reset gate's weight matrices.

The Reset Gate formula is as follows:  

\[ r_t = \sigma (x_t \cdot U_r + H_{t-1} \cdot W_r) \]
The Update Gate formula is as follows:  \[ u_t = \sigma (x_t * u + H_t * W_u) \]

5.1.3 Model Architecture

The proposed model architecture for the classification task involves the use of an embedding layer, which takes the vocabulary size, embedding dimension, and embedding matrix as input. The input length parameter specifies the length of the input sequence, which is set to 2. The trainable parameter is enabled, allowing the weights of this layer to be updated during training. Subsequently, a long short-term memory (LSTM) layer with 64 memory units is added to the model, along with a dropout parameter of 0.2 to mitigate overfitting. This is followed by the addition of a gated recurrent unit (GRU) layer with 32 units, along with a dropout parameter of 0.2.

Finally, a dense layer with 16 units and a SoftMax activation function is added to the model. The SoftMax activation function is typically used in multi-class classification problems, as it produces a probability distribution over the different classes. Figure 23. below depicts the architecture of the hybrid LSTM-GRU model, the first model used in this study.

5.2 Transformer – GCN and GCNN

5.2.1 Transformer

Transformers are presently the most advanced models for manipulating sequences. The most prominent of these models is machine translation, which has gained prominence predominantly in text processing tasks. The dominance of transformer models on benchmark leaderboards for diverse NLP and NLG applications, extending from grammar correction to speech processing, demonstrates the efficiency of transformer models in natural language generation (NLG). As a result, transformer architectures are
enduring a significant development surge, with numerous advancements aimed at enhancing their performance and capabilities. Transformers that operate as feed-forward-only devices require a slightly different hardware design. Transformers are better suited to operate on modern machine learning accelerators than recurrent networks because, unlike recurrent networks, there is no sequential processing: the model does not need to process a sequence of elements to develop a useful hidden cell state. Prior to understanding the functionality of the transformer architecture, it is necessary to develop an understanding of the underlying principles of positional encoding and attention.

### 5.2.1.1. Positional Encoding

The technique of positional encoding is employed in transformer-based models to convey the sequence order to the model. As transformers lack the inherent concept of sequential ordering present in recurrent models, positional encoding is added to the input embeddings to provide the model with information about the position of each token in the sequence.

Our approach to positional encoding involves the use of sine and cosine functions. Initially, we compute the angle rates for the positional encoding using a method that considers the position of each token in the sequence, the index of each dimension in the embedding vector, and the dimensionality of the embedding vector. In Figure 24. We use the formula described below in order to compute the angle rates for each position in the sequence

![Figure 24: Computing Angle Rates](image)

```python
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
    return pos * angle_rates
```

The angle rates are determined using a formula that generates values dependent on the position and index of each token in the sequence. Specifically, the formula used is –

\[
1 / (10000^{2 \times \text{embedding\_vector}/2} / \text{dimension\_of\_embedding\_vector}) \] [3]

The positional encoding is then applied to the input sequence using the previously generated angle rates. The sine and cosine functions are applied to the angles generated for each position and index. Combining the resulting values produces the final positional encoding matrix. The sin() function is specifically applied to the even indices of the encoding matrix, while the cos() function is applied to the odd indices.
5.2.1.2. Self-Attention

The self-attention mechanism computes a weighted sum of the values of an input sequence based on the proximity of each position to the current position. The self-attention mechanism enables the inputs to interact with each other by computing the attention of all other inputs regarding a specific input.

- The input vectors are first multiplied by three weight matrices to derive the key (K), query (Q), and value (V) vectors, which are then used to calculate self-attention.
- The query vector (Q) of the current input is multiplied with the key vectors of the other inputs, and the resulting score is then divided by the square root of the dimensions of the key vector (dk).
- After applying the SoftMax function to the scores, the value vector is multiplied by the resulting vector.

The self-attention matrix for input matrices (Q, K, V) is calculated as the concatenation of the query, key, and value vectors using the following formula presented in Figure 27 –
5.2.1.3. Multi-Headed Attention

Multi-headed attention is another type of attention mechanism used in transformer-based models. The process for calculating multi-headed self-attention involves the following steps:

- Produce embeddings for each word in the sentence input.
- Create \( h \) unique attention heads, each with a unique weight matrix \( (W(Q), W(K), W(V)) \).
- Multiply the input matrix by each weight matrix to generate the key, value, and query matrices for each attention head.
- Apply the attention mechanism to each attention head's query, key, and value matrices to generate an output matrix.
- The output of the multi-headed attention layer is produced by concatenating the output matrices of each attention head and performing a dot product with the weight matrix \( W^o \).

Mathematically multi-head attention is represented by the following formula:

\[
MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^o
\]

5.2.1.4. Encoder-Decoder Architecture

The encoder is responsible for processing the input sequence and generating a high-dimensional vector representation of the sequence that can be utilized as input by the decoder. It consists of numerous layers of self-attention and feed-forward neural networks, each of which performs a distinct function on the input sequence. The self-attention layers allow the encoder to examine the entire input sequence at once and to discover dependencies between various locations within the sequence. Layers of feed-forward neural networks provide nonlinear transformations that enable the encoder to characterize complex relationships between input characters.

Combining the self-attention layer with a dense feed-forward layer results in a single encoder layer. The feed-forward layer consists of two linear layers separated by a rectified linear unit (ReLU). That is,
the input is first transformed by a linear layer (matrix multiply), the resulting values are then clipped to always be 0 or greater, and the result is then fed into a second linear layer to generate the output of the feed-forward layer. The architecture of the encoder block in a transformer is depicted by the Figure 30 below.

Based on the encoder's output and the previous tokens generated by the decoder, the decoder in this model is responsible for generating the output sequence one token at a time. The decoder, like the encoder, consists of multiple layers of self-attention and feed-forward neural networks. It also includes a masked self-attention layer to ensure that each token in the output sequence can only attend to previous tokens, preventing the decoder from deceiving by attending to future tokens.

Encoder and decoder are integrated in a multi-layer neural network to generate a high-quality output sequence given an input sequence. Using self-attention and feed-forward layers, the encoder generates a high-dimensional vector representation of the input sequence, providing contextual information for each token in the input sequence to the decoder. The decoder then combines this contextual information with the previously generated tokens to produce the subsequent token in the output sequence.

5.2.2 Graph Convolutional Network (GCN)

GCN (Graph Convolutional Network) is a form of neural network that works with graph data. It learns representations of nodes and edges in the graph using a convolutional approach. Unlike traditional convolutional networks, which operate on 2-D arrays, Graph Convolutional Networks (GCN) operate on graphs. Figure 31 illustrates the key difference between a normal convolution and graph convolution.
GCN employs an adjacency matrix $A$ in which each element $A_{ij}$ equals 1 if nodes $i$ and $j$ are connected and 0 otherwise. In addition, all diagonal elements are set to the value 1 to denote self-connections. This adjacency matrix assists in identifying a node's companions. A node's output in a hidden layer is dependent on itself and its companions. The adjacency matrix $A$ is changed to, which is equal to $A$ plus $I$. This modification assures that the convolutional operation is performed on the node and its neighbors. GCN applies convolution over a graph utilizing the adjacency matrix $A$ in order to discover representations of nodes that encompass both local and global information. Figure 32 is a representation of the Adjacency Matrix used in GCN.

\[
A = \begin{bmatrix}
0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

Figure 32: Adjacency Matrix [28]

5.2.3 **Gated Convolutional Neural Network (GCNN)**

A GCNN is an alternative to recurrent networks for capturing long-term dependencies while avoiding sequential operations for enhanced parallelizability. Consequently, gated temporal convolutions replace the recurrent connections typically used in RNNs. Combining convolutional networks with a gating mechanism, a Gated Convolutional Network is a form of language model. Zero padding is used to prevent future context from being visible. Gated convolutional layers can be layered hierarchically on top of one another. The predictions of the model are then obtained using an adaptive SoftMax layer. In the original paper, the hidden layer after the Gated CNN is defined as:

\[
h(X) = (X \ast W + b) \odot \sigma(X \ast V + c) [29]
\]

where $X$ is the input tensor of size $N \times m$, $W$ is the weight tensor of size $k \times m \times n$, $b$ is the bias vector of size $n$, $V$ is the weight tensor of size $k \times m \times n$, $c$ is the bias vector of size $n$, $\sigma$ is the sigmoid function, and $\odot$ is the Hadamard product operator. Here, $m$ and $n$ are the number of input and output feature maps.
(i.e., the number of channels), and k is the patch size. Please note that the batch size is omitted in this notation. Gated CNN is typically implemented with 1D convolutions, where the input tensor X has dimensions of \([ B, m, N]\), where B is the batch size, m is the number of channels, and N is the length of the token sequence.

5.2.4 Model Architectures

**GCN-Transformer Model Architecture**

The model is an attention-based sequence-to-sequence (seq2seq) model. It contains an encoder with input, embedding, positional encoding, and self-attention layers, as well as a feedforward network with GCN layers. The decoder is comprised of an input layer, an embedding layer, a positional encoding layer, a self-attention layer with a causal signal, an encoder-decoder attention layer, and a feedforward network with dense layers. Some layers contain residual connections, and layer normalization is used to enhance training stability. The output is a dense layer that generates a probability distribution over the vocabulary of interest. The Table 4 represents the model architecture that has been designed for this study to evaluate GCN-Transformer model.

<table>
<thead>
<tr>
<th>Component</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Type</strong></td>
<td>seq2seq GCN-Transformer</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>Two input layers for the source and target sequences respectively</td>
</tr>
<tr>
<td><strong>Encoder</strong></td>
<td>• Input layer</td>
</tr>
<tr>
<td></td>
<td>• Embedding layer</td>
</tr>
<tr>
<td></td>
<td>• Positional encoding layer</td>
</tr>
<tr>
<td></td>
<td>• Self-attention layer</td>
</tr>
<tr>
<td></td>
<td>• Feedforward network with GCN layers:</td>
</tr>
<tr>
<td></td>
<td>• 1 GCN layer with ReLU activation function and 16 filters</td>
</tr>
<tr>
<td></td>
<td>• 1 GCN layer without an activation function and d_model filters</td>
</tr>
<tr>
<td></td>
<td>• Residual connection</td>
</tr>
<tr>
<td></td>
<td>• Layer normalization</td>
</tr>
<tr>
<td><strong>Decoder</strong></td>
<td>• Input layer</td>
</tr>
<tr>
<td></td>
<td>• Embedding layer</td>
</tr>
<tr>
<td></td>
<td>• Positional encoding layer</td>
</tr>
<tr>
<td></td>
<td>• Self-attention layer</td>
</tr>
<tr>
<td></td>
<td>• Feedforward network with dense layers:</td>
</tr>
<tr>
<td></td>
<td>• 1 dense layer with ReLU</td>
</tr>
</tbody>
</table>
GCNN-Transformer Model Architecture

The following table outlines our model architecture for a GCNN-Transformer. To achieve language modelling, we utilize a customized BERT-based model in conjunction with custom GCNN layers. Table 5 under, explains the architecture of the custom GCNN-Transformer architecture that we implement for experimentation.

<table>
<thead>
<tr>
<th>GatedCNNLayer:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• hidden_size:</td>
<td>The number of hidden units in each layer is 768</td>
</tr>
<tr>
<td>• conv1: A 1D convolutional layer with the same input and output dimensions as the hidden_size.</td>
<td></td>
</tr>
<tr>
<td>• conv2: Another 1D convolutional layer with the same input and output dimensions as the hidden_size.</td>
<td></td>
</tr>
<tr>
<td>• sigmoid: A sigmoid activation function used to squash the output of conv1 between 0 and 1, representing the gating values.</td>
<td></td>
</tr>
<tr>
<td>• tanh: A hyperbolic tangent activation function used to squash the output of conv2, representing the activations.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CustomBertModel:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• gated_cnn_layers: A list of GatedCNNLayer modules, one for each of the 2 hidden layers</td>
<td></td>
</tr>
<tr>
<td>• forward method: The method takes the following inputs:</td>
<td></td>
</tr>
<tr>
<td>• input_ids: Token indices for input text.</td>
<td></td>
</tr>
<tr>
<td>• attention_mask: Mask to avoid attending to padding tokens.</td>
<td></td>
</tr>
<tr>
<td>• token_type_ids: Segment token indices to indicate the first and second portions of the input.</td>
<td></td>
</tr>
</tbody>
</table>
• position_ids: Position indices for input tokens.
• head_mask: Mask to nullify selected heads of the self-attention modules.
• inputs_embeds: Optionally, instead of passing input_ids, you can pass pre-computed embeddings.
• encoder_hidden_states: Optional hidden states for the encoder
• encoder_attention_mask: Optional attention mask for the encoder
• past_key_values: Tuple of key and value states for sequence generation.
• use_cache: Flag to indicate whether to use the past key and value states
• output_attentions: To output the attention weights.
• output_hidden_states: To output the hidden states.

5.3 Reformer Language Model

The Reformer model, augmented with a language modeling (LM) head, is a recent extension to the transformer-based architecture that addresses the challenge of efficiently handling very long sequences of text data. The Reformer model leverages two key strategies to mitigate attention and memory allocation issues: the use of locality-sensitive hashing (LSH) to reduce the computational cost of attending across extended sequences, and reversible residual layers to maximize the utilization of available memory.

5.3.1 Locality Sensitive Hashing (LSH)

Finding equivalent vector pairs is a computationally intensive task, especially when dealing with large datasets. Even with relatively modest datasets, the sheer number of comparisons required to compare all vectors can become unmanageable. To address this difficulty, it becomes necessary to reduce the number of required comparisons. Ideally, the process of comparison should be restricted to those vectors that are most likely to be similar, or at least potential similarities.

Locality-sensitive hashing (LSH) is a technique that can make this procedure easier. It enables us to restrict comparisons to only those vectors considered to be potential matches, thereby reducing the computational burden of the similarity matching procedure. The Figure 33 shows an overview of how LSH algorithm works using hash functions.
In LSH, a defined set of hash functions is applied to each vector in the dataset. The hash function's output is a value that represents a specific bin or receptacle. By mapping similar vectors to the same bin, LSH enables the high probability retrieval of similar vectors. To accomplish this, LSH attempts to maximize collisions between similar vectors. A collision occurs when the hash function maps two distinct vectors to the same bin. By adjusting the number of hash functions and the size of the segments, LSH can minimize the number of comparisons required to identify similar vectors while controlling the probability of collision between similar vectors.

This method is comparable to how Python dictionaries map a key-value pair to a particular container using a hash function. In contrast, the goal of LSH is to maximize collisions between similar inputs, as opposed to minimizing them as in conventional hashing.

5.3.2 Model Architecture

The model architecture for a Reformer with Language Modelling Head (LM Head) that we implement for our study is

- **Input Encoding**: The input sequence is first embedded using a trainable embedding layer. Then, it is processed by a reversible multi-layer Transformer encoder, which uses the Reformer's unique locality-sensitive hashing mechanism to reduce the memory required for processing very long sequences.
- **Masked Self-Attention**: The Reformer model also uses masked self-attention, where the attention mechanism is restricted to attend only to positions within a certain distance from each token in the input sequence. This allows the model to handle very long sequences without using excessive memory.
- **Feed-Forward Layers**: After the masked self-attention layer, the Reformer model includes two feed-forward layers, followed by layer normalization. This decoder uses typical self-attention and masked self-attention techniques to process the encoded input sequence and build a probability distribution across the output vocabulary.
- **Lastly**, to generate the output sequence, a trainable LM head is installed on top of the decoder.
Figure 34 below explains the configurations of the Reformer architecture used for this study.

```python
config = ReformerConfig(
    vocab_size=30522,
    hidden_size=256,
    num_hidden_layers=6,
    num_buckets=None,
    num_attention_heads=1,
    feed_forward_size=512,
    max_position_embeddings=512,
    axial_pos_embs=True,
    axial_pos_embs_dim=(64, 192),
    attn_layers=['local', 'lsh', 'local', 'lsh', 'local', 'lsh'],
    lsh_attn_chunk_length=64,
    local_attn_chunk_length=64,
    is_decoder=True,
)
```

Figure 34: Reformer Model Configuration

- **vocab_size**: The size of the vocabulary, which is the number of unique tokens it is set to 30,522.
- **hidden_size**: The size of the hidden layer in the model it is set to 256.
- **num_hidden_layers**: The number of hidden layers in the model, 6 hidden layers.
- **num_buckets**: The number of buckets for the LSH (Locality Sensitive Hashing) attention (AUTO)
- **num_attention_heads**: The number of attention heads for the model. In this case, it is set to 1.
- **feed_forward_size**: The dimensionality of the feed-forward layer is set to 512.
- **max_position_embeddings**: The maximum number of position embeddings is set to 512.
- **axial_pos_embs**: set to True.
- **axial_pos_embs_dim**: The dimensions of the axial position embeddings (64, 192), which sums up to the hidden_size (256).
- **attn_layers**: A list specifying the type of attention layer for each hidden layer in the model it alternates between "local" (Local Self-Attention) and "lsh" (LSH Self-Attention).
- **lsh_attn_chunk_length**: The chunk length for LSH attention. In this configuration, it is set to 64.
- **local_attn_chunk_length**: The chunk length for local attention. In this configuration, it is also set to 64.
- **is_decoder**: A boolean indicating whether the model should be set up as a decoder. In this case, it is set to True, which means the model will be created with a language modeling head.

### 5.3.3 Language Model Head

The Language Modeling (LM) head in the Reformer model layers from Hugging Face constitutes a crucial component that is trainable during the model's training process. The primary constituents of the LM head include:

- A trainable output embedding layer, responsible for projecting the decoder's output into a high-dimensional space.

```python
```
• A linear layer, which processes the output embedding layer's output to form a probability distribution over the output vocabulary.
• A softmax activation function that converts the linear layer's output into a probability distribution.

During the training phase, the LM had is optimized by minimizing the cross-entropy loss between the predicted output sequence and the ground truth output sequence. This optimization process involves adjusting the weights of the LM head to decrease the negative log-likelihood of the actual output sequence, given the input sequence. The LM head plays a vital role in generating the final output sequence derived from the decoder's high-dimensional vector representation of the input sequence. By optimizing the LM head, the model learns to generate more accurate and contextually relevant output sequences, thereby improving its performance on various natural language processing tasks.

5.4 Transfer Learning with BERT

5.4.1. BERT – Bidirectional Encoder Representations

BERT, which stands for Bidirectional Encoder Representations from Transformers, is a Google-developed deep learning model that generates contextualized word embeddings using transformer-based neural networks. The model is pre-trained on a massive corpus of unlabeled text data using a masked language modeling task and a next sentence prediction task. The task of masked language modeling involves randomly masking certain words in a sentence and training the model to predict the masked words based on the context of the surrounding words. In contrast, the next sentence prediction task trains the model to predict whether two given sentences will follow each other in a particular text sequence.

15% of the words in each word sequence are replaced with [MASK] tokens prior to being input into BERT. The model then attempts to predict the original value of the masked words based on the context provided by the unmasked words. In terms of computer science, the prediction of output words requires:

• Adding a layer of classification to the encoder output.
• By multiplying the output vectors by the embedding matrix, the vocabulary dimension is generated.
• Using softmax to calculate the probability of each word in the vocabulary. Below illustrated is the architecture of BERT explaining the input layers of the model in Figure 31.
5.4.2. GPT-2

The GPT-2 model employs a similar architecture to BERT, albeit with a decoder instead of an encoder. The decoder consists of a series of transformer decoder layers, each of which includes a multi-head self-attention mechanism and a position-wise feedforward network. In contrast to BERT, the GPT-2 model is trained on a generative language modeling task that attempts to predict the next word in a given sequence of words.

5.4.3. Model Architecture

The following is the architecture for this BERT language model –

- Define the tokenizer, encoder, and decoder for the model. In this case, the tokenizer is a BERT tokenizer, the encoder is a pre-trained BERT model, and the decoder is a pre-trained GPT-2 language model with a language modelling head (LMHead).

- Define the model as a sequence-to-sequence model, which takes as input the input_ids and attention_mask of the input sequence, as well as the decoder_input_ids and decoder_attention_mask of the output sequence.

- In the forward() method of the TransformerSeq2Seq class, the input sequence is first encoded using the BERT encoder, and the resulting encoder_output is passed as the past key values to the GPT-2 decoder, along with the decoder_input_ids and decoder_attention_mask of the output sequence. The decoder_output is then returned as the logits of the output sequence.

The Table 6 demonstrates the architecture of the BERT model that has been utilized during the course of this research.
Table 6: BERT Model Architecture

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERTEncoder</td>
<td>([batch_size, seq_len, d])</td>
<td>108,310,272</td>
</tr>
<tr>
<td>GPT2Decoder</td>
<td>([batch_size, seq_len, d])</td>
<td>345,931,776</td>
</tr>
<tr>
<td>LMHead</td>
<td>([batch_size, seq_len, d])</td>
<td>502,593</td>
</tr>
<tr>
<td>Total params</td>
<td></td>
<td>454,744,641</td>
</tr>
</tbody>
</table>
6. METRICS AND EXPERIMENTS

6.1 Metrics

To evaluate the model, the primary metric used commonly are accuracy and loss.

- Accuracy: It measures how well a model can correctly predict the class labels of a set of test instances. The formula below illustrates how accuracy is computed, especially for classification tasks: \[
\text{accuracy} = \frac{\text{number of correctly classified instances}}{\text{total number of instances}}
\]

- Loss: It is also known as cost or objective function, is a quantitative measure of how well a machine learning model is performing on a given task. In supervised learning, the loss function is used to calculate the difference between the predicted outputs of the model and the actual outputs, also known as the labels, for a given set of input data. The goal of the model is to minimize this difference, which is also known as the error or the cost.

However, since our models address the natural language processing task of text to speech normalization, we use the metrics better suited to validate language models –

- Word Error Rate (WER): This metric measures the percentage of words in the output that differ from the target text. WER is commonly used in speech recognition tasks and can also be applied to text-to-speech normalization to evaluate the accuracy of the generated speech. WER has a simple math formula. Sum of Errors divided by the total number of words. The only nuance is that errors include Substitutions (S), Insertions (I) and Deletions (D). WER cannot be a negative number, but it could be above 100%. The formula for WER is shown below in Figure 36.

\[
\text{WER} = \frac{S + I + D}{N}
\]

Figure 36: Word Error Rate

- Sentence Error Rate (SER): It is an evaluation metric used in natural language processing (NLP) to measure the accuracy of a system that automatically detects and corrects errors in sentences. SER is calculated as the percentage of sentences in a test set that contain at least one error and were not correctly corrected by the system. In other words, it measures the proportion of incorrect sentences that were not corrected, plus the proportion of correct sentences that were incorrectly marked as incorrect. SER is the primary metric of measuring our model performances.

6.2 Experiments

6.2.1. LSTM-GRU for Classification

In this study, we conducted an experiment to evaluate the performance of the LSTM-GRU model in
two different tasks. Firstly, we used the model for a basic classification task, where all columns in the dataset were considered as input and the "class" column was considered as the target column. Secondly, we assessed the performance of the same model as a language model, where the input was a sequence of words and the output was a normalized sequence of words. Our aim was to compare the performance of the LSTM-GRU model in these two distinct tasks.

The input of this model is fundamentally different from our other experiments as it outputs a class-based classification result based on the other features. The code here in Figure 37 describes the input and output columns that are defined for this experiment.

![Figure 37: LSTM-GRU Classification Input](image)

### 6.2.1.1 Results and Observations

In Figure 38, we can see the results generated with this experiment while trying to perform the classification task using our hybrid LSTM-GRU model.

![Figure 38: LSTM-GRU Accuracy and Loss](image)

This experiment only focuses on the primary parameters of classification model – accuracy, loss, precision, and recall. This experiment yields in a training accuracy of 97% with validation loss 40%. This reflects that the model significantly overfits on the “PLAIN” class which dominates the dataset. There is a significant difference in the training and accuracy due to high amounts of loss in the model. The red dotted line signifies the point of overfitting.
6.2.2. LSTM-GRU for seq2seq model

In this section, we maintain the same LSTM-GRU hybrid model architecture but adapt it for a different application. Specifically, we design the model to function as a language model that processes input sequences representing textual sentences. The goal of the model is to generate output sequences comprising tokens that represent the corresponding text-to-speech conversion of the input sentence. This experimental setup allows us to investigate the model's performance and capabilities in handling natural language processing tasks such as text-to-speech conversion.

In this LSTM-GRU model configuration, the model's input sequences consist of the text form of the terms from the "before" column. The output sequences are generated based on the "after" column, which comprises the words' normalized speech forms.

The input sequences are preprocessed by tokenizing the text in the "before" column and then padding or truncating them to the specified length. Similarly, the output sequences are preprocessed by tokenizing the text in the "after" column and appropriately buffering or truncating them.

- Input sequence: ["$42", "3.14"]
- Output sequence: ["forty-two dollars", "three point one four"]

6.2.2.1 Results and Observations

![Error Rate over epochs](image)

Figure 39: LSTM-GRU WER and SER

The Figure 39 above illustrated the results of the model with respect to the different error rate metrics. Examining the results revealed that the LSTM model performed the classification task with a high degree of accuracy. However, the model also exhibited overfitting, indicating that its ability to generalize to unobserved data may be limited. In comparison, the LSTM model performed marginally better on the sequence-to-sequence normalization assignment for text-to-speech. Despite this, the LSTM model encountered difficulties when processing lengthy sequences due to its inherent short-term memory limitations. This limitation may hinder the model's ability to acquire and retain information across extensive input sequences, thereby affecting its overall performance on tasks requiring the storage of long-range dependencies.
6.2.3 GCN-Transformer

In the present experiment, we employ the GCN-Transformer model architecture as delineated in the preceding section. Due to the inclusion of GCN layers, it becomes imperative to transform both input and output sequences into corresponding graphs and subsequently generate their adjacency matrices. These matrices will serve as inputs to the encoder, functioning analogously to the embedding matrices typically generated for the initial layer of the experimental models under consideration. This adaptation ensures that the GCN-Transformer model can effectively process and learn from the graph-based input data structure. In Figure 40 the code describes the approach used to convert input or output sequences into a graph representation.

```python
def create_graph(tokenized_sequences):
    G = nx.DiGraph()
    for seq in tokenized_sequences:
        for i in range(len(seq) - 1):
            G.add_edge(seq[i], seq[i + 1])
    return G

input_graph = create_graph(input_sequences)
output_graph = create_graph(output_sequences)

def graph_to_adj_matrix(G):
    return nx.to_numpy_array(G)

input_adj_matrix = graph_to_adj_matrix(input_graph)
output_adj_matrix = graph_to_adj_matrix(output_graph)
```

Figure 40: Convert Sequences to Graph

As the sequences are encoded prior to being processed by the function that generates graphs, it is difficult to visualize the output of this function. To illustrate the fundamental principles of our methodology, the accompanying Figure 41 depicts a sample sequence that is passed through the function prior to token encoding and tensor creation. This visualization is intended to aid comprehension of the experimental framework's data transformation processes.

Figure 41: Sequence Graph
6.2.3.1 Results and Observations

![Graph showing Accuracy over epochs and Loss over epochs for GCN-Transformer](image)

Figure 42: GCN-Transformer Accuracy and Loss

In Figure 42, we visualize the output of the accuracy and loss of this model per epoch. This model, despite displaying an improvement in training time per epoch, cannot demonstrate a significant improvement in output sequence prediction. This may be because sentences are not inherently graphical data, but their relationship is seq2seq, which may result in data loss when converted to graphs. It has a validation accuracy of 75.33%. The figure below illustrates the error rates achieved with this experiment.

![Graph showing Error Rate over epochs for GCN-Transformer](image)

Figure 43: GCN-Transformer WER and SER

6.2.4. GCNN-Transformer

In this experiment, we implement the GCNN model delineated in the preceding section, utilizing encoded input sequences as the input. Due to computational limitations, we constrain the number of epochs to 5. We modified the approach by substituting the GCN layers with GCNN layers, which have been established in previous research to demonstrate superior performance in comparison. This adapted approach results in substantial improvements in the model's performance, even with the restricted number of epochs. Additionally, we investigate the impact of altering the number of stacked GCNN layers and determine that an arrangement of three layers produces the most advantageous outcomes. A set of 3 stacked GCNN layers yield the best results.
6.2.4.1 Results and Observations

The results for the fourth model in our experimentation are illustrated above in the Figure 44 above to show the accuracy and loss over time. Transformer's architecture is predicated on self-attention mechanisms that allow the model to selectively concentrate on distinct input segments. GCNNs can capture both local and global data dependencies. By combining these two methods, a robust instrument for dealing with sequential data, such as text-to-speech normalization, can be created. Figure 45 shows the error rates achieved for GCNN-Transformer model.

![Figure 44: GCNN-Transformer Accuracy and Loss](image)

The Table 7 shows a comparative depiction of the results of GCN and GCNN based Transformers.

<table>
<thead>
<tr>
<th>Model</th>
<th>CNN Type</th>
<th>Accuracy</th>
<th>WER</th>
<th>SER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>GCN</td>
<td>75.33%</td>
<td>12%</td>
<td>8.3%</td>
</tr>
<tr>
<td></td>
<td>GCNN</td>
<td>85%</td>
<td>11%</td>
<td>4.6%</td>
</tr>
</tbody>
</table>

6.2.5. Reformer

We implement the Reformer using the LMHead model architecture, as described in the Reformer implementation section. The model's inputs are identical to those of GCNN-Transformer. Using this design, we investigate the effect of Byte Level and BERT tokenizers on the overall normalization
outcomes. In addition, we experiment with varying the activation function hyperparameters, specifically employing 'ReLU' and 'Softmax' because both are appropriate for a multi-label classifier, similar to our language modeling task.

### 6.2.5.1 Byte Level Tokenizer

Byte-Pair Encoding (BPE) is a commonly used data compression algorithm in natural language processing (NLP) to generate subword representations of text data. The algorithm operates by repeatedly integrating the most common pairs of bytes or characters in a corpus until the desired vocabulary size is reached. Consequently, the derived subword vocabulary makes encoding and decoding text data more compact and efficient.

We construct a tokenizer utilizing BPE-based layers from the Hugging Face Tokenizers library, train and save the tokenizer on our dataset, and then evaluate the performance of the tokenizer. The following is a succinct summary of the tokenization technique that employs BPE for Reformer input:

- Create a Tokenizer object utilizing a BPE model.
- Preprocess the input data by configuring the pre-tokenizer, decoder, and normalization.
- Set the parameters for training, including the training file(s), vocabulary capacity, minimum frequency, and special tokens.
- The tokenizer is trained using the BPETrainer with the specified parameters, while progress is displayed.
- Save the trained tokenizer to a file in the JSON format. In Figure 46, we demonstrate the code used to implement the BPE Tokenizer.

```python
from tokenizers import Tokenizer, models, pre_tokenizers, decoders, trainers, normalizers

# Create a tokenizer with a Byte-Pair Encoding (BPE) model
tokenizer = Tokenizer(models.BPE())

# Set up pre-tokenizer, decoder, and normalization
tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel()  
tokenizer.decoder = decoders.ByteLevel()  
tokenizer.normalizer = normalizers.Sequence([normalizers.to_lowercase(), normalizers.strip_accents()])

# Define the training parameters
training_files = ['/en_train.csv'] # Replace with your dataset file(s)
 vocab_size = 32000
 min_freq = 2
 special_tokens = ['<PAD>', '<UNK>', '<EOS>', '<UNK>']

# Train the tokenizer
trainer = trainers.BPETrainer(vocab_size=vocab_size, min_freq=min_freq, special_tokens=special_tokens, show_progress=True)
trainer.train(training_files, tokenizer)

# Save the tokenizer
tokenizer.save('tokenizer.json')
```

**Figure 46: BPE-Tokenizer**

### 6.2.5.2 BERT Tokenizer

The BERT tokenizer is based on WordPiece tokenization, an algorithm similar to BPE for tokenizing subwords. The following is a brief description of the architecture -

The input text is initially segmented into individual words using whitespace and punctuation as
delimiters for WordPiece Tokenization. Each word is then divided into a series of subwords using the
WordPiece algorithm, which divides the most frequent subword pairs in a corpus iteratively until a
predetermined vocabulary size is reached. By decomposing out-of-vocabulary words into smaller
subword units, the resultant subword vocabulary can represent them.

- Special Tokens: The tokenizer also inserts special tokens into the input sequence, such as a
  [CLS] token at the beginning of the sequence to indicate the beginning of a new sentence and
  a [SEP] token between sentences to separate them.
- Masking and Padding: The tokenizer conceals particular tokens in the input sequence, such as
  the [MASK] token used during training to predict missing words, and pads the sequence with
  [PAD] tokens to maintain a fixed length.
- Each subword in the input sequence is mapped to a unique integer ID within the tokenizer's
  vocabulary. This mapping permits the input sequence to be encoded as a sequence of integer
  identifiers that can be fed into a neural network for further processing.

We conduct experiments using both BPE and BERT tokenizers as input to our model. The code in
Figure 47 is used to load the pre-trained BERT tokenizer from the HuggingFace library.

![Figure 47: BERT Pre-trained Tokenizer](image)

### 6.2.5.3 ReLu Activation

In the discipline of deep learning and neural networks, the Rectified Linear Unit (ReLU) is a widely
used activation function. It is a simple nonlinear function that incorporates nonlinearity into the network,
allowing it to learn complex data patterns and relationships. The mathematical definition of the ReLU
function is:

\[ f(x) = \max(0, x) \]

where x represents the function's input. The function returns the greatest value between 0 and the input
value x. Essentially, ReLU replaces all negative input values with 0, while positive input values are
unaffected.

The advantages of the ReLU activation function are its simplicity and computational efficacy. It
mitigates the vanishing gradient issue typically encountered in deep neural networks with saturating
activation functions such as sigmoid or hyperbolic tangent.
6.2.5.4 Softmax Activation

Softmax is a prominent activation function for multi-class classification issues in deep learning and neural networks. It transforms a vector of real numbers into a probability distribution, ensuring that the output values lie between (0, 1) and sum to 1. Given the vector \( x \) as input, the Softmax function is defined as:

\[
\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum\exp(x_j)}
\]

where \( x_i \) is the \( i \)-th element of the input vector \( x \), \( \exp \) represents the exponential function, and \( \sum\) is a summation over all vector elements \( x_j \).

Softmax transforms the input values so they can be interpreted as class probabilities. The output of the function is a probability distribution over the classes, with the class corresponding to the highest probability being the predicted class.

6.2.5.5 Results and Observations

![Figure 48: Reformer Accuracy and Loss](image)

Figure 48 depicts the change in the values of accuracy and loss over 10 epochs for the Reformer model. Similarly, Table 8 presents the accuracy and error rates for each of our experiments using different tokenizers and activation function combinations.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tokenizer</th>
<th>Activation</th>
<th>Accuracy</th>
<th>WER</th>
<th>SER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reformer</td>
<td>BPE</td>
<td>ReLu</td>
<td>93%</td>
<td>8%</td>
<td>7.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Softmax</td>
<td>90.01%</td>
<td>8.18%</td>
<td>7.11%</td>
</tr>
<tr>
<td></td>
<td>BERT</td>
<td>ReLu</td>
<td>96%</td>
<td>5.22%</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Softmax</td>
<td>92%</td>
<td>6.33%</td>
<td>6.23%</td>
</tr>
</tbody>
</table>

6.2.6. Transfer Learning with BERT

In our research, we explore a pre-trained BERT Transformer model, which has undergone training on
English language data. Transfer learning, a machine learning technique where a pre-existing model is used as the basis for a new model on a different task, is employed in this architecture. Utilizing pre-trained models as the starting point for natural language processing tasks is a common strategy in deep learning, given the substantial computing and time resources required to develop neural network models for such problems, as well as the significant advancements in performance that they offer for related problems. The pre-trained BERT generates the results shown in Table 9 and Figure 49 illustrates the accuracy and error rates per epoch in the graph -

<table>
<thead>
<tr>
<th>Model</th>
<th>Tokenizer</th>
<th>Accuracy</th>
<th>WER</th>
<th>SER</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Transformer</td>
<td>BERT</td>
<td>95.83%</td>
<td>4.5%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Figure 49: BERT Accuracy, WER and SER

6.2.7. Output to Audio Conversion

In this section, the Reformer model is used to predict the output sequences of sentences provided as input. We select example sentences from the validation dataset and convert them to speech using the pyttsx3 library. Unlike other libraries, pyttsx3 functions offline and is compatible with both Python 2 and Python 3. Figure 50 shows how we use the best performing model of our study to generate an audio output for the text-to-speech normalization output.
The generated audio for each sentence is then saved as a .wav file. We utilize the scipy.io library, which provides functions for reading .wav files, to visualize the audio waves. The resulting waveforms provide a representation of the audio produced by the Reformer model, allowing us to analyze the text-to-speech normalization performance in greater detail. Figure 51 illustrates the sample .wav file generated during experimentation.
7. RESULTS AND ANALYSIS

7.1 Model Comparison

7.1.1 Result with Wikipedia Dataset

The findings from the conducted experiments indicate that BERT achieved the best overall performance. However, it should be noted that BERT was trained on a vast amount of external data that is beyond the scope of our dataset. Conversely, the Reformer model exhibited significantly better performance when trained solely on our dataset due to its unique design and improvisational ability provided by the LM Head. The Reformer model demonstrated superior learning of the language grammar in the absence of pre-training. Table 10 summarizes the results of all the models evaluated in the experiments on the Wikipedia dataset.

Table 10: Results with Wikipedia Dataset

<table>
<thead>
<tr>
<th>Language Model</th>
<th>Tokenizers</th>
<th>Accuracy</th>
<th>WER (%)</th>
<th>SER (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM-GRU</td>
<td>Keras</td>
<td>97% (overfit)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GCN-Transformer</td>
<td>Keras</td>
<td>75.33%</td>
<td>12%</td>
<td>8.3%</td>
</tr>
<tr>
<td>GCNN-Transformer</td>
<td>Keras</td>
<td>85%</td>
<td>11%</td>
<td>4.6%</td>
</tr>
<tr>
<td>Reformer</td>
<td>BPE</td>
<td>90.01%</td>
<td>8.18%</td>
<td>7.11%</td>
</tr>
<tr>
<td></td>
<td>BERT</td>
<td>96%</td>
<td>5.22%</td>
<td>4%</td>
</tr>
<tr>
<td>BERT-Transformer</td>
<td>BERT</td>
<td>95.83%</td>
<td>4.5%</td>
<td>3%</td>
</tr>
</tbody>
</table>

The Reformer model, with its BERT tokenizer and capacity to handle long sequences of text, is particularly well-suited for managing large datasets. Additionally, the Reformer model's ability to process noisy data with low WER and SER makes it an efficient option for natural language processing tasks. The parallelizable architecture of the Reformer model enables rapid processing of large datasets, further increasing its utility in practical applications.

In summary, while the BERT model achieved the best overall performance, the Reformer model offers several significant advantages when trained exclusively on our dataset, such as superior learning of the language grammar in the absence of pre-training, capacity to manage long sequences of text, efficiency in processing noisy data, and rapid processing of large datasets.

7.1.2 Result with Twitter Dataset

The significant performance decline of all models when applied to the Twitter dataset is a noteworthy finding from our experiments. This decline in performance is primarily attributable to Twitter users'
frequent use of non-standard English words and slang, which presents a significant challenge for the models. Therefore, the models require training not only on standard English but also on Twitter-specific language, such as vernacular and the correct pronunciations for acronyms, a class that was absent from our original datasets. Table 11 summarizes the results of all the models evaluated in the experiments on the Twitter dataset.

Table 11: Results with Twitter Dataset

<table>
<thead>
<tr>
<th>Language Model</th>
<th>Tokenizers</th>
<th>Accuracy</th>
<th>WER (%)</th>
<th>SER (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM-GRU</td>
<td>Keras</td>
<td>97% (overfit)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GCN-Transformer</td>
<td>Keras</td>
<td>72%</td>
<td>23%</td>
<td>26%</td>
</tr>
<tr>
<td>GCNN-Transformer</td>
<td>Keras</td>
<td>71%</td>
<td>19.77%</td>
<td>15.12%</td>
</tr>
<tr>
<td>Reformer</td>
<td>BPE</td>
<td>70%</td>
<td>17.01%</td>
<td>~16.85%</td>
</tr>
<tr>
<td></td>
<td>BERT</td>
<td>74%</td>
<td>17%</td>
<td>15.8%</td>
</tr>
<tr>
<td>BERT-Transformer</td>
<td>BERT</td>
<td>90%</td>
<td>8.2%</td>
<td>7.5%</td>
</tr>
</tbody>
</table>

The informal nature of Twitter posts presents a unique challenge for natural language processing models posed by the characteristics of the Twitter dataset, as it deviates significantly from standard written English. To obtain optimal performance on the Twitter dataset, models must be trained on a dataset that captures the nuances of social media specific language, including acronyms and slang. Future research should concentrate on the creation of large-scale datasets consisting of informal language for training models that can show improved performance on the Twitter dataset.

7.2 Model Analysis

7.2.1 Advantages of Gated CNN

Graph Convolutional Neural Networks (GCNNs) offer several benefits over Graph Convolutional Networks (GCNs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks for text-to-speech (TTS) normalization tasks:

- **Ability to handle variable-length input**: GCNNs can handle variable-length input by processing graphs with varying number of nodes, while RNNs and LSTMs are typically designed for fixed-length sequences.
- **Efficient processing of large graphs**: GCNNs use spatially-localized filters that can efficiently operate on large graphs, while RNNs and LSTMs require sequential processing of the input, which can be slow and computationally expensive.
- **Improved accuracy**: GCNNs have been shown to achieve state-of-the-art performance on a wide
range of graph-based tasks, including TTS normalization, due to their ability to capture complex relationships between nodes in a graph.

- **Robustness to noise and missing data**: GCNNs are designed to handle noisy and incomplete data, which is common in TTS normalization tasks where transcription errors and missing data can occur.

### 7.2.2 Advantages of Reformer

In this research, the Reformer exhibits several advantages over other transformers and BERT for TTS normalization:

- **Improved scalability**: The Reformer's use of locality-sensitive hashing (LSH) enables it to handle long sequences more efficiently than other transformers, making it particularly well-suited for TTS normalization tasks where the input sequences can be very long.

- **Memory efficiency**: The Reformer's use of reversible layers allows it to process very long sequences with limited memory resources, which can be a critical advantage in resource-constrained environments.

- **Better handling of sequential data**: The Reformer's positional embeddings are learned dynamically based on the input data, which allows it to handle sequential data more effectively than other transformers that rely on fixed positional embeddings.

- **Improved accuracy**: The Reformer has been shown to achieve state-of-the-art performance on a wide range of natural language processing tasks, including TTS normalization, due to its ability to capture long-term dependencies and handle very long sequences.

- **Faster training and inference**: The Reformer's efficient use of memory and dynamic positional embeddings can significantly reduce training and inference times compared to other transformers including BERT, especially for pre-training it on a large corpus.
8. CONCLUSION AND FUTURE WORK

In this research, several language models with various tokenizers, including LSTM-GRU, GCN-Transformer, GCNN-Transformer, Reformer and BERT, were evaluated. The LSTM-GRU and GCN-Transformer models utilized the Keras tokenizer, while the Reformer model utilized the Byte Pair Encoding (BPE) and BERT tokenizer. The BERT-Transformer models were implemented using a transformer-based architecture with pre-trained data.

The LSTM-GRU model obtained the highest accuracy of 97%, according to our findings. However, it was observed to overfit the data, resulting in insufficient generalization to new data. The GCN-Transformer model obtained an accuracy of 75.33 percent, which is lower than the accuracy of other models in the study. It performed better in terms of Word Error Rate (WER) and Sentence Error Rate (SER), proving it to be appropriate for graph-based data.

The GCNN-Transformer model, a modification of the GCN-Transformer with gated convolutional layers stacked in the encoder, outperformed the GCN-Transformer with an accuracy of 85%. The Reformer model obtained an accuracy of 90.01 percent with BPE Tokenizer and had a low WER and SER, which made it more effective for processing large datasets. The Reformer model with BERT Tokenizer was the most accurate, with a 96% accuracy rate and the lowest WER and SER of all models. BERT is a powerful tool for natural language processing tasks due to its capacity to construct contextual embeddings that capture the meaning of words in a sentence.

In terms of WER and SER, the BERT-Transformer model, which is a modified version of the original BERT model, obtained slightly lower accuracy than the Reformer but still outperformed the majority of other models in the study. In conclusion, the GCNN-Transformer, Reformer, and BERT models performed well in this evaluation, with each model possessing a distinct advantage in handling various categories of data and tasks.

8.1 Challenges and future works

Our experiments have identified several key challenges in converting social media text to speech using sophisticated models like Reformers. While standard English language text can be readily normalized into speech using these models, the language used on social media platforms such as Twitter differs significantly from standard English and does not conform to the same grammar and language rules. Therefore, it is crucial to treat social media language as a separate language when modeling classes and rules for it.

The language used in social media posts poses a unique challenge for natural language processing models, as the informal nature of social media posts deviates considerably from standard written English. Consequently, it is essential to develop models that can capture the nuances of social media language, including acronyms and slang. To achieve optimal performance in converting non-standard language
forms to comprehensive speech that does not sound odd, it is critical to develop models that can capture the specific characteristics of social media language.

Future research should focus on developing models that can handle social media language more effectively. This could involve creating large-scale social media-specific datasets and developing models that can capture the nuances of social media language, including informal grammar and language rules. Additionally, researchers should explore the use of transfer learning techniques that leverage pre-trained models to improve the performance of social media language models.
REFERENCES


