Innovating Multilingual Communication: NLP for Thanglish Interpretation

A PROJECT WORK I REPORT

Submitted by

DINESH KUMAR M
21ALR016

DHANUSH G
21ALR012

VIGNESH R
21ALL065

in partial fulfilment of the requirements

for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE-638 060

MAY 2024

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE – 638 060

MAY 2024

BONAFIDE CERTIFICATE

This is to certify that the Project Report titled **Innovating Multilingual Communication: NLP for Thanglish Interpretation** is the bonafide record of project work done by **DINESH KUMAR M (21ALR016)** , **DHANUSH G (21ALR012)** , **VIGNESH R (21ALL065)** in partial fulfilment of the requirements for the award of Degree of Bachelor of Technology in Artificial Intelligence of Anna University, Chennai during the year 2023-2024.

SUPERVISOR **HEAD OF THE DEPARTMENT**
(Signature with seal)

Date:

Submitted for the end semester viva voce examination held on_____.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE – 638 060

MAY 2024

DECLARATION

We affirm that the Project Report titled **Innovating Multilingual Communication: NLP for Thanglish Interpretation** being submitted in partial fulfilment of the requirements for the award of Bachelor of Technology is the original work carried out by us. It has not formed part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Date:

DINESH KUMAR M
(21ALR016)

DHANUSH G
(21ALR012)

VIGNESH R
(21ALL065)

I certify that the declaration made by the above candidate is true to the best of my knowledge.

Name & Signature of the Supervisor with seal

Date:

ABSTRACT

In linguistics, Thanglish refers to the hybrid language that combines Tamil and English elements. With the advent of technology and the increasing usage of social media platforms, thanglish has become prevalent in online communication. This abstract presents a method for converting Thanglish sentences into English using Natural Language Processing (NLP) techniques. Firstly, the thanglish sentence is tokenized, breaking it down into individual words or phrases. Then, each token is tagged with its corresponding part of speech using a pre-trained NLP model. This step helps in identifying the grammatical structure of the sentence. Following tagging, a translation algorithm is employed to convert each Tamil word or phrase into its English equivalent. This translation algorithm utilizes a bilingual dictionary or word embeddings trained specifically for Tamil and English languages. The algorithm takes into account contextual information to ensure accurate translations. Additionally, the algorithm handles code-switching phenomena commonly found in thanglish, where speakers alternate between Tamil and English within the same sentence. By analyzing the context and linguistic patterns, the algorithm determines the appropriate language for each token and translates it accordingly. Moreover, to enhance the accuracy of the translation, the algorithm considers common Thanglish expressions and colloquialisms. It utilizes sentiment analysis techniques to capture the intended emotions or tones conveyed in the original Thanglish sentence. Furthermore, the system incorporates a feedback mechanism where users can provide corrections or suggestions for improving the translation quality. This feedback loop helps in continuously refining the NLP model and increasing its effectiveness over time. In conclusion, the proposed method offers a solution for converting thanglish sentences into English by leveraging NLP technologies.

ACKNOWLEDGEMENT

First and foremost, we acknowledge the abundant grace and presence of the almighty throughout different phases of the project and its successful completion.

We wish to express our sincere gratitude to our honorable Correspondent **Thiru.A.K.ILANGO B.Com.,M.B.A.,LLB.,** and other trust members for having provided us with all necessary infrastructures to undertake this project.

We extend our hearty gratitude to our honorable Principal **Dr.V.BALUSAMY B.E.(Hons)., MTech., Ph.D.,** for his consistent encouragement throughout our college days.

We would like to express our profound interest and sincere gratitude to our respected Head of the department **Dr.C.S.KANIMOZHISELVI ME., Ph.D.,** for his valuable guidance.

A special debt is owed to the project coordinator **Ms. P. JAYADHARSHINI B.Tech., M.Tech.**. Assistant Professor, Department of Artificial Intelligence for their encouragement and valuable advice that made us to carry out project work successfully.

We extend our sincere gratitude to our beloved guide **Ms. R. THANGAMANI B.Tech., M.E.,** Department of Artificial Intelligence for her ideas and suggestions, which have been very helpful to complete the project.

We are grateful to all the faculty and staff members of the Department of Artificial Intelligence and persons who directly and indirectly supported this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ML	:	Machine Learning
NLP	:	Natural Language Processing
LSTM	:	Long Short-Term Memory
GRU	:	Gated Recurrent Unit
ROUGE	:	Recall – Oriented Understudy for Gisting Evaluation

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

In today's interconnected world, the evolution of language is a dynamic process influenced by cultural, social, and technological factors. One such linguistic phenomenon gaining prominence, particularly in digital communication, is Thanglish—a unique blend of Tamil and English. Stemming from the cultural diversity of regions where both languages are spoken, Thanglish represents a fusion of linguistic elements, enabling speakers to convey ideas with a rich tapestry of expressions.

As Thanglish permeates online platforms, from social media interactions to informal conversations, the need for tools to bridge the gap between this hybrid language and standard English becomes increasingly evident. Natural Language Processing (NLP), a branch of artificial intelligence concerned with the interaction between computers and human language, offers promising avenues for addressing this linguistic challenge.

This abstract presents a method for converting Thanglish sentences into English using NLP techniques. By harnessing the power of computational linguistics, this approach aims to facilitate seamless communication across linguistic boundaries, enhancing accessibility and comprehension in digital discourse. Through a combination of tokenization, part-of-speech tagging, translation algorithms, and sentiment analysis, the proposed method seeks to accurately capture the nuances of Thanglish expressions while providing fluent and contextually appropriate English translations.

In the following sections, we will delve into the intricacies of Thanglish and explore the mechanisms underlying our NLP-based approach for converting Thanglish sentences to English. By elucidating the methodology and potential applications of this transformative technology, we aim to contribute to the advancement of multilingual communication in the digital age.

1.2 OBJECTIVE

- Building a robust NLP model to accurately parse Thanglish sentences, recognizing linguistic features and code-switching.
- Implementing translation algorithms using bilingual dictionaries, word embeddings, and contextual analysis for fluent English conversions.
- Integrating sentiment analysis to capture emotions and cultural nuances, ensuring culturally sensitive translations.
- Establishing a feedback mechanism for users to provide corrections and suggestions, enabling continual refinement of the NLP model for enhanced accuracy and adaptability to evolving language patterns.

1.3 SCOPE

- Develop a robust NLP system for accurately parsing and translating Thanglish sentences into grammatically correct English.
- Integrate sentiment analysis techniques to ensure cultural sensitivity in translated outputs.
- Establish a user feedback mechanism for continual refinement and improvement of the NLP model.

CHAPTER 2

SYSTEM ANALYSIS

2.1 LITERATURE REVIEW

"Investigation of Methods to Improve the Recognition Performance of Tamil-English Code-Switched Data in Transformer Framework": Explores techniques to enhance recognition of code-switched Tamil-English speech using Transformer models.

"Multilingual speech recognition with a single end-to-end model": Demonstrates a unified approach for recognizing speech in multiple languages using a single end-to-end model.

"Listen, attend and spell: A neural network for large vocabulary conversational speech recognition": Introduces a neural network architecture for accurately transcribing conversational speech with large vocabularies.

"Investigating end-to-end speech recognition for Mandarin-English code-switching": Investigates methods for improving end-to-end speech recognition performance in Mandarin-English code-switched speech.

"Language-independent end-to-end architecture for joint language identification and speech recognition": Presents an architecture capable of jointly identifying languages and recognizing speech without language-specific preprocessing.

"Towards end-to-end code-switching speech recognition": Advances research towards achieving end-to-end code-switching speech recognition systems.

"An end-to-end language-tracking speech recognizer for mixed-language speech": Develops a language-tracking speech recognizer designed to handle mixed-language speech scenarios.

"Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks": Introduces the Connectionist Temporal Classification (CTC) method for labeling unsegmented sequential data, particularly in the context of speech recognition.

2.2 SUMMARY

The research landscape in speech recognition has seen significant advancements, as evidenced by a series of studies focusing on various aspects of this domain. "Investigation of Methods to Improve the Recognition Performance of Tamil-English Code-Switched Data in Transformer Framework" targets the enhancement of recognition accuracy in code-switched Tamil-English speech, employing Transformer models. Meanwhile, "Multilingual speech recognition with a single end-to-end model" offers a unified approach to speech

recognition across multiple languages, streamlining the process with a single end-to-end model. "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition" introduces a neural architecture specialized in accurately transcribing conversational speech containing extensive vocabularies. Moreover, "Investigating end-to-end speech recognition for Mandarin-English code-switching" explores methodologies to improve end-to-end speech recognition in Mandarin-English code-switched contexts. Additionally, "Language-independent end-to-end architecture for joint language identification and speech recognition" presents an architecture capable of language identification and speech recognition without language-specific preprocessing steps. These studies collectively contribute to advancing the field towards more robust and versatile speech recognition systems capable of handling diverse linguistic scenarios with high accuracy and efficiency.

CHAPTER 3
SYSTEM REQUIREMENTS

3.1 HARDWARE REQUIREMENTS

CPU type	:	Intel corei5 processors
Ram size	:	16 GB
Hard disk capacity	:	512 GB

3.2 SOFTWARE REQUIREMENTS

Operating System	:	Windows 11
Language	:	Python (version 3+)
Tool	:	Google Colab

3.3 SOFTWARE DESCRIPTION

3.3.1 Python

Python is a powerful programming language that is simple to learn. Its extensive use of information structures and straightforward but effective approach to handling object-oriented programming. Python is a perfect language for prearranging and swift application advancement in many fields in the best stages due to its refined phrase structure and imperative composing, which are close to its made sense nature. The core ideas and capabilities of Python 3 are covered in this essay. A large number of the functions that come with Python when it is installed make up its standard library. There are numerous additional libraries that the Python language can use to experiment with more things available online. These libraries provide it strength and enable it to do a wide range of tasks. In the provided Python code snippets, several essential libraries and methods are employed to perform a range of tasks. The Pandas library is utilized for data manipulation and analysis, particularly for reading and processing data from CSV files, enabling data preprocessing. Scikit- Learn (sklearn) comes into play as a machine

learning library, assisting in model selection, data splitting for training and testing, and the calculation of accuracy scores.

3.3.2 Google Colab Notebook

Google Colab, short for Google Colaboratory, is a cloud-based platform that provides free access to Jupyter notebooks, making it easy for users to run and share code. It offers a GPU and TPU-powered environment for data analysis, machine learning, and deep learning tasks. Users can collaborate in real-time and store their work in Google Drive. Colab notebooks can be shared in the same way as Sheets or Google Docs. Either click the Share button located at the upper right corner of any Colab notebook, or adhere to these guidelines for sharing files from Google Drive. Google Drive search is available for Colab notebooks. All notebooks in Drive will be visible when you click the Colab logo in the upper left corner of the notebook view. By selecting File > Open notebook, you can also look for notebooks that you have recently opened. Your account-only virtual machine is used to run code. Virtual machines have a maximum lifetime that is enforced by the Colab service, and they are deleted after being inactive for some time. Any Colab notebook you've created can be downloaded by following these guidelines, from Google Drive, or it can be accessed via the File menu in Colab.

CHAPTER 4

PROPOSED SYSTEM

4.1 SYSTEM ARCHITECTURE

The system architecture begins with data collection from external sources, followed by preprocessing to clean and normalize the raw text data. Tokenization and vocabulary building are then performed to convert the text into numerical sequences and construct dictionaries for both Tamil and English languages. These sequences are padded to ensure uniform length, facilitating input to the neural network models. The models, including LSTM, GRU, and Transformer architectures, are trained on the padded sequences to learn the mapping between Tamil and English sentences. After training, model evaluation is conducted using validation data to assess accuracy and loss. Finally, the trained models are utilized for translation tasks, where input Tamil sentences undergo tokenization, padding, and translation to produce corresponding English translations. This architecture illustrates the sequential flow of data and processes, from data ingestion to translation output, within the system.

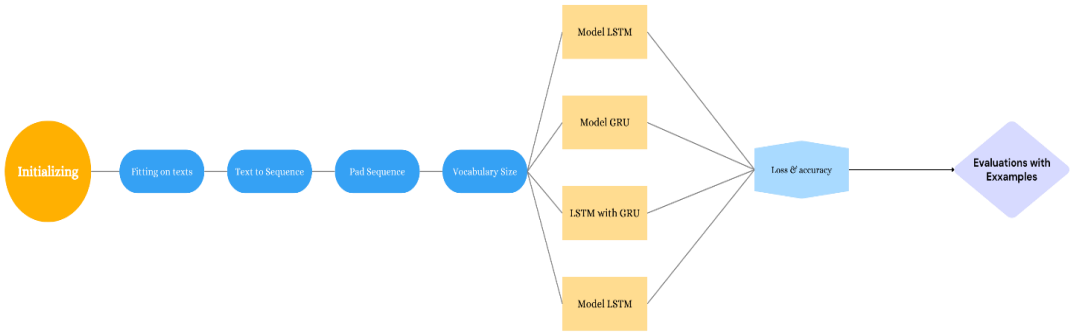

Fig. 4.1 Proposed model workflow

The heart of the system lies in the neural network models, which encompass architectures such as LSTM, GRU, and Transformer. These models are trained on the padded sequences to learn intricate patterns and relationships between Tamil and English sentences, enabling accurate translation. Throughout the training phase, model evaluation is conducted using validation data to assess performance metrics such as accuracy and loss, thereby optimizing the models' efficacy.

Once trained, the models are deployed for translation tasks, where input Tamil sentences undergo tokenization, padding, and translation to produce corresponding English translations. This translation pipeline encapsulates the essence of the system, seamlessly bridging the gap between different languages. Overall, the architecture embodies a holistic approach towards machine translation, integrating data preprocessing, model training, evaluation, and deployment to deliver accurate and efficient translations.

4.2 MODULE DESCRIPTION

4.2.1 Dataset Collection

In the creation of a Thanglish to English translation dataset, a diverse range of Thanglish sentences was gathered from various sources like social media, forums, and written texts, forming a sizable dataset. Each Thanglish sentence was meticulously paired with its English translation through either manual translation by bilingual speakers or automatic translation followed by human verification. This dataset was structured into a table format, typically in an Excel sheet or CSV file, comprising two columns representing Thanglish sentences and their corresponding English translations. The dataset's characteristics, including sentence lengths, expression types, and linguistic challenges encountered during annotation, were thoroughly examined.

4.2.2 Data pre-processing

The preprocessing techniques employed in the provided code are integral for preparing raw text data for subsequent processing and model training. Lowercasing ensures uniformity by converting all letters to lowercase, thereby standardizing the text and reducing vocabulary size. Tokenization segments the text into smaller units like words or sub words, facilitating vocabulary creation and sequence indexing. Through padding, sequences are adjusted to uniform length, crucial for neural network input. These techniques collectively streamline the text data, making it suitable for machine learning tasks, maintaining sequence consistency, and establishing the foundation for effective model training and translation tasks.

- **Tokenizer:** A preprocessing module from Keras in TensorFlow used for tokenizing text data. It converts text into numerical sequences by assigning a unique integer to each word in the corpus.
- **Pad Sequences:** A function from Keras in TensorFlow used for padding sequences to ensure uniform length. It adds zeros to sequences shorter than the maximum length, making them all the same length.
- **Lowercase:** Lowercasing helps to standardize the text by removing variations caused by different capitalizations of the same word.

4.2.3 Implementation of NLP model Architecture

4.2.3.1 LSTM

The LSTM model for Tamil-English translation employs embedding, bidirectional LSTM layers, dropout, and time-distributed dense layers. Embedding converts input integers to dense vectors. Bidirectional LSTM captures context, with the first layer outputting sequences and the second processing them. Dropout prevents overfitting, and time-distributed dense predicts the next word. Compiled with Adam optimizer and sparse categorical cross-entropy loss, it effectively translates between Tamil and English.

$$pt = p_t^f + p_t^b$$

pt : Final probability vector of the network.
p_t^f : Probability vector from the forward LSTM network.
p_t^b : Probability vector from the backward LSTM network.

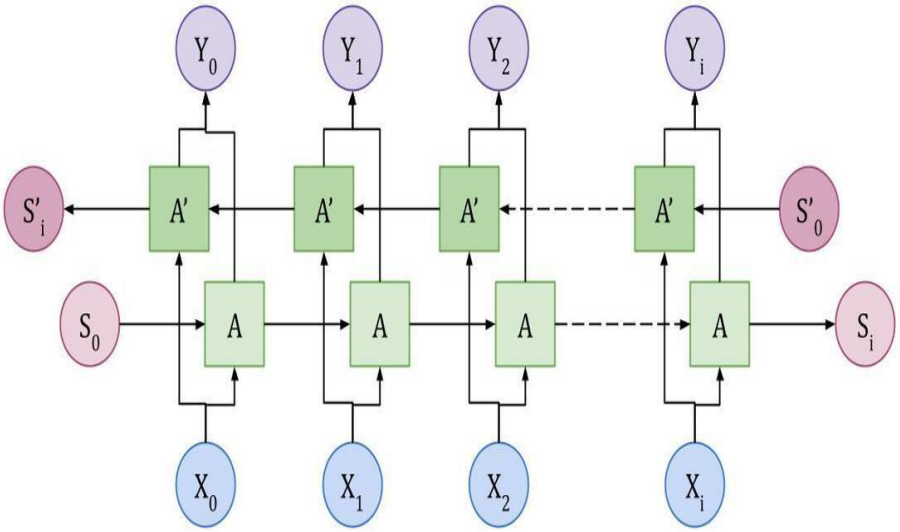

Fig 4.2 Bidirectional LSTM layer Architecture

4.2.3.2 Gated Recurrent Unit Networks (GRU)

In the GRU model architecture, a similar sequence of layers is employed to facilitate translation tasks. Commencing with an embedding layer, the model converts input sequences into dense vectors, facilitating the learning of word embeddings specific to the vocabulary. Subsequently, two bidirectional GRU layers are applied, processing input sequences in both directions to capture comprehensive contextual information. Dropout layers are judiciously inserted to prevent overfitting by randomly deactivating a fraction of input units during training. A time-distributed dense layer follows, producing a probability distribution over the

vocabulary for each timestep, aiding accurate prediction of the subsequent word. Compiled with the Adam optimizer and sparse categorical cross-entropy loss function, this architecture empowers the GRU model to discern intricate patterns and relationships within the input data, facilitating precise translation between Tamil and English.

Reset gate: r_t = sigmoid(W_r * [h_{t-1}, x_t])

Update gate: z_t = sigmoid(W_z * [h_{t-1}, x_t])

Candidate hidden state: h_t' = tanh(W_h * [r_t * h_{t-1}, x_t])

Hidden state: h_t = (1 – z_t) * h_{t-1} + z_t * h_t'

where W_r, W_z, and W_h are learnable weight matrices, x_t is the input at time step t, h_{t-1} is the previous hidden state, and h_t is the current hidden state.

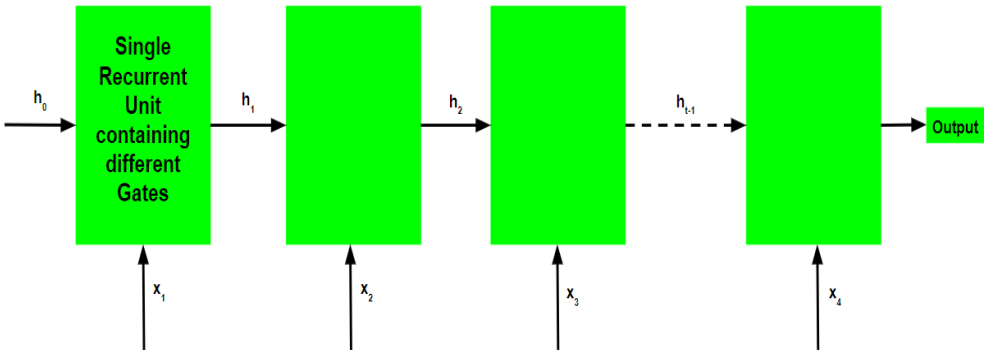

Fig 4.3 Work-Flow of a
Gated Recurrent Unit Network

4.2.3.3 Combined LSTM and GRU Model:

The combined LSTM and GRU model architecture amalgamates the strengths of both LSTM and GRU architectures to enhance translation capabilities. Initiated with an embedding layer, the model converts input sequences into dense vectors, facilitating the learning of word embeddings specific to the vocabulary. Subsequently, bidirectional LSTM and GRU layers are employed to process input sequences in both forward and backward directions, capturing comprehensive contextual information. Dropout layers are strategically incorporated to mitigate overfitting by randomly deactivating a fraction of input units during training. A time-distributed dense layer is then applied, generating a probability distribution over the vocabulary for each timestep, facilitating accurate prediction of the subsequent word. Compiled with the Adam optimizer and sparse categorical cross-entropy loss function, this architecture leverages the complementary strengths of LSTM and GRU models to discern intricate patterns and relationships within the input data, thereby facilitating precise translation between Tamil and English.

4.2.3.4 Transformer Model

The Transformer model architecture represents a paradigm shift in machine translation, eschewing recurrent neural networks in favor of a self-attention mechanism. This architecture dispenses with sequential processing and adopts a parallelized approach, enabling more efficient learning of long-range dependencies. Comprising multiple Transformer encoder layers, each layer employs multi-head self-attention and feedforward neural networks to process input sequences. Positional encoding layers are integrated to provide positional information to the model, enabling it to handle sequences without relying on recurrence. Dropout layers are incorporated to prevent overfitting during training. The model outputs a probability distribution over the vocabulary for each timestep, facilitating accurate prediction of the subsequent word. Compiled with the Adam optimizer and sparse categorical cross-entropy loss function, this architecture harnesses the power of self-attention mechanisms to discern intricate patterns and relationships within the input data, thereby facilitating precise translation between Tamil and English.

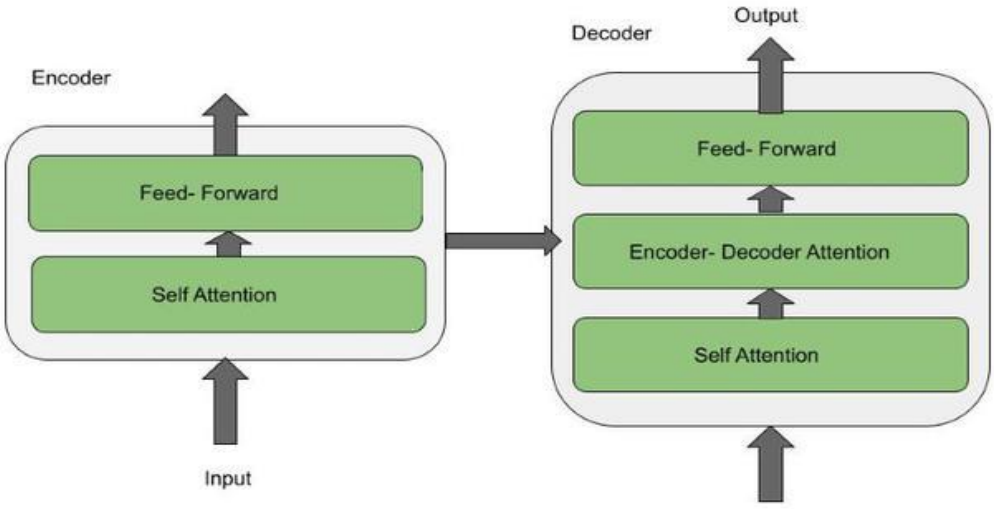

Fig 4.4 Encoder-Decoder Architecture

4.3 VISUALIZATION

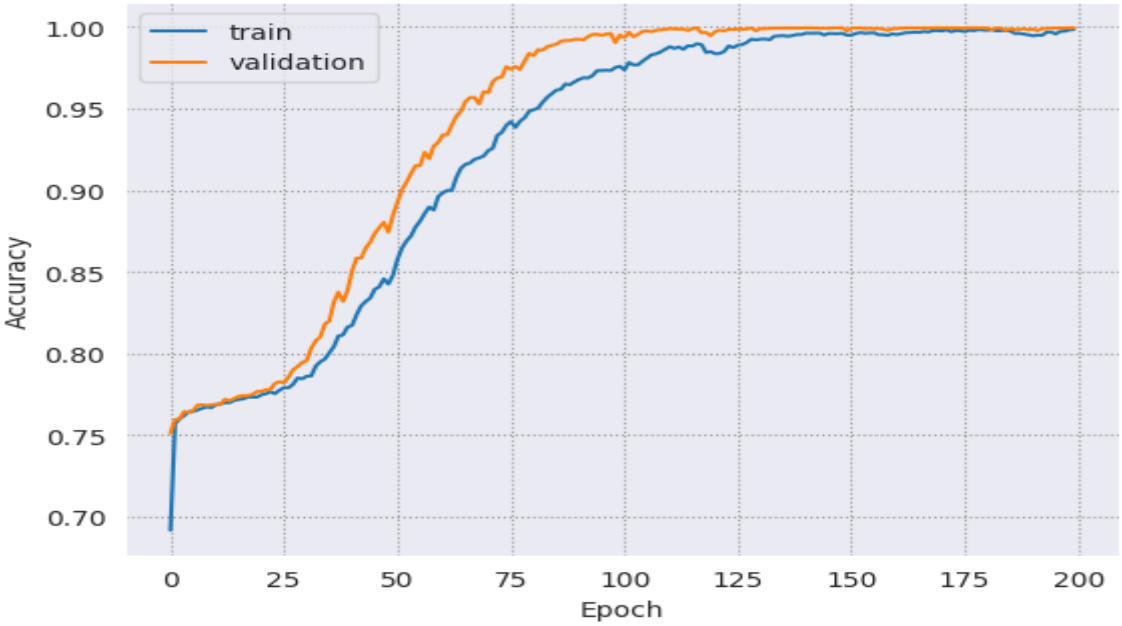

Fig 4.5 Accuracy curve of LSTM

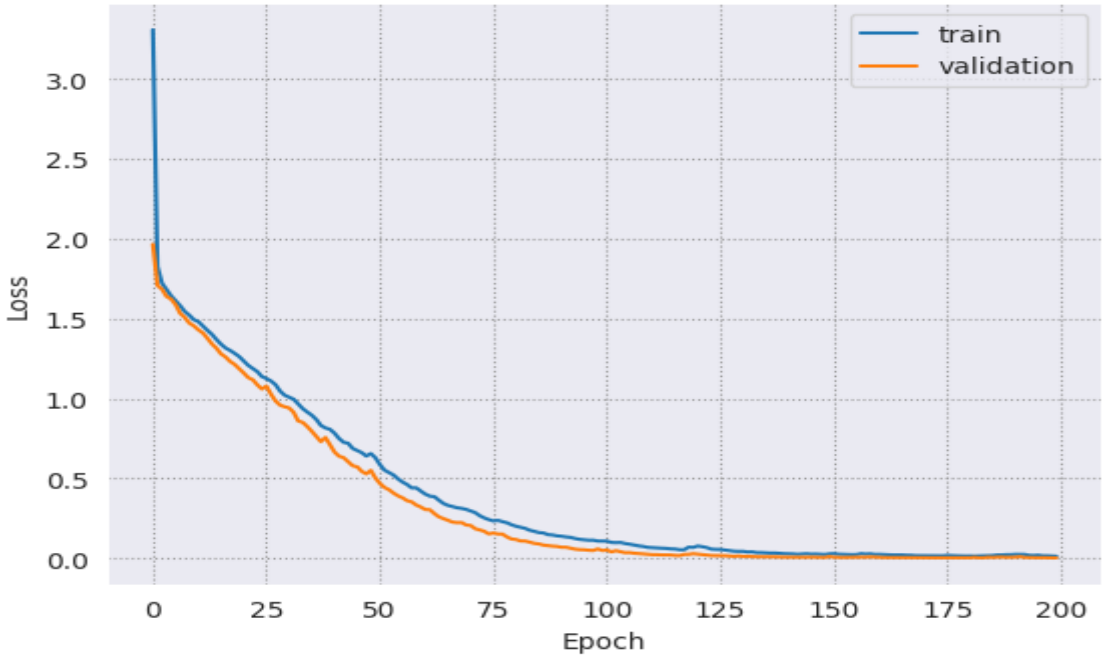

Fig 4.6 Loss curve of LSTM

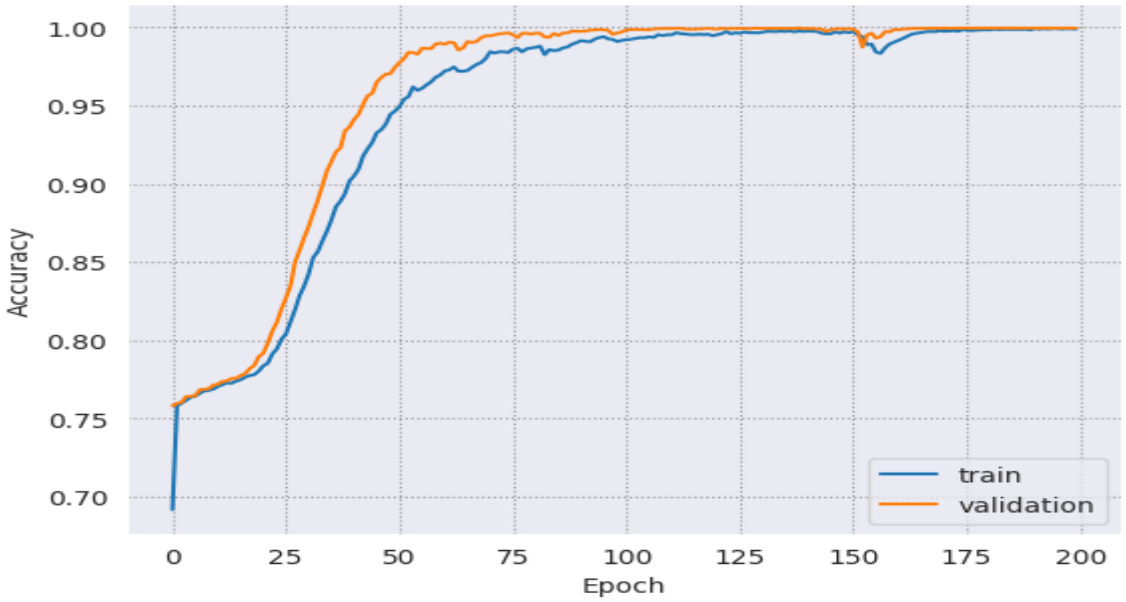

Fig 4.7 Accuracy curve of GRU

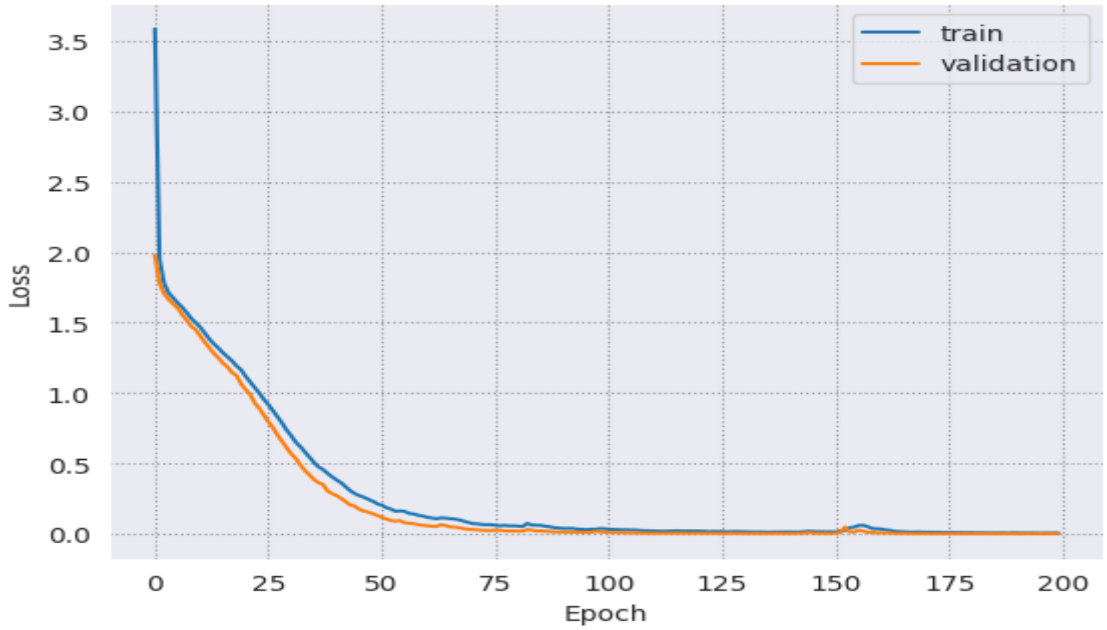

Fig 4.8 Loss curve of GRU

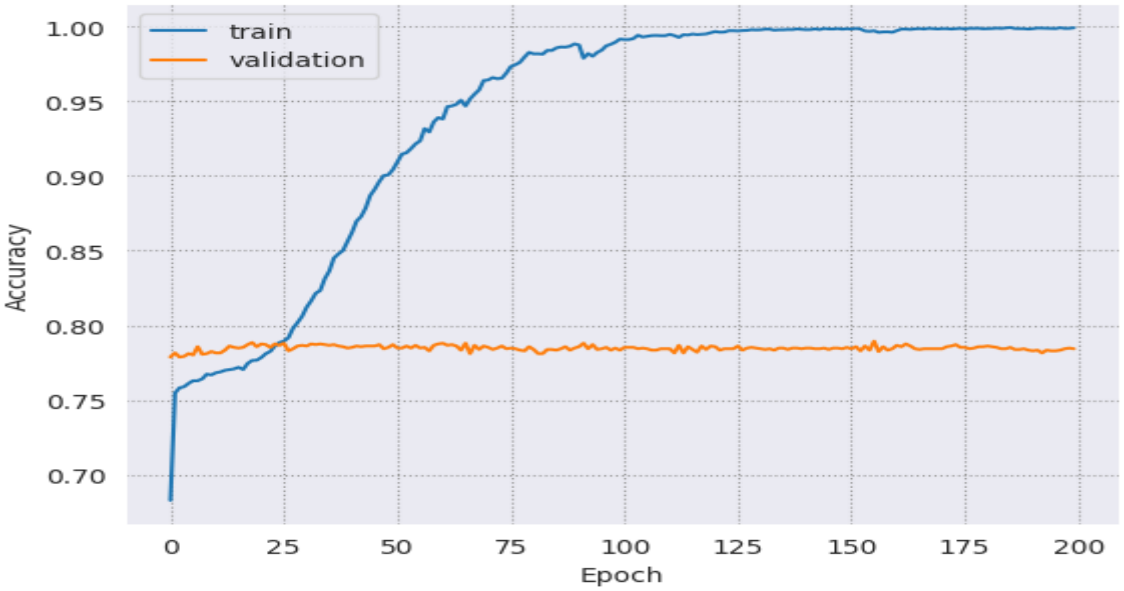

Fig 4.9 Accuracy curve of Combined LSTM and GRU

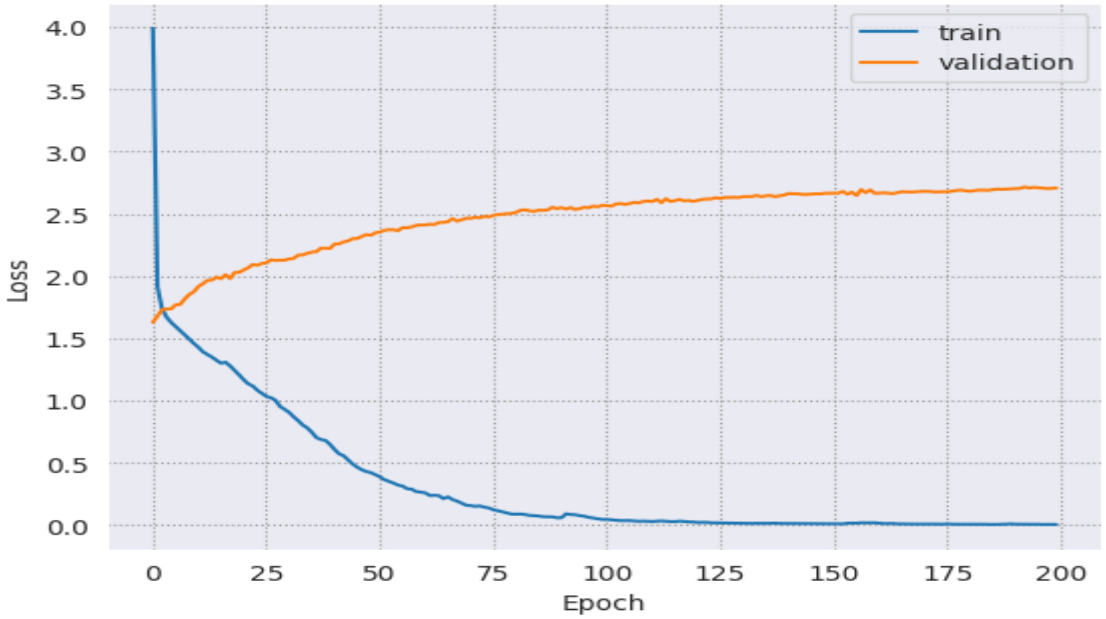

Fig 4.10 Loss curve of Combined LSTM and GRU

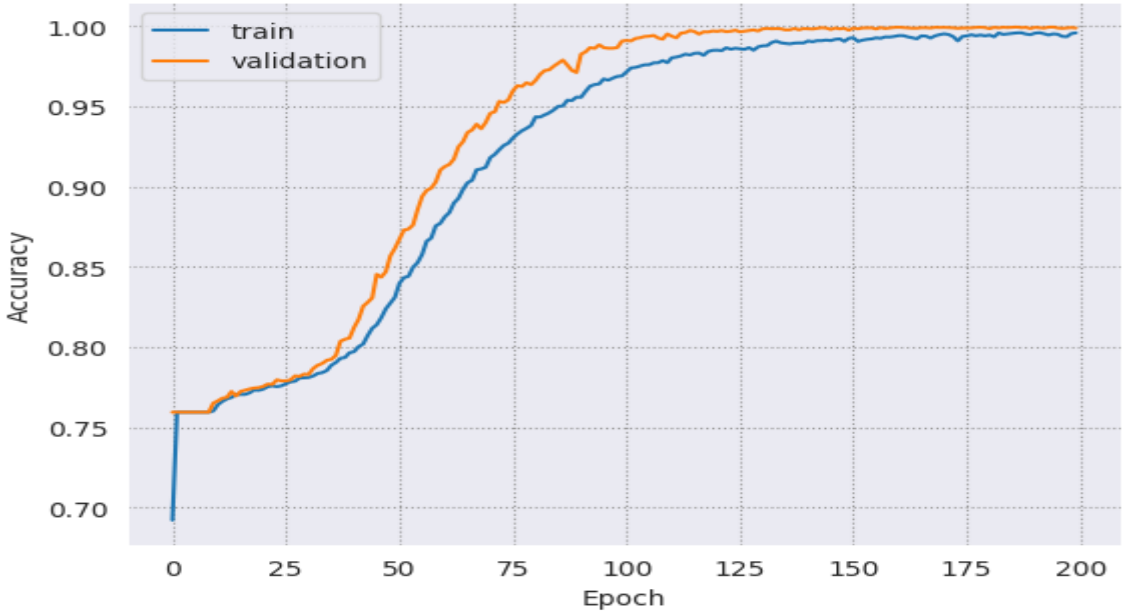

Fig 4.11 Accuracy curve of Transformer model

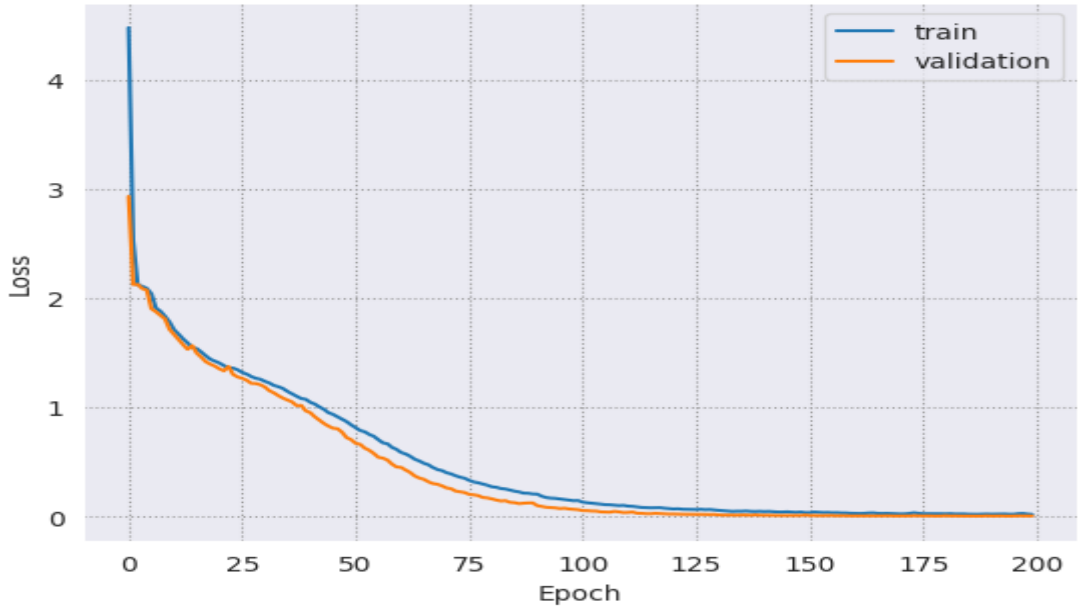

Fig 4.12 Loss Curve of Transformer model

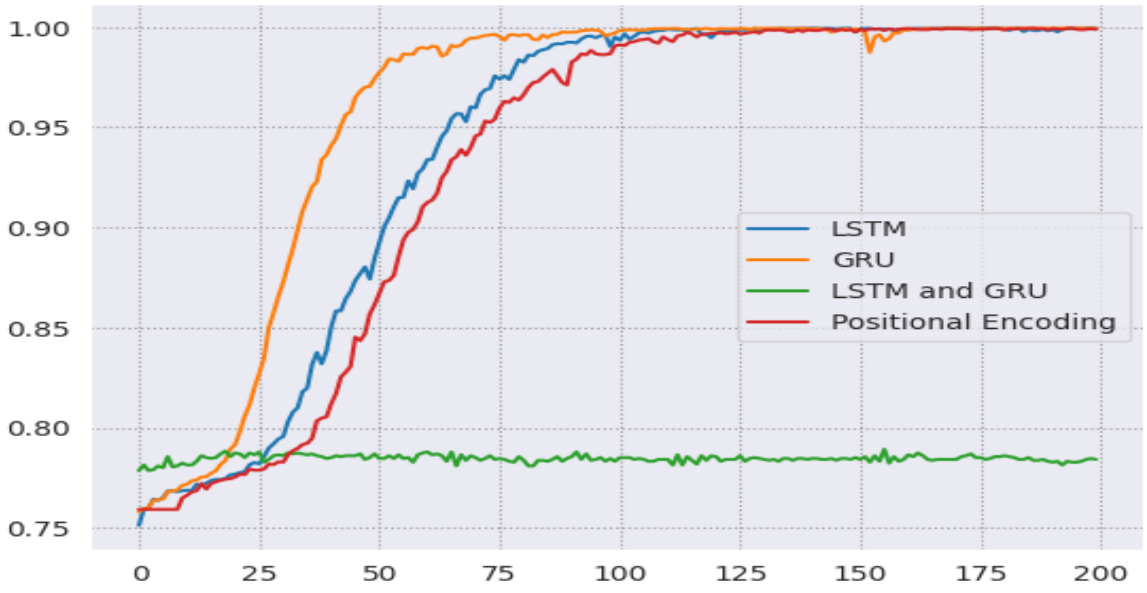

Fig 4.13 Validation Accuracy Curve comparison of different machine translation model

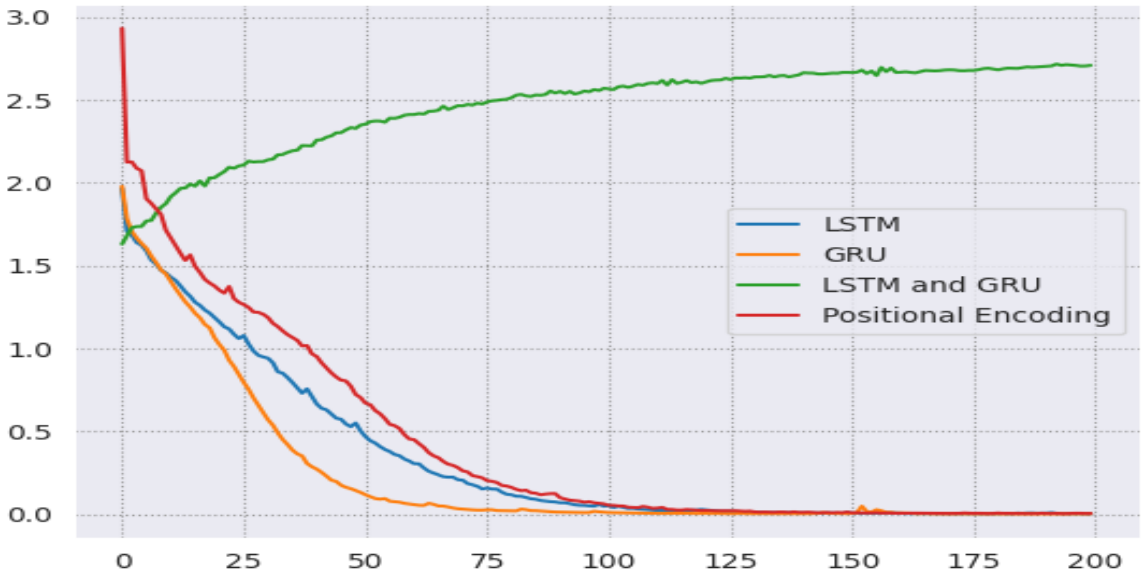

Fig 4.14 Validation Loss Curve comparison of different machine translation model

CHAPTER 5

PERFORMANCE ANALYSIS

- Loss: The loss function used is sparse categorical cross-entropy. It measures the difference between the true labels and the predicted probabilities for each class. Lower loss values indicate better model performance.

$$\text{Loss} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

where k is number of classes in the data

Where ;

- k => classes,
- y => actual value,
- $\hat{y}$ => Neural Network prediction.

- Accuracy: Accuracy measures the proportion of correctly classified samples out of the total number of samples. In the context of this program, it indicates the percentage of correctly translated sentences out of the total number of sentences. Higher accuracy values indicate better model performance.

The formula for accuracy can be expressed as:

$$Accuracy = (No.of.correct\ predictions \div Total\ No.of.predictions) \times 100\%$$

- ROUGE-1 Precision: This metric measures the proportion of n-grams (in this case, unigrams) in the generated summary that are also present in the reference summary. It indicates how many of the words/phrases in the generated summary are relevant to the reference summary.
- ROUGE-1 Recall: This metric measures the proportion of n-grams (unigrams) in the reference summary that are also present in the generated summary. It indicates how much of the information in the reference summary is captured by the generated summary.
- ROUGE-1 F1 Score: This metric combines precision and recall into a single score, providing a harmonic mean of the two. It gives a balanced measure of both precision and recall, with higher values indicating better overall performance.

- ROUGE-L Precision: ROUGE-L focuses on the longest common subsequence (LCS) between the generated and reference summaries. ROUGE-L Precision measures the proportion of LCS elements in the generated summary compared to the reference summary. It assesses how well the generated summary preserves the structure and sequence of information in the reference summary.
- ROUGE-L Recall: ROUGE-L Recall measures the proportion of LCS elements in the reference summary compared to the generated summary. It evaluates how much of the structural information in the reference summary is captured by the generated summary.
- ROUGE-L F1 Score: Similar to ROUGE-1 F1 Score, ROUGE-L F1 Score combines precision and recall of the LCS elements into a single score, providing an overall assessment of the generated summary's performance compared to the reference summary.

Model	ROUGE-1 Precision	ROUGE-1 Recall	ROUGE-1 F1 Score	ROUGE-L Precision	ROUGE-L Recall	ROUGE-L F1 Score
LSTM	0.25	1.0	0.4	0.25	1.0	0.4
GRU	0.25	1.0	0.4	0.25	1.0	0.4
LSTM and GRU	0.25	1.0	0.4	0.25	1.0	0.4
Positional Encoding	0.75	0.75	0.75	0.75	0.75	0.75

Table 5.1. Comparison of different Machine Translation Models

CHAPTER 6

RESULTS AND DISCUSSION

In summary, the analysis presents the training dynamics and performance of four distinct neural network architectures: LSTM, GRU, combined LSTM and GRU, and a transformer model. The LSTM model demonstrates consistent improvement in both loss and accuracy metrics, with high validation accuracy indicating robust generalization. Similarly, the GRU model exhibits effective learning without significant signs of overfitting, benefiting from its capability to capture long-range dependencies. However, the combined LSTM and GRU model shows evidence of overfitting, with training accuracy surpassing validation accuracy, necessitating implementation of regularization techniques and model complexity reduction for improvement. Meanwhile, the transformer model's training progresses smoothly, with both training and validation accuracies increasing over epochs, suggesting convergence. ROUGE metrics help quantify the similarity between the generated and reference summaries, with higher scores indicating better quality and alignment between the two.

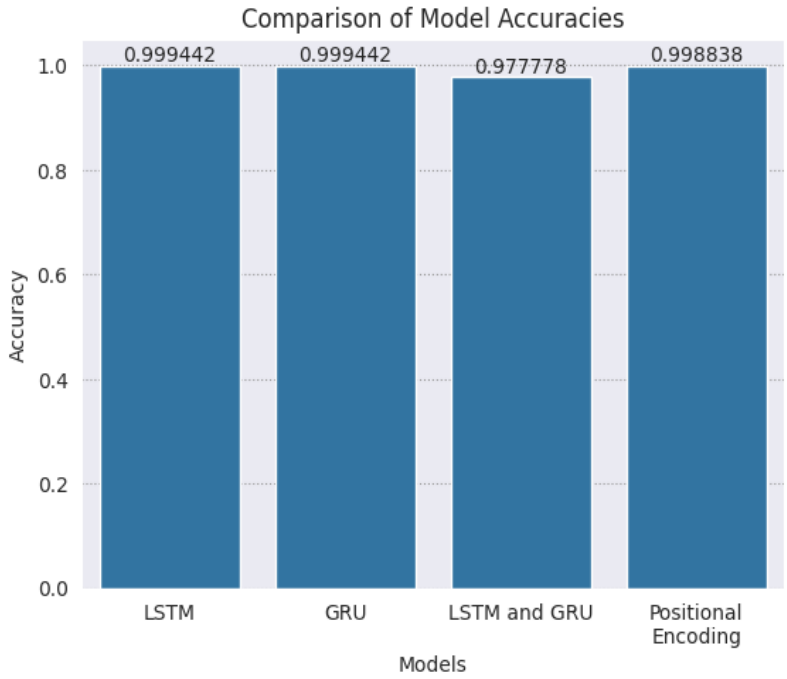

Fig.6.1 Comparison of different machine translation models

Though validation accuracy plateauing around 99% is promising, vigilance against potential overfitting remains essential. Overall, the LSTM, GRU, and transformer architectures showcase promising performance, while the combined LSTM and GRU model requires adjustments to enhance its generalization capability.

CHAPTER 7

CONCLUSION

In conclusion, the analysis underscores the efficacy of LSTM, GRU, and transformer architectures in the context of neural machine translation. The LSTM and GRU models demonstrate commendable performance, with the LSTM model showcasing consistent improvement and the GRU model effectively capturing long-range dependencies. However, the combined LSTM and GRU model exhibits signs of overfitting, highlighting the need for regularization techniques and model simplification. Despite these challenges, the transformer model shows promising convergence and high validation accuracy, indicating its potential for robust translation tasks. Moving forward, continued vigilance against overfitting and refinement of model architectures will be essential to further enhance the generalization capability of machine translation models. Meanwhile, the transformer model's promising convergence and high validation accuracy signify its prowess in seamlessly translating Thanglish expressions into fluent English sentences. As research in this area progresses, ongoing efforts to mitigate overfitting and optimize model architectures will be crucial for enhancing the accuracy and reliability of Thanglish to English translations, thus facilitating smoother communication across linguistic boundaries. Overall, it represents a significant step forward in addressing the communication challenges posed by Thanglish and underscores the transformative potential of natural language processing in fostering linguistic inclusivity and understanding across diverse communities. As we continue to refine and expand upon our model, we remain committed to advancing the field of multilingual communication and empowering individuals to connect and communicate effectively across language barriers.

CHAPTER 8

APPENDICES

1. APPENDIX – 1 CODING

```
no_of_epochs = 200
size_batch = 64
example_sent = "Inga enna nadakkuthu?"

# Importing dependencies

## Module importing

import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import re
import requests
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Dropout, LayerNormalization,
MultiHeadAttention, Embedding, GlobalAveragePooling1D
from tensorflow.keras.models import Model


# Pre-processing

## Reading dataset

def read(url):
  response = requests.get(url)
  text_content = response.text
  return text_content

tamil_sentences = read(r"https://github.com/gdhanush27/Tanglish-to-
English/raw/main/train.txt").splitlines()
english_sentences = read(r"https://github.com/gdhanush27/Tanglish-to-
English/raw/main/trainen.txt").splitlines()

len(tamil_sentences)

df = pd.DataFrame({'Tanglish': tamil_sentences, 'English': english_sentences})

## Case conversion
def lowercase_sentences(sentences):
    return [sentence.lower() for sentence in sentences]

tamil_sentences= lowercase_sentences(tamil_sentences)
english_sentences=lowercase_sentences(english_sentences)
```

Tokenizer

A tokenizer plays a crucial role in preparing text data for NLP tasks by breaking it down into meaningful units (tokens) and encoding them in a format that can be understood by machine learning algorithms.

```
class TanglishTokenizer:
    def __init__(self):
        self.word_index = {}
        self.index_word = {}
        self.num_words = 0

    def fit(self, texts):
        for text in texts:
            for word in text.split():
                if word not in self.word_index:
                    self.word_index[word] = self.num_words + 1
                    self.index_word[self.num_words + 1] = word
                    self.num_words += 1

    def toSequences(self, texts):
        sequences = []
        for text in texts:
            sequence = [self.word_index[word] for word in text.split() if word in self.word_index]
            sequences.append(sequence)
        return sequences
```

1. **Initialization (`__init__` method)**:
   - In the `__init__` method, we initialize three instance variables: `word_index`, `index_word`, and `num_words`.
   - `word_index`: This dictionary will store the mapping from words to integer indices.
   - `index_word`: This dictionary will store the mapping from integer indices to words.
   - `num_words`: This counter keeps track of the number of unique words encountered.

2. **Fitting on Texts (`fit_on_texts` method)**:
   - The `fit_on_texts` method takes a list of texts as input and learns the vocabulary from them.
   - It iterates through each text in the input list and splits it into individual words.
   - For each word in the text, it checks if the word is already present in the vocabulary (`word_index`).
   - If the word is not present, it assigns it a unique integer index, updates the `word_index` and `index_word` dictionaries, and increments the `num_words` counter.

3. **Texts to Sequences (`texts_to_sequences` method)**:
   - The `texts_to_sequences` method takes a list of texts as input and converts them into sequences of integer indices.
   - It iterates through each text in the input list and splits it into individual words.
   - For each word in the text, it checks if the word is present in the vocabulary (`word_index`).
   - If the word is in the vocabulary, it retrieves its corresponding integer index from the `word_index` dictionary and adds it to the sequence.
   - The method returns a list of sequences, where each sequence represents the integer indices of words in the corresponding text.

```
tamil_tokenizer = TanglishTokenizer()
tamil_tokenizer.fit(tamil_sentences)

# Convert Tamil sentences to sequences of integers
tamil_sequences = tamil_tokenizer.toSequences(tamil_sentences)

# Create and fit custom English tokenizer
english_tokenizer = TanglishTokenizer()
english_tokenizer.fit(english_sentences)

# Convert English sentences to sequences of integers
english_sequences = english_tokenizer.toSequences(english_sentences)

print("Tamil Sequences:", tamil_sequences)
print("English Sequences:", english_sequences)
```

Padding

```
# Pad sequences to ensure uniform length
def pad_sequences(sequences, maxlen, padding='post'):
    """
    Pad sequences to ensure uniform length.

    Parameters:
    sequences (list): List of sequences of integers.
    maxlen (int): Maximum length to pad the sequences.
    padding (str): 'pre' or 'post', padding position.

    Returns:
    padded_sequences (numpy.ndarray): Padded sequences.
    """
    padded_sequences = []
    for seq in sequences:
        if padding == 'pre':
            padded_seq = [0] * (maxlen - len(seq)) + seq[:maxlen]
        elif padding == 'post':
            padded_seq = seq[:maxlen] + [0] * (maxlen - len(seq))
        padded_sequences.append(padded_seq)
    return np.array(padded_sequences)
```

 # Calculate max length

```
max_length = max(len(seq) for seq in tamil_sequences + english_sequences)

# Pad sequences
tamil_padded = pad_sequences(tamil_sequences, maxlen=max_length, padding='post')
english_padded = pad_sequences(english_sequences, maxlen=max_length, padding='post')

print("Tamil Padded Sequences:\n", tamil_padded)
print("\nEnglish Padded Sequences:\n", english_padded)
```

Vacabulary size

```
input_vocab_size = len(tamil_tokenizer.word_index) + 1
```

```
target_vocab_size = len(english_tokenizer.word_index) + 1
```

 # Model (LSTM)

Model architecture

```
# Define the model architecture
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=len(tamil_tokenizer.word_index) + 1, output_dim=256,
input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(512, return_sequences=True)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(512, return_sequences=True)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(len(english_tokenizer.word_index) + 1,
activation='softmax'))
])
```

 # Compile the model

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Model training

```
history1 = model.fit(tamil_padded, english_padded, epochs=no_of_epochs, batch_size = size_batch
,validation_data=(tamil_padded, english_padded))
```

Model evaluation

```
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['accuracy'])
plt.plot(history1.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['train','validation'])
plt.show()

sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['train','validation'])
plt.show()
```

Loss and accuracy

```
loss1, accuracy1 = model.evaluate(tamil_padded, english_padded)

# Print the evaluation results
print(f"Loss: {loss1}")
print(f"Accuracy: {accuracy1}")
```

Evaluation with example

```
def translate(sentence):
    sequence = tamil_tokenizer.toSequences([sentence])
    padded_sequence = pad_sequences(sequence, maxlen=max_length, padding='post')
    translation = model.predict(padded_sequence)
    translation = tf.argmax(translation, axis=-1).numpy()[0]
    translated_sentence = ' '.join([list(english_tokenizer.word_index.keys())[idx-1] for idx in
translation if idx != 0])
    return translated_sentence

tamil_sentence = example_sent
res1 = translate(tamil_sentence)
print(f"Tamil: {tamil_sentence}")
print(f"English: {res1}")
```

 # Model (GRU)

Model Architecture

```
model3 = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=len(tamil_tokenizer.word_index) + 1, output_dim=256,
input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(512, return_sequences=True)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(512, return_sequences=True)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(len(english_tokenizer.word_index) + 1,
activation='softmax'))
])

# Compile the model
model3.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Model Training

```
history2 = model3.fit(tamil_padded, english_padded, epochs=no_of_epochs,
batch_size=size_batch,validation_data=(tamil_padded, english_padded))
```

Model Evaluation

```
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history2.history['accuracy'])
plt.plot(history2.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['train','validation'])
plt.show()

sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.xlabel('Epoch')
```

```
plt.ylabel('Loss')
plt.legend(['train','validation'])
plt.show()
```

Loss and accuracy

```
loss2, accuracy2 = model3.evaluate(tamil_padded, english_padded)

# Print the evaluation results
print(f"Loss: {loss2}")
print(f"Accuracy: {accuracy2}")
```

Evaluation with example

```
def translate(sentence):
    sequence = tamil_tokenizer.toSequences([sentence])
    padded_sequence = pad_sequences(sequence, maxlen=max_length, padding='post')
    translation = model3.predict(padded_sequence)
    translation = tf.argmax(translation, axis=-1).numpy()[0]
    translated_sentence = ' '.join([list(english_tokenizer.word_index.keys())[idx-1] for idx in
translation if idx != 0])
    return translated_sentence

tamil_sentence = example_sent
res2 = translate(tamil_sentence)
print(f"Tamil: {tamil_sentence}")
print(f"English: {res2}")
```

 # Combined model (LSTM and GRU)

Model Architecture

```
input_layer = tf.keras.layers.Input(shape=(max_length,))
embedding_layer = tf.keras.layers.Embedding(input_dim=len(tamil_tokenizer.word_index) + 1,
output_dim=256, input_length=max_length)(input_layer)
lstm_output = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(512,
return_sequences=True))(embedding_layer)
gru_output = tf.keras.layers.Bidirectional(tf.keras.layers.GRU(512,
return_sequences=True))(embedding_layer)
concatenated_output = tf.keras.layers.Concatenate()([lstm_output, gru_output])
dropout_layer = tf.keras.layers.Dropout(0.5)(concatenated_output)
output_layer =
tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(len(english_tokenizer.word_index) + 1,
activation='softmax'))(dropout_layer)
```

 # Create model

```
model_concat = tf.keras.models.Model(inputs=input_layer, outputs=output_layer)


# Enable eager execution
tf.config.run_functions_eagerly(True)

# Compile the model with run_eagerly=True
```

```
model_concat.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'], run_eagerly=True)
```

Model Training

```
# Train the model with validation data
history3 = model_concat.fit(tamil_padded, english_padded, epochs=no_of_epochs,
batch_size=size_batch, validation_split=0.1)

# Disable eager execution after training
tf.config.run_functions_eagerly(False)
```

Model Evaluation

```
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history3.history['accuracy'])
plt.plot(history3.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['train','validation'])
plt.show()

sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['train','validation'])
plt.show()
```

Loss and accuracy

```
loss3, accuracy3 = model_concat.evaluate(tamil_padded, english_padded)

# Print the evaluation results
print(f"Loss: {loss3}")
print(f"Accuracy: {accuracy3}")
```

Evaluation with example

```
# Preprocess the input Tamil sentence
input_sequence = tamil_tokenizer.toSequences([example_sent])

# Pad the sequence to match the model's input length
input_sequence_padded = tf.keras.preprocessing.sequence.pad_sequences(input_sequence,
maxlen=max_length, padding='post')

# Get the predicted English translation from the model
predicted_indices = model_concat.predict(input_sequence_padded)

# Convert predicted indices to English words using the English tokenizer
predicted_words = []
for idx in np.argmax(predicted_indices, axis=-1)[0]:
```

```
    # Check if the index exists in the tokenizer's index_word dictionary
    if idx in english_tokenizer.index_word:
        predicted_words.append(english_tokenizer.index_word[idx])

# Join the words to form the translated sentence
res3 = ' '.join(predicted_words)

# Print the translated sentence
print("Translated Sentence:", res3)
```

 # Transformer Model

Model architecture

```
class TransformerEncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, ff_dim, dropout_rate):
        super(TransformerEncoderLayer, self).__init__()
        self.multi_head_attention = MultiHeadAttention(num_heads=num_heads,
                                                       key_dim=d_model // num_heads)
        self.dropout1 = Dropout(dropout_rate)
        self.layer_norm1 = LayerNormalization(epsilon=1e-6)
        self.dense1 = Dense(ff_dim, activation='relu')
        self.dense2 = Dense(d_model)
        self.dropout2 = Dropout(dropout_rate)
        self.layer_norm2 = LayerNormalization(epsilon=1e-6)

    def call(self, inputs):
        attention_output = self.multi_head_attention(inputs, inputs)
        attention_output = self.dropout1(attention_output)
        attention_output = self.layer_norm1(inputs + attention_output)
        ff_output = self.dense2(self.dense1(attention_output))
        ff_output = self.dropout2(ff_output)
        ff_output = self.layer_norm2(attention_output + ff_output)

        return ff_output
```

`TransformerEncoderLayer` Class

This class defines a single encoder layer in a transformer model. It inherits from `tf.keras.layers.Layer`.

Attributes:

- `multi_head_attention`: A `MultiHeadAttention` layer, which computes multi-head self-attention.
- `dropout1`: A `Dropout` layer for the first dropout operation.
- `layer_norm1`: A `LayerNormalization` layer for the first layer normalization operation.
- `dense1`: A `Dense` layer for the first feedforward neural network.
- `dense2`: A `Dense` layer for the second feedforward neural network.
- `dropout2`: A `Dropout` layer for the second dropout operation.
- `layer_norm2`: A `LayerNormalization` layer for the second layer normalization operation.

Methods:

- `call(inputs)`: Method that defines the forward pass of the layer.
  - `inputs`: Input tensor to the layer.
  - Computes multi-head self-attention on the input tensor.
  - Applies dropout and layer normalization to the attention output.
  - Passes the normalized attention output through a feedforward neural network.
  - Applies dropout and layer normalization to the output of the feedforward neural network.
  - Returns the output tensor.

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, max_len, d_model):
        super(PositionalEncoding, self).__init__()
        self.pos_encoding = self.positional_encoding(max_len, d_model)

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

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(np.arange(position)[:,np.newaxis], \
                                     np.arange(d_model)[np.newaxis,:],  \
                                     d_model)
        # apply sin to even indices in the array; 2i
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        # apply cos to odd indices in the array; 2i+1
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads[np.newaxis, ...]
        return tf.cast(pos_encoding, dtype=tf.float32)

    def call(self, inputs):
        return inputs + self.pos_encoding[:, :tf.shape(inputs)[1], :]
```

`**PositionalEncoding` Class**

This class defines the positional encoding layer used in transformer models. It inherits from `tf.keras.layers.Layer`.

Attributes:

- `pos_encoding`: The positional encoding matrix computed using the `positional_encoding` method.

Methods:

- `__init__(self, max_len, d_model)`: Constructor method that initializes the positional encoding layer.
  - `max_len`: Maximum sequence length.
  - `d_model`: Dimensionality of the model.
  - Computes the positional encoding matrix using the `positional_encoding` method.

- `get_angles(self, pos, i, d_model)`: Method to compute the angle rates used in the positional encoding formula.
  - `pos`: Position indices.
  - `i`: Dimension indices.
  - `d_model`: Dimensionality of the model.
  - Computes the angle rates based on the provided position and dimension indices.

- `positional_encoding(self, position, d_model)`: Method to compute the positional encoding matrix.
  - `position`: Maximum sequence length.
  - `d_model`: Dimensionality of the model.
  - Computes the positional encoding matrix using sine and cosine functions.

- `call(self, inputs)`: Method that defines the forward pass of the layer.
  - `inputs`: Input tensor to the layer.
  - Adds the positional encoding to the input tensor.
  - Returns the sum of the input tensor and the positional encoding.

```
def transformer_model(input_vocab_size, target_vocab_size, max_length, d_model=128, num_heads=4, ff_dim=512, dropout=0.1):
    inputs = Input(shape=(max_length,))
    embedding_layer = Embedding(input_vocab_size, d_model, input_length=max_length)(inputs)
    positional_encoding = PositionalEncoding(max_length, d_model)(embedding_layer)
    x = Dropout(dropout)(positional_encoding)

    for _ in range(2):  # Stack of 2 encoder layers
        x = TransformerEncoderLayer(d_model, num_heads, ff_dim, dropout)(x)

    outputs = Dense(target_vocab_size, activation='softmax')(x)
    model = Model(inputs=inputs, outputs=outputs)
    return model

model1 = transformer_model(input_vocab_size, target_vocab_size, max_length)
model1.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Model Training

```
# Assuming you have your padded sequences tamil_padded and english_padded
history4 = model1.fit(tamil_padded, english_padded, batch_size=size_batch, epochs=no_of_epochs, validation_data=(tamil_padded, english_padded))
```

Model evaluation

```
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history4.history['accuracy'])
plt.plot(history4.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['train','validation'])
plt.show()
```

```
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history4.history['loss'])
plt.plot(history4.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['train','validation'])
plt.show()
```

Loss and accuracy

```
loss4, accuracy4 = model1.evaluate(tamil_padded, english_padded)

# Print the evaluation results
print(f"Loss: {loss4}")
print(f"Accuracy: {accuracy4}")
```

Evaluation with example

```
def translate(tamil_sentence, tamil_tokenizer, english_tokenizer, model, max_length):
    # Preprocess the Tamil sentence
    tamil_sentence = lowercase_sentences([tamil_sentence])[0]
    tamil_sequence = tamil_tokenizer.toSequences([tamil_sentence])[0]

    # Pad the Tamil sequence
    tamil_padded = pad_sequences([tamil_sequence], maxlen=max_length, padding='post')
    # Translate the Tamil sequence to English
    english_padded = model1.predict(tamil_padded)
    # Decode the English sequence
    english_sequence =
        [english_tokenizer.index_word.get(np.argmax(token), '') for token in english_padded[0]]
    english_translation = ' '.join(english_sequence).strip()
    return english_translation

tamil_sentence = example_sent
res4 = translate(tamil_sentence, tamil_tokenizer, english_tokenizer, model, max_length)
print("Translated English sentence:", res4)

# Comparision of results
accuracy = [accuracy1,accuracy2,accuracy3,accuracy4]
loss = [loss1,loss2,loss3,loss4]
l = ["LSTM","GRU","LSTM and GRU","Positional Encoding"]
res = [res1,res2,res3,res4]



## Accuracy comparision
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['accuracy'])
plt.plot(history2.history['accuracy'])
plt.plot(history3.history['accuracy'])
plt.plot(history4.history['accuracy'])
plt.legend(l)
plt.show()
```

```
## Validation Accuracy comparision
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['val_accuracy'])
plt.plot(history2.history['val_accuracy'])
plt.plot(history3.history['val_accuracy'])
plt.plot(history4.history['val_accuracy'])
plt.legend(l)
plt.show()

## Loss comparision
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['loss'])
plt.plot(history2.history['loss'])
plt.plot(history3.history['loss'])
plt.plot(history4.history['loss'])
plt.legend(l)
plt.show()

## Validation Loss comparision
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
plt.plot(history1.history['val_loss'])
plt.plot(history2.history['val_loss'])
plt.plot(history3.history['val_loss'])
plt.plot(history4.history['val_loss'])
plt.legend(l)
plt.show()

## Comparision of Example Sentance
print("Test Sentence = "+example_sent)
for i in range(len(l)):
  print("Translated by",l[i],"=",res[i])

## Overall Comaparision of Accuracy and Loss
l = ["LSTM","GRU","LSTM and GRU","Positional\nEncoding"]
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
ax = sns.barplot(x=l, y=accuracy, errorbar=None,legend="brief")
ax.bar_label(ax.containers[0], fontsize=10);
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.title('Comparison of Model Accuracies')
plt.show()



l = ["LSTM","GRU","LSTM and GRU","Positional\nEncoding"]
sns.set_style("darkgrid", {"grid.color": ".6", "grid.linestyle": ":"})
ax = sns.barplot(x=l, y=loss, errorbar=None,legend="brief")
ax.bar_label(ax.containers[0], fontsize=10);
plt.xlabel('Models')
plt.ylabel('Loss')
plt.title('Comparison of Model Losses')
plt.show()
```

APPENDIX – 2 SCREENSHOTS

```
50/50 [==============================] - 1s 22ms/step - loss: 0.0029 - accuracy: 0.9992
Loss: 0.002886169822886586
Accuracy: 0.9991500973701477
```

Fig A2.1: Output of LSTM model

```
50/50 [==============================] - 1s 20ms/step - loss: 0.0035 - accuracy: 0.9992
Loss: 0.003512718016281724
Accuracy: 0.9991500973701477
```

Fig A2.2: Output of GRU model

```
50/50 [==============================] - 2s 43ms/step - loss: 0.2847 - accuracy: 0.9762
Loss: 0.28469714522361755
Accuracy: 0.9762020111083984
```

Fig A2.3: Output of combined both LSTM AND GRU model

```
50/50 [==============================] - 1s 11ms/step - loss: 0.0013 - accuracy: 0.9996
Loss: 0.0013171829050406814
Accuracy: 0.9996336698532104
```

Fig A2.4: Output of Transformer model

```
Model = LSTM
Translated = what
ROUGE-1 Precision: 0.25
ROUGE-1 Recall: 1.0
ROUGE-1 F1 Score: 0.4
ROUGE-L Precision: 0.25
ROUGE-L Recall: 1.0
ROUGE-L F1 Score: 0.4

Model = GRU
Translated = happening
ROUGE-1 Precision: 0.25
ROUGE-1 Recall: 1.0
ROUGE-1 F1 Score: 0.4
ROUGE-L Precision: 0.25
ROUGE-L Recall: 1.0
ROUGE-L F1 Score: 0.4
```

Fig A2.5: Output of Rouge Score for LSTM AND GRU

```
Model = LSTM and GRU
Translated = happening
ROUGE-1 Precision: 0.25
ROUGE-1 Recall: 1.0
ROUGE-1 F1 Score: 0.4
ROUGE-L Precision: 0.25
ROUGE-L Recall: 1.0
ROUGE-L F1 Score: 0.4

Model = Positional Encoding
Translated = what's happening here?
ROUGE-1 Precision: 0.75
ROUGE-1 Recall: 0.75
ROUGE-1 F1 Score: 0.75
ROUGE-L Precision: 0.75
ROUGE-L Recall: 0.75
ROUGE-L F1 Score: 0.75
```

Fig A2.6: Output of Rouge Score for Combined of (LSTM AND GRU) and Transformer Model

REFERENCES

1. M. S. Mary N J, V. M. Shetty and S. Umesh, "Investigation of Methods to Improve the Recognition Performance of Tamil-English Code-Switched Data in Transformer Framework," ICASSP 2020 - 2020 IEEE
2. Shubham Toshniwal, Tara N Sainath, Ron J Weiss, Bo Li, Pedro Moreno, Eugene Weinstein, et al., "Multilingual speech recognition with a single end-to-end model", 2018 IEEE
3. William Chan, Navdeep Jaitly, Quoc Le and Oriol Vinyals, "Listen attend and spell: A neural network for large vocabulary conversational speech recognition", 2016 IEEE
4. Changhao Shan, Chao Weng, Guangsen Wang, Dan Su, Min Luo, Dong Yu, et al., "Investigating end-to-end speech recognition for mandarin-english code-switching", ICASSP 2019-2019 IEEE
5. Zhiping Zeng, Yerbolat Khassanov, Van Tung Pham, Haihua Xu, Eng Siong Chng and Haizhou Li, "On the end-to-end solution to mandarin-english code-switching speech recognition", 2018
6. Shinji Watanabe, Takaaki Hori and John R Hershey, "Language independent end-to-end architecture for joint language identification and speech recognition", 2017 IEEE