CHAPTER 1: INTRODUCTION

1.1. Introduction

Machine translation (MT) is a branch of artificial intelligence where computers automatically translate sentences, phrases, and words from one language to another using dictionaries, programs, and logical instructions. This field combines linguistics, computing, and statistics and has evolved to include four main types: rule-based MT, statistical MT, hybrid MT, and Neural Machine Translation (NMT) [1]. Among these, NMT has become the most widely used approach, relying on deep learning algorithms and artificial neural networks to capture the context and grammar of text during translation. Recent advancements, such as the encoder-decoder architecture with attention mechanisms, have significantly improved the ability to handle long-term dependencies. These developments are particularly effective for translating between structurally different languages like English and Nepali.

This system aims to help in communication by translating English into Nepali using advanced deep learning techniques, specifically Long Short-Term Memory (LSTM). In this system, advanced techniques like Local Attention Mechanism and Teacher Forcing are used such that the model learns faster and generalises the sentences correctly, which will eventually increase the quality of translation. The model is deployed on a localhost server using django and can be accessed through a simple web interface. This system addresses the challenges faced by individuals struggling with Nepali language comprehension and serves diverse groups, including students, teachers, locals and professionals. This is similar to existing translation tools like Google Translate, but the system leverages LSTM to deliver more accurate translations.

1.2. Problem Statement

Every year, many people visit Nepal but face challenges in communicating effectively due to the language barrier with Nepali. Similarly, native Nepali speakers often struggle to find appropriate Nepali words for specific English terms. While various machine translation models exist, most fail to deliver accurate results for Nepali due to its resource-scarce nature and

significant linguistic differences from English. Nepali's Subject-Object-Verb (SOV) grammatical structure, compared to English's Subject-Verb-Object (SVO) order, often leads to errors in sentence construction. Furthermore, Nepali's rich morphology and the inherent ambiguity of English phrases add layers of complexity to the translation process. Existing English-to-Nepali translation systems, such as Google Translate, frequently fall short in preserving contextual meaning, handling grammar, or accurately translating specialised terminology in fields like education, healthcare, and law. This lack of reliable, context-aware translation tools limits the accessibility of information and effective communication between English and Nepali speakers.

Therefore, there is an urgent need for a good translation system that addresses these challenges. By leveraging advanced deep learning techniques, such as encoder-decoder architecture with LSTM cells and a global attention mechanism, our project aims to provide accurate, context-sensitive English-to-Nepali translations, bridging the linguistic gap and facilitating cross-language communication.

1.3. Objectives

The main aim of this project is to develop a web-based language translation system which translates English text to Nepali text using Encoder Decoder architecture and Attention mechanism. To fulfil this aim, some of the objectives are

1. To develop an LSTM model with Encoder Decoder architecture.
2. To develop a web-based system using Django.

1.4. Scope and Limitation

Scope

1. Creating a web-based user-friendly interface to input English text and get Nepali text as a result.
2. Developing LSTM based encoder decoder model with global attention mechanism for translation.

3. Training model on the pre-processed dataset and fine-tuning it to optimise performance metrics
4. Providing a complete documentation including technical details.

Limitations

1. Limited to unidirectional translation from English to Nepali only
2. Length of sentences cannot be longer than 10
3. Limited availability of data for Nepali could impact the model's generalisation capabilities.
4. Using evaluation metrics may not fully capture translation quality for Nepali text.

1.5. Development Methodology

The development of the English to Nepali translation system follows an iterative development methodology to ensure continuous improvement and adaptation. This approach allows for repeated refinement and optimization of the model based on performance feedback at each stage

Iteration 1: Data Collection and Preprocessing:

- Data Collection: The first step is to collect a large set of English and Nepali sentence pairs. These sentence pairs should come from different sources to cover a variety of topics and contexts. This helps ensure that the model can handle different types of sentences, from simple everyday conversation to more complex formal language. By using diverse data, the model will learn to translate across a wide range of sentence structures and expressions.
- Initial Preprocessing: Clean the data by tokenizing the sentences, converting text to lowercase, removing special characters, and handling punctuation. Special tokens like <sos>, <eos>, and <unk> are added to equal sentence lengths.
- Initial Model Setup: Split the dataset into training and testing sets for evaluation during each iteration.

Iteration 2: Model Design and Initial Training:

- Encoder-Decoder Architecture: The initial model was built using the Encoder-Decoder architecture with LSTM units. The encoder processes English sentences, converting them into a form the decoder can understand, which then generates the corresponding Nepali translation. This structure enables the model to learn translation by grasping the meaning and structure of the source language and producing accurate translations in the target language.
- Integration of Attention Mechanism: To enhance translation quality, we integrated a global attention mechanism into the decoder. This mechanism allows the model to focus on important words or phrases in the English sentence, improving the accuracy and fluency of the Nepali translation.
- Initial Training: The model was then trained on the prepared dataset, where it compared its generated translations to the correct ones. Basic hyperparameters such as batch size, learning rate, and the number of epochs were tuned to optimize performance. Training loss and validation accuracy were closely monitored to ensure consistent improvement.

Iteration 3: Evaluation and Performance Analysis:

- Evaluation on Test Set: After training, the model's performance is measured using WER.
- Error Analysis: After training, the model is analysed to identify challenges such as differences in grammar, complex morphology, and domain-specific vocabulary. And misinterpretation of specialised terms were common. This analysis highlighted areas for improvement, guiding model adjustments for better performance.

Iteration 4: Model Refinement:

- Refining the Architecture: Based on the feedback from evaluation and error analysis, adjust the model architecture. This includes increasing the number of LSTM layers, modifying the attention mechanism, or adjusting the sequence length.
- Hyperparameter Tuning: Experimented with different hyperparameters, such as changing the number of epochs, learning rate, and batch size, to improve training stability and performance.

- Retraining: Retrain the model on the updated dataset with refined settings and evaluate the model's performance again.

Iteration 5: Continuous Testing and Optimization:

- Testing in Real-World Scenarios: Test the model with real-world data or edge cases, including complex sentences.
- Model Optimization: Implemented techniques such as pruning, quantization, or knowledge distillation to optimise the model for faster inference and deployment.

Iteration 6: Final Evaluation and Deployment:

- Final Evaluation: After several iterations of improvement, a final evaluation of the model using standard metrics.
- Deployment: After final evaluation model is deployed using Django

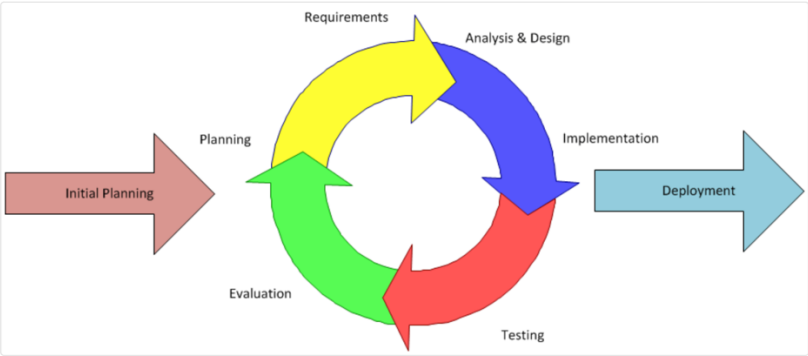


Figure 1.1: Iterative Development Strategy

1.6. Report Organization

This report is structured to provide a comprehensive overview of the English-to-Nepali translation system, from introduction to final deployment.

Chapter 1: Introduction

This chapter introduces the project, highlighting the significance of machine translation, particularly for English to Nepali. It explains the problem statement, objectives, and limitations, along with the development methodology used for building the translation system.

Chapter 2: Background Study and Literature Review

This chapter delves into the background of machine translation, reviewing the relevant literature in the field, including existing approaches and challenges in translating English to Nepali. This sets the foundation for the proposed model.

Chapter 3: System Analysis

This chapter covers the analysis of the system, including functional and non-functional requirements. It also includes feasibility analysis, assessing the technical, operational, economic, and schedule aspects of the system development.

Chapter 4: System Design

Here, the design of the translation system is presented, detailing the architecture and algorithms used, including the encoder-decoder LSTM model and attention mechanism that power the translation.

Chapter 5: Implementation and Testing

This chapter describes the implementation of the system, including the tools used and the details of the modules developed. It also includes a section on testing, covering both unit and system testing, as well as result analysis based on the evaluation metrics.

Chapter 6: Conclusion and Future Recommendations

The report concludes with a summary of the project's outcomes, followed by recommendations for future improvements or potential areas of research.

CHAPTER 2: BACKGROUND STUDY AND LITERARURE REVIEW

2.1. Background Study

The LSTM Encoder-Decoder architecture consists of two distinct components: the encoder and the decoder, both implemented using Long Short-Term Memory (LSTM) networks. To grasp the functionality of this framework, it is crucial to first understand the underlying principles of neural networks and how Long Short-Term Memory networks differ from standard neural networks. This section provides an overview of the fundamentals of neural networks and highlights the characteristics that distinguish LSTM networks from other types of neural networks.

One-hot encoding is a common technique used to represent categorical data, particularly in natural language processing (NLP) tasks. In this approach, each word or token in the vocabulary is represented as a unique vector of zeros and ones, where the length of the vector is equal to the size of the vocabulary. The vector has a single 1 at the position corresponding to the word's index in the vocabulary, and 0s in all other positions.In our project, one-hot encoding is used to feed input sequences to the encoder and output sequences to the decoder, especially in the initial steps of the model. Each word in the English sentence is first transformed into a one-hot vector, which is then passed into the encoder.

Mathematically, if we have a sentence with NNN words, the input sequence would be represented as:

$$X = [x1,x2,x3,.....,xN]$$

where each x_i is the one-hot encoded vector corresponding to the i-th word in the sentence. For instance, if the word "apple" corresponds to index 1 in the vocabulary of size V, its one-hot encoding x_{apple} would be:

$$X_{apple} = [1,0,0,0,....0]^T$$

The embedding layer learns a dense vector e_{apple} ^{d} where d is the desired dimensionality This embedding vector for each word contains more semantic information and helps the model understand the relationships between different words.

$$e_{apple} = W \cdot x_{apple}$$

where W is the embedding matrix of size V X d (vocabulary size by embedding dimension) and x_{apple} is the one-hot vector.

The encoder is responsible for processing the input sequence, which in the case of NMT is a sentence in the source language. It takes in the sequence of words, typically represented as one-hot encoded vectors (or, more efficiently, through embeddings), and produces a context vector that encapsulates the information of the entire sentence.

Mathematically, for an input sentence X = [x1,x2,x3,.....,xN] , where each x_i is the representation of a word in the vocabulary (after one-hot encoding or embedding), the encoder processes these words step by step using Long Short-Term Memory (LSTM).The encoder can be described as follows:

$$h_t = \text{Encoder}(h_{t-1}, x_t) \text{ [2]}$$

Here:

- h_t is the hidden state at time step t, which is a representation of the input up to that time step.
- x_t is the input at time step t
- h_{t-1} is the hidden state from the previous time step.

At each time step, the encoder updates its hidden state based on the current input word and the previous hidden state. The final hidden state h_N after processing all words in the input sequence X is often referred to as the context vector C which contains the compressed information of the entire input sequence.

$$C = h_N$$

This context vector C is passed to the decoder as the initial state, which allows it to generate the output sequence in the target language.The decoder takes the context vector C from the encoder and generates the output sequence, which is the translation in the target language. It does this by producing one word at a time, conditioned on the context vector and the previously generated word. The decoder can be described as follows:

$$st = \text{Decoder}(s_{t-1}, y_{t-1}, C) \text{ [2]}$$

Where:

- s_t is the hidden state at time step t of the decoder.

- y_{t-1} is the word generated at the previous time step (or the actual word during training with teacher forcing).
- C is the context vector passed from the encoder.

The decoder generates the probability distribution over the vocabulary for the next word in the sequence:

$$P(y_t \mid y_1,.....,y_{t-1}, C) = \text{softmax}(Ws_t + b) \text{ [3]}$$

The Long Short-Term Memory (LSTM) architecture is a widely adopted variant of Recurrent Neural Networks (RNN), primarily due to its ability to effectively address the problem of vanishing gradients. Unlike standard RNNs, LSTMs include special components called "gates" that help manage the flow of information through the network and improve back-propagation. Specifically, at each time step t = s the LSTM updates the hidden state h_t = s by combining data from the current input vector x_t = s, the previous hidden state h_{t-1}, and an additional memory cell c_t=s. The update process involves three main gates: the input gate (i), forget gate (f), and output gate (o). These gates determine how much information should be retained from the current input versus what should be remembered from the memory cell. The calculations for these gates are based on the concatenation of the current input x_t = s and the previous hidden state h_{t-1} forming a combined vector of size (H+X) x 1where H and X are the dimensions of the hidden state and input vectors, respectively. Each gate has its own weight matrix, which transforms this concatenated vector into a representation that matches the size of the memory cell, and each result is then passed through the sigmoid function to generate a gate-specific output

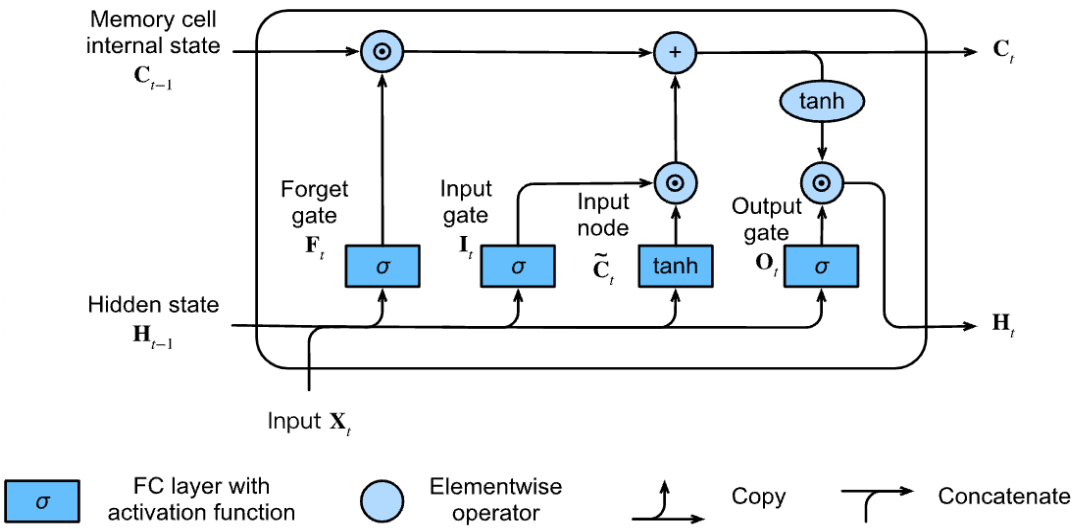


Figure 2.1: Single LSTM cell Architecture of English to Nepali Translation Model

2.2. Literature Review

The development of Neural Machine Translation (NMT) systems has rapidly transformed the field of machine translation, leading to significant improvements over traditional approaches.Before the rise of NMT, machine translation largely relied on rule-based systems and Statistical Machine Translation (SMT). Rule-based systems used predefined linguistic rules to translate text, while SMT utilized statistical models trained on parallel corpora to generate translations based on word frequencies and phrases. While SMT improved translation quality over earlier methods, it struggled with handling complex sentence structures, idiomatic expressions, and context. These limitations led to inaccurate translations, especially when translating between languages with significant grammatical differences, like English and Nepali.

MT, introduced in the early 2010s, represents a significant leap in machine translation technology. Unlike SMT, which breaks down translation into small pieces such as words or phrases, NMT models treat translation as a sequence-to-sequence task, where an encoder reads the input sentence, and a decoder generates the translated output. Early NMT systems used Recurrent Neural Networks (RNNs), but more recent advancements have employed Long Short-Term Memory (LSTM) networks to handle long-term dependencies in the data.

LSTM-based models can better capture the meaning and context of a sentence, making them ideal for complex languages like Nepali. This ability to process long sequences of words helps improve the quality of translation, particularly for languages that require understanding entire sentences rather than individual words.

Several researchers have worked on English-to-Nepali translation using NMT techniques, particularly focusing on the challenges posed by Nepali grammar and morphology
In 2020, G. Tiwari, A. Sharma, A. Sahotra, and R. Kapoor [4] developed an English-Hindi Neural Machine Translation model using LSTM-based Seq2Seq and ConvS2S architectures. The system was trained on a large dataset of English and Hindi sentence pairs and demonstrated significant improvements in translation accuracy. Their work highlighted the effectiveness of LSTM and ConvS2S models in overcoming challenges in neural machine translation for complex language pairs, achieving better performance compared to traditional machine translation methods.

In 2024, Shabdapurush Poudel, Bal Krishna Bal, and Praveen Acharya [5] build a Bidirectional English-Nepali translation model . Neural Machine Translation system with an encoder-decoder architecture was used , training it on a legal corpus consisting of 126,000 parallel sentences. This system achieved BLEU scores 7.98 for English to Nepali and 6.63 for English to Nepali translation.

In 2020, Dan Guo, Wengang Zhou, Houqiang Li, and Meng Wang [6] proposed a hierarchical-LSTM (HLSTM) encoder-decoder model for Continuous Sign Language Translation (SLT). The model integrates visual content and word embeddings to address the challenge of word order misalignment in SLT. It uses 3D CNNs to extract spatio-temporal cues from video clips and incorporates adaptive key clip mining to enhance viseme representation. By employing a temporal attention-aware weighting mechanism and additional LSTM layers for semantic translation, the model improves performance on both seen and unseen sentences. The proposed LSTM model demonstrated promising results in reducing encoding time and computational complexity while achieving high accuracy in translating sign language.

In 2018, Acharya and Bal (2018) [7] used a small portion of the parallel corpus from Nepali National Corpus (NNC) collected by Yadava et al. They applied SMT and NMT techniques. On their test sets, the highest BLEU scores they obtained were 5.27 and 3.28 in SMT and NMT respectively English to Nepali using SMT was another project that aimed to translate English to the most likely Nepali sentences applying the SMT (Statistical Machine Translation) approach. This project was able to give accuracy of 68% that is 2.7 on 4. Translation of English to Nepali text using rule based MT taking input as Nepali document and tokenizing to words using Document Tokenize then to Syllabic Tokenizer using Finite State Machine producing equivalent feature structure for target language.[8]

In 2020 , Nemkul and Subarna Shakya [9] build an English - Nepali translation model using LSTMs cell in an encoder-decoder architecture with attention mechanism. The model performance was measured using BLEU score and was able to achieved of BLEU score of 8.9.

CHAPTER 3: SYSTEM ANALYSIS

3.1. System Analysis

System analysis is the process of studying and understanding a system to identify its components, structure, and functionality. It involves examining how a system operates, the flow of data and processes, and how the system interacts with its environment. In context of this translation project, system analysis involves thoroughly understanding the requirements, challenges, and workflow needed to develop an effective translation model using an LSTM-based encoder-decoder architecture. This process includes:

1. Understanding Objectives: The primary goal of the system is to accurately translate Nepali text into English while addressing unique linguistic challenges like grammar structures and vocabulary differences.
2. Requirement Gathering: Identifying the key features of the system, such as data preprocessing (tokenization, normalization), the use of word embeddings and the architecture of the translation model (LSTM-based encoder-decoder).
3. Analyzing Components: Breaking the system into modules such as data preprocessing, vocabulary building, model training, and evaluation. This ensures that each part aligns with the overall project goal.
4. Identifying Challenges: Recognizing issues like handling large datasets (150,000+ rows), ensuring accurate translation for rare Nepali words, and achieving acceptable accuracy scores during evaluation.
5. Proposing Solutions: Addressing challenges through the use of specialized libraries efficiency, and hyperparameter tuning to optimize model performance.

3.1.1. Requirement Analysis

Functional Requirements

• Admin shall train the translation model with new datasets to enhance accuracy.

• Admin shall update the system with improved versions or features.

• Admin shall evaluate the model to ensure quality and performance.

• User shall provide English text input for conversion into Nepali.

• User shall trigger the system to translate English text into Nepali.

• User shall view the translated Nepali text directly within the interface.

• User shall download or access the final translated text output.

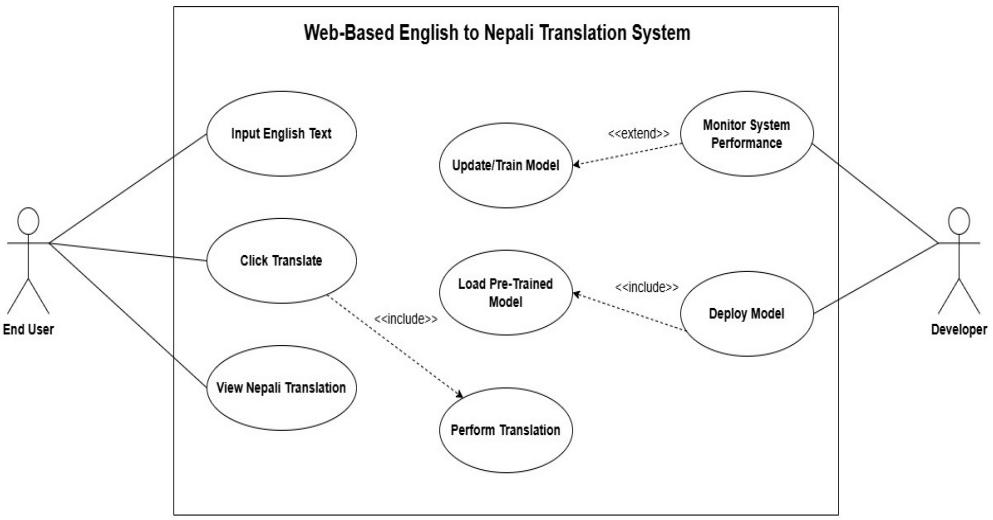


Figure 3.1: Use Case Diagram of English to Nepali Translation Model

Non-Functional Requirements

- Performance:

  The system is designed to deliver translations with maximum accuracy, evaluated using accuracy scores and WER, while ensuring real-time processing. This is particularly critical in contexts such as educational and professional environments where quick responses are essential. To achieve this, the system incorporates optimization techniques and focuses on efficient data handling and real-time processing.

- Usability:

  Usability is prioritized by ensuring that the system is easy and efficient to use. The user interface is simple and intuitive, catering to users with minimal technical skills. A clean and minimalistic layout is implemented, featuring clearly labelled input and output fields for seamless interaction with the translation interface.

- Scalability:

  The system is scalable to handle large datasets efficiently. Techniques such as distributed computing, data compression, and batch processing are employed. The use of batch processing enables the handling of large datasets in chunks, either in parallel or sequentially in smaller subsets, improving memory management and speeding up data processing.

3.1.2. Feasibility Analysis

i. Technical Feasibility

This project was successfully implemented using existing technologies and resources. The technical requirements fulfilled for this project included:

- A laptop/computer with at least 8GB RAM and a GPU to ensure efficient processing during model training and testing.
- High-speed internet was utilized for downloading the dataset and accessing necessary online resources.
- Python, along with essential libraries like PyTorch, numpy and Pandas, was used for developing and training the LSTM-based encoder-decoder models.

ii. Operational Feasibility

The project was completed by a small team of three developers, demonstrating its operational feasibility. The system for translating texts from English to Nepali was successfully created using LSTM and an encoder-decoder architecture, achieving accuracy and efficiency within the project's scope.

iii. Economic Feasibility

The project proved to be economically feasible as it leveraged free and open-source software like Python and pytorch for development. The computational requirements were met using existing resources, and free hosting services were utilized where applicable. The resulting benefits, including an effective translation system, outweighed the minimal costs, making the project cost-effective for future use.

iv. Schedule Feasibility

The project was completed within the estimated timeline, demonstrating its schedule feasibility. The timeline was planned by considering potential constraints, including technical challenges, budget limitations, and resource availability. Despite minor adjustments, the project proceeded as scheduled, and all key milestones were achieved on time, ensuring successful completion without significant delays.

Table 3.1: Project Schedule of English to Nepali Translation Model

Task	Start Date	Days
Initiation	24/08/2024	3
Data Collection and Preprocessing	27/08/2024	10
Initial model design and implementation	06/09/2024	14
Model training and evaluation	20/09/2024	10
Iteration Review and model Improvement	30/09/2024	14
Model retraining and re-evaluation	14/10/2024	10
Final model training and evaluation	24/10/2024	12
UI design, Integration and Testing	05/11/2024	7
Feedback, Refinement and Closure	12/11/2024	7
Documentation	24/08/2024	87

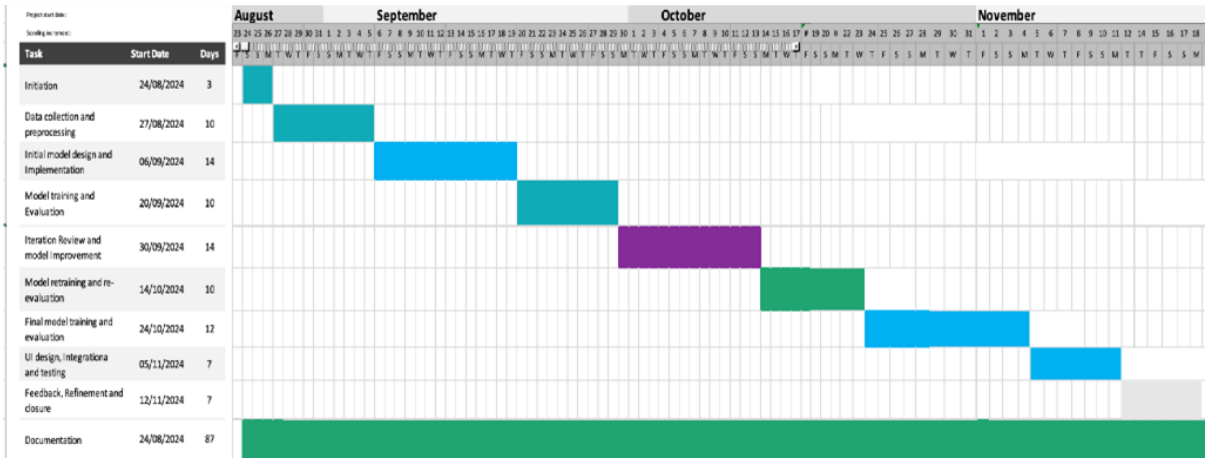


Figure 3.2: Gantt Chart of English to Nepali Translation Model

3.1.3. Analysis

- **Class and Object Diagram**

This class diagram illustrates the structure and relationships among key components in a machine translation project using LSTM-based models, integrated with a Django web framework. The top-level class, Django, represents the web framework used for managing routing, views, and settings for the application. It provides methods such as route() for defining URL patterns and render() for rendering templates with context data. Django manages interactions with the Lang class, which encapsulates the logic for language processing, including vocabulary generation and preprocessing. The relationship between Django and Lang is depicted as a "manages" relationship, indicating that Django controls multiple instances of Lang.

The Lang class is central to this project as it handles core functionalities like tokenization, word-to-index and index-to-word mappings, vocabulary trimming, and string normalization. It also includes methods for preparing datasets for translation by dividing them into training and testing subsets. Lang interacts with the Encoder and Decoder classes, which represent the LSTM-based Encoder-Decoder model. These classes include attributes like hidden_size, vocab_size, and embedding, along with methods for initialization (__init__), forward propagation, and creating hidden states. The Encoder and Decoder are connected to Lang via "uses" relationships, emphasizing that they rely on the language data

processed by Lang. Together, these classes form the foundational architecture for training and evaluating the translation model while integrating seamlessly with the web framework for deployment.

Following is the class diagram for the integration of various components in the language translation system. In this, Django class provides a web framework which is responsible for routing and provides views and settings for an application. Communication with the user is handled via a route() method for routing the request and appropriate responses via the render() method. The Lang class would contribute the central location of preprocessed, managed language data, which in turn would have the attributes: language_name, word_to_index, index_to_word, and vocab_size, applied in mapping and handling words within the vocabulary. The methods then would include addSentence(), addWord(), and prepareData() to build a clean and well-structured dataset for either training or inference.

The Lang class links to the neural network components that would, for example, be implemented by classes such as Encoder and Decoder, respectively, for the processing of language data into its translation. Similarly, both hold some attributes such as hidden_size and vocab_size which describe the model architecture, and methods such as forward(), init_hidden() for propagating data through a network and initializing the hidden state of it.

The relationships between these components are pretty clear: Django manages one or more Lang instances, so that the application can support multiple languages; Lang depends on Encoder and Decoder for the core translation logic. This modularity makes it flexible in such a way that the neural network and web framework can be adapted or extended independently.

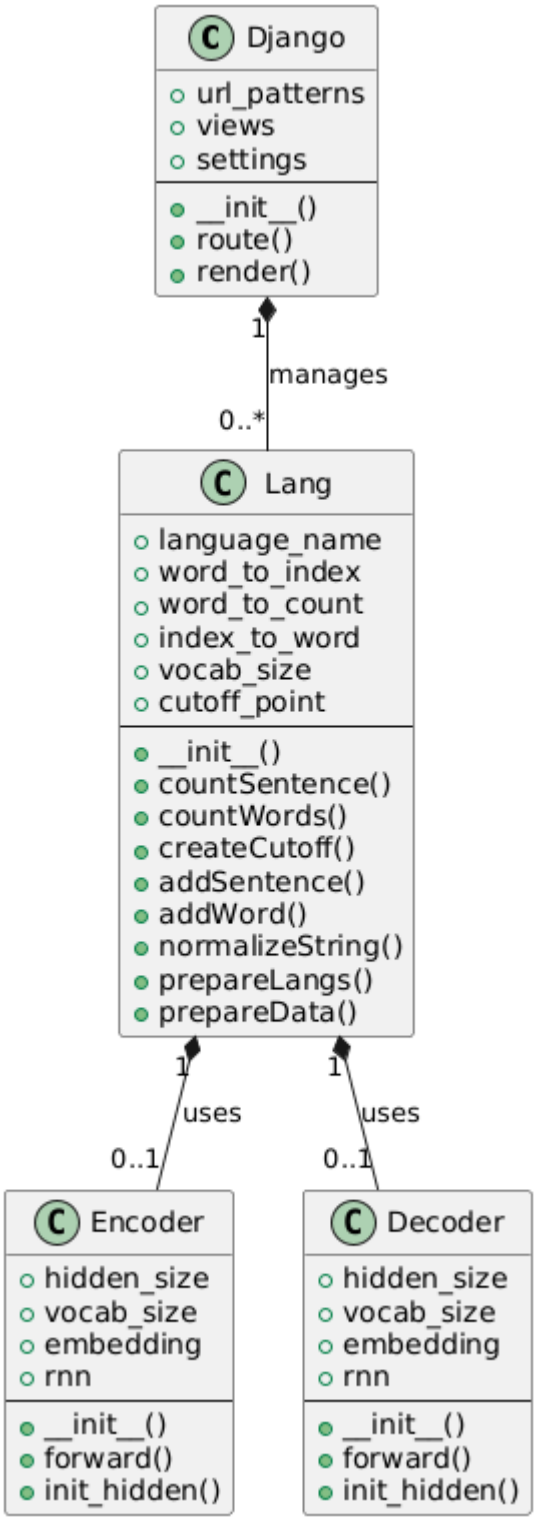


Figure 3.3: Class Diagram of English to Nepali Translation

- **State and Sequence Diagram**

A state diagram (also known as a state machine diagram or state chart diagram) is a type of behavior diagram in Unified Modeling Language (UML) used to model the dynamic behavior of a system by showing its various states and how the system transitions from one state to another based on events or conditions. It is particularly useful for representing objects or components that can be in one of many states during their lifecycle.

- The user starts through input of English text.
- The input text is processed and passed to the translation model.
- The pretrained model is loaded in the background.
- The model processes the input and returns the Nepali translation, which is displayed to the user.

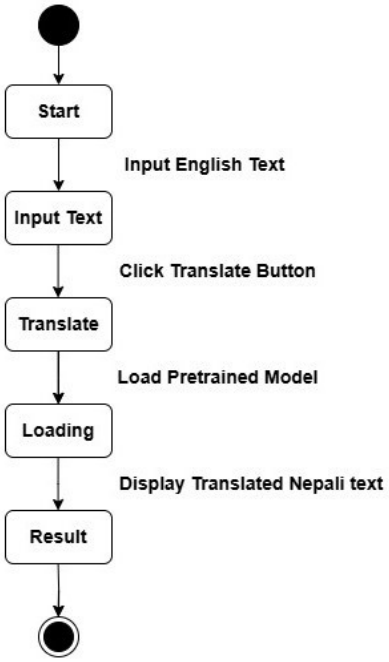


Figure 3.4: State Diagram of English to Nepali Translation Model

A sequence diagram is a type of interaction diagram in Unified Modeling Language (UML) that shows how objects or components interact with each other in a particular sequence to perform a specific task or process. It focuses on the order of messages exchanged between objects over time.This sequence diagram describes the flow of communication between different components of the system:

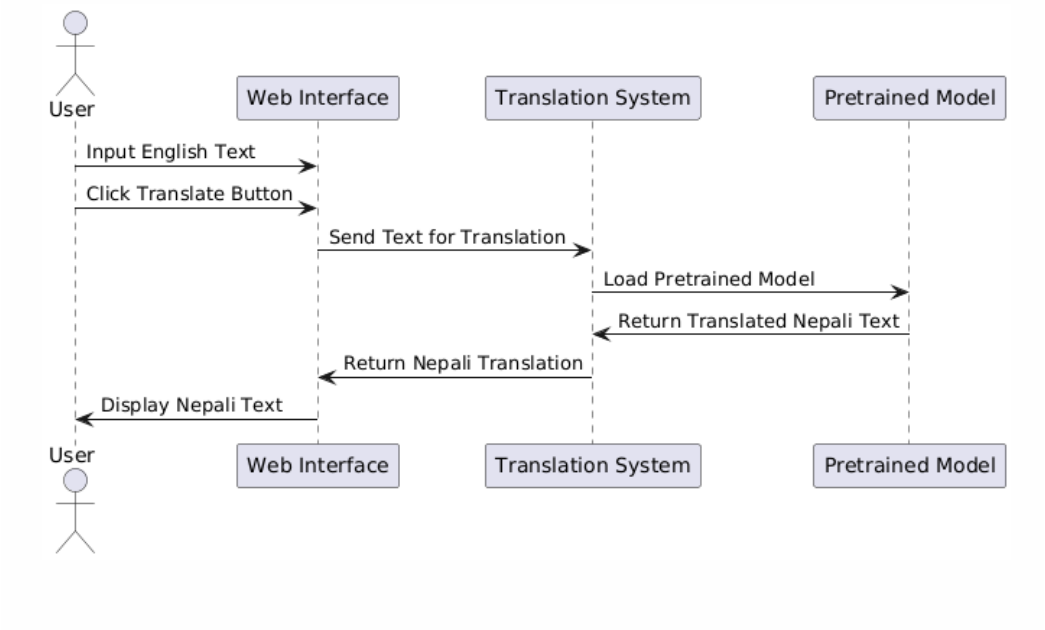


Figure 3.5: Sequence Diagram of English to Nepali Translation Model

- **Activity Diagram**

  An activity diagram is a type of behavior diagram in Unified Modeling Language (UML) that illustrates the flow of control or activities within a system or process. It represents the sequence of actions and decisions that occur as a part of a workflow or process. Activity diagrams are particularly useful in modeling the dynamic aspects of a system, showing the flow of control from one activity to another.

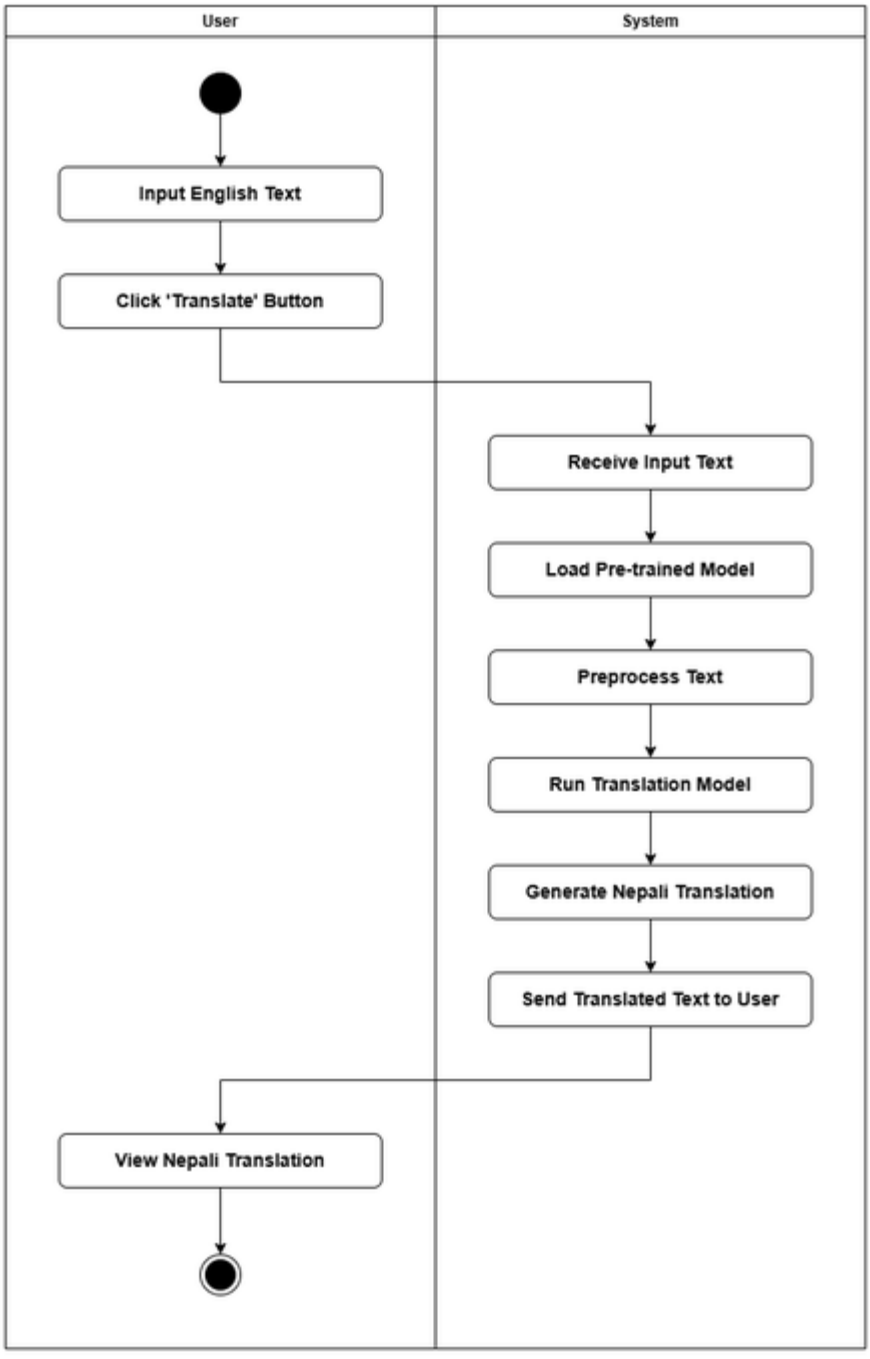


Figure 3.6: Activity Diagram of English to Nepali Translation Model

CHAPTER 4: SYSTEM DESIGN

4.1. Design

- **Refinement of class Diagram**

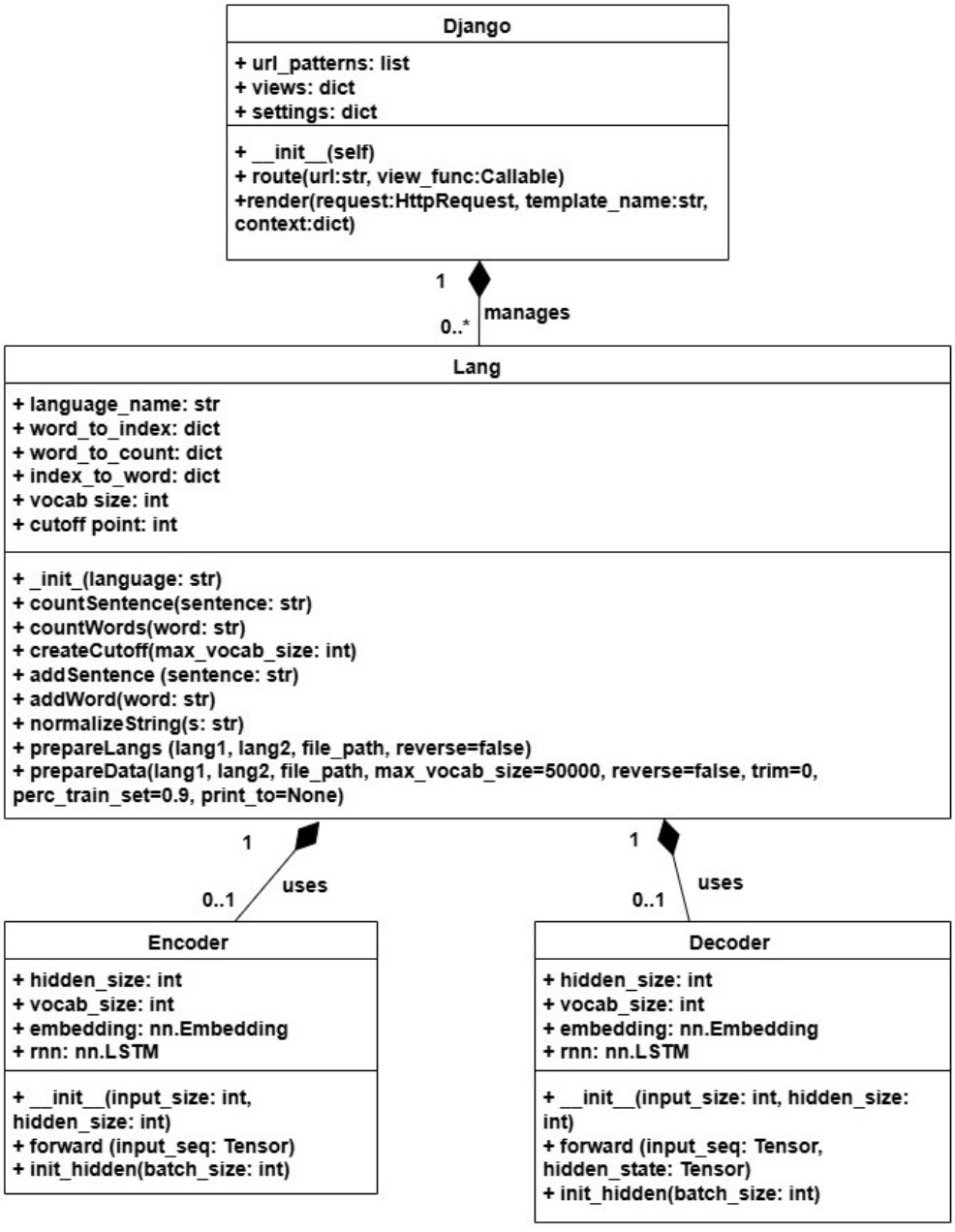


Figure 4.1: Refinement of class Diagram of English to Nepali Translation Model

- **Refinement of state diagram**

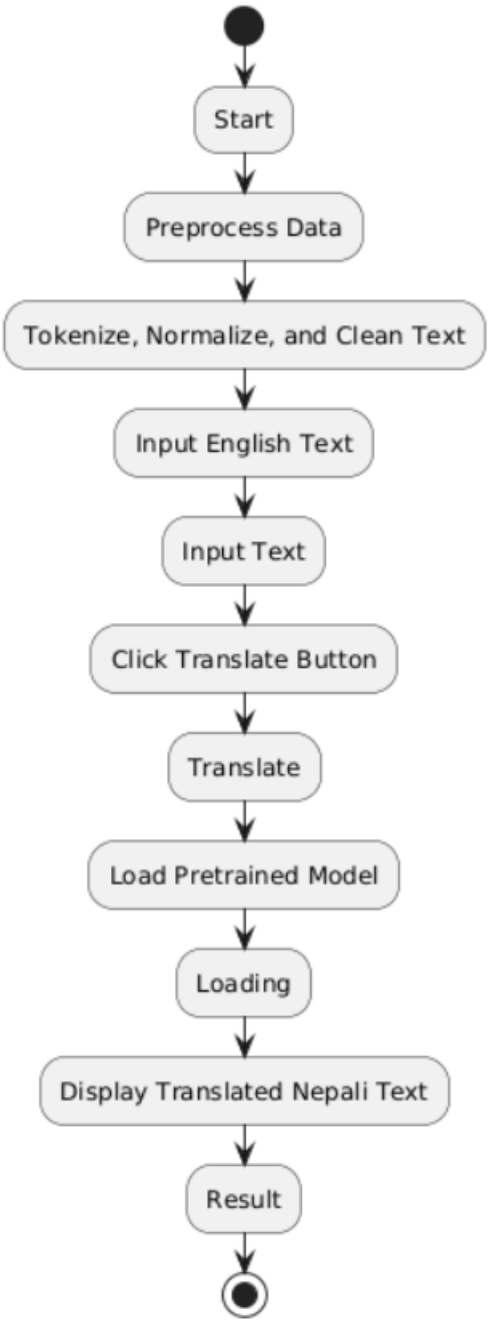


Figure 4.2: Refinement of state diagram of English to Nepali Translation Model

- **Refinement of sequence diagram**

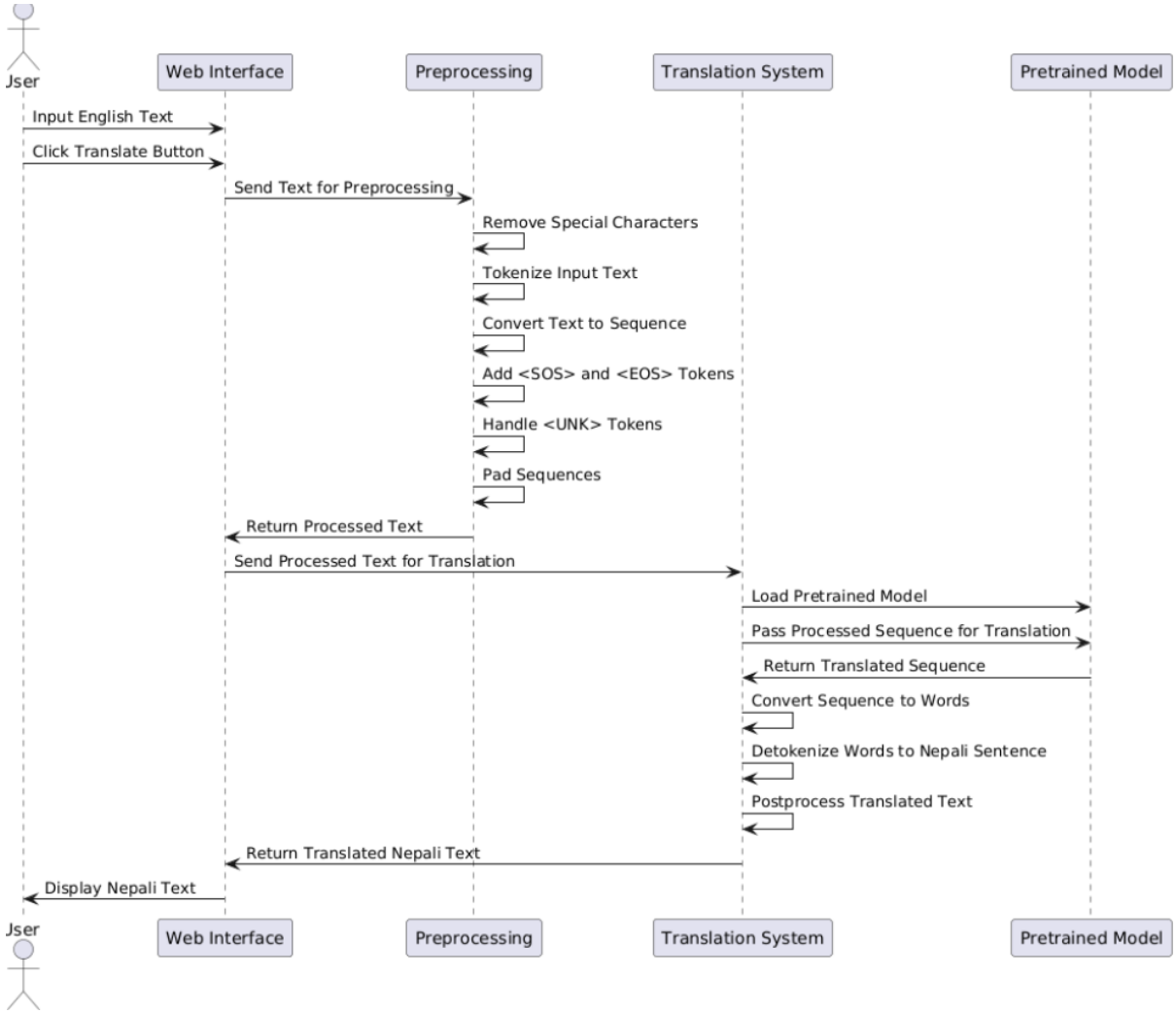


Figure 4.3: Refinement of sequence diagram of English to Nepali Translation Model

1. Deployment Diagram

This deployment diagram illustrates the integration of a machine translation model using Django for providing English-to-Nepali translations via a web interface. The system is divided into several modular components. The user interacts with a Django Web Interface, comprising HTML, CSS, and JavaScript for a seamless front-end experience. User input (English text) is sent to the Django API, which manages backend logic through views.py, urls.py, and settings.py. This API processes requests and routes them to the translation service. The

translation service (translation_model.py) is responsible for generating the translated output using a pre-trained neural translation model.

The model weights are pre-loaded and stored in the Model Storage component (model_weights.h5). The Model Loader (load_model.py) ensures the pre-trained model is loaded into memory before processing translation requests. When the user submits text for translation, the API checks if the model is already loaded and passes it to the translation service. The generated Nepali translation is then returned to the Django Web Interface and displayed to the user. This design decouples the translation logic from the user interface, ensuring efficient deployment and a clear separation of responsibilities between front-end, API logic, and machine learning components.

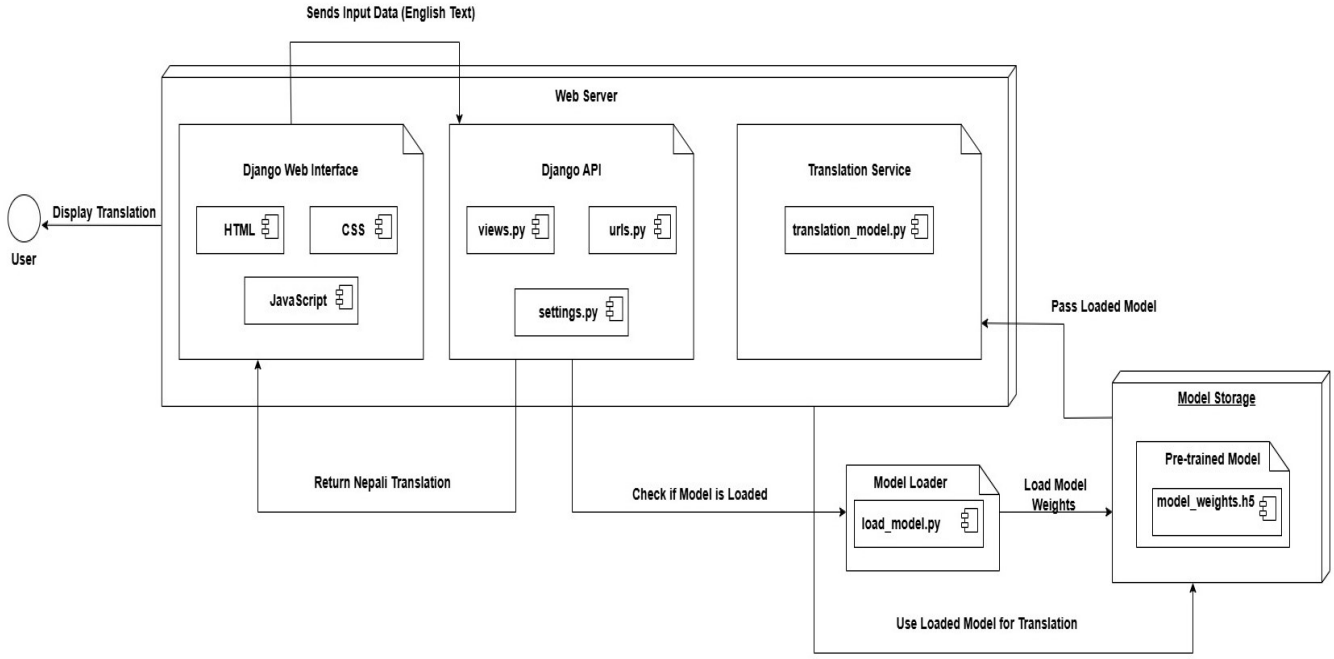


Figure 4.4: Deployment Diagram of English to Nepali Translation Model

2. Flowchart of Model building

The flowchart in figure below represents the process of building a machine translation (MT) model using an LSTM-based Encoder-Decoder architecture. It begins with refining and preprocessing the dataset, ensuring it is clean and structured for training. The dataset is then split into training and testing subsets. The next step involves initializing the LSTM Encoder-Decoder model, which is trained using the training dataset and evaluated using both the training and testing subsets. The results are used to establish the final model. The WER score is calculated to assess the translation quality, followed by a final evaluation of the model's performance. The output of this process is the translated text generated by the trained model.

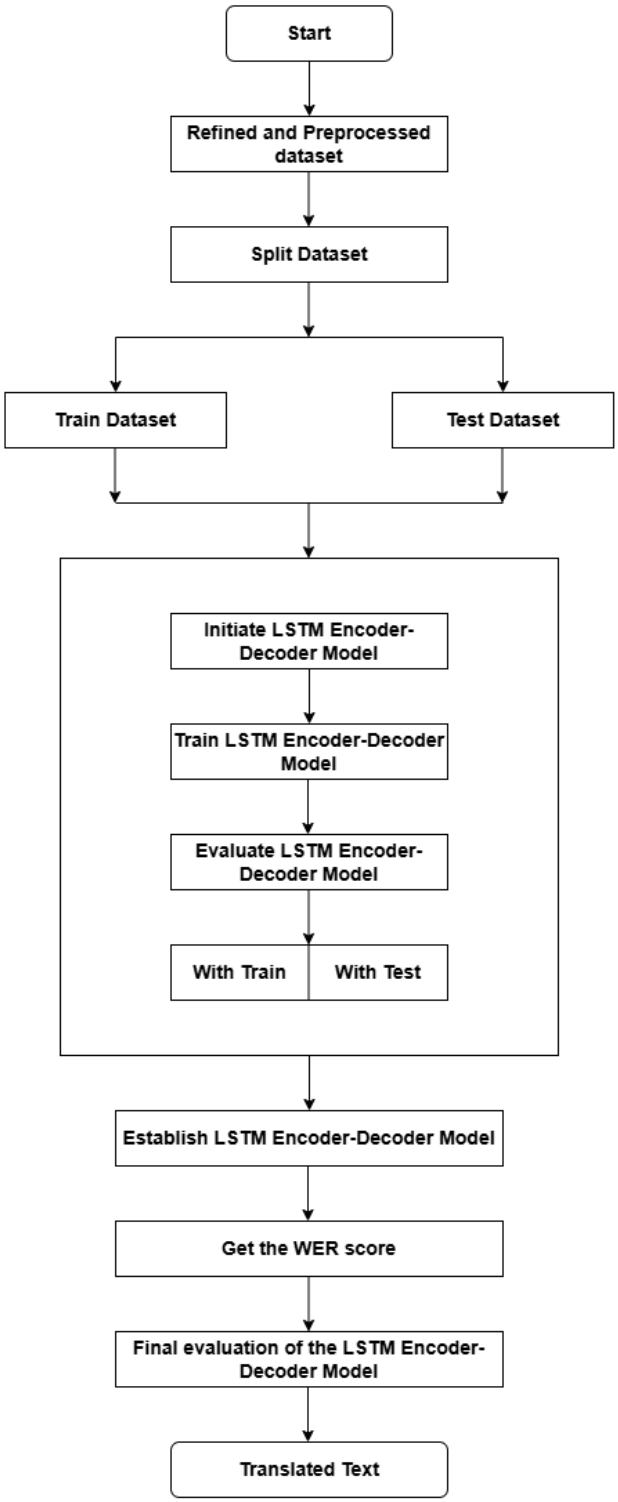


Figure 4.5: Flowchart of English to Nepali Translation Model

4.2. Algorithm Details

The system implements a machine translation model based on the Encoder-Decoder architecture with Long Short-Term Memory (LSTM) cells and incorporates a global attention mechanism to improve translation quality. This approach is well-suited for handling the sequential nature of language data and resolving long-term dependencies.

The key steps in the algorithm include:

1. Preprocessing of input sentences to tokenize and prepare them for the model.
2. Encoding the source sentence into a context vector using an LSTM-based encoder.
3. Decoding the context vector into the target sentence using an LSTM-based decoder with attention.
4. Training the model using teacher forcing and evaluating it using metrics like WER.

Encoder Decoder Architecture

The encoder is responsible for processing the input sentence $X = x_1, x_2, \ldots., x_T$ where each x_i is a token in the source language. The encoder processes this sequence token-by-token using a recurrent neural network (RNN) with Long Short-Term Memory (LSTM) units to produce a sequence of hidden states

$$H^E = \{h_1^E, h_2^E, \ldots, H_T^E\}$$

Each input token xt is first converted into a one-hot vector, v_t of size V, where V is the vocabulary size. This one-hot vector is then mapped to a dense embedding vector $e_t \in \mathbb{R}^d$ using an embedding matrix $W^E \in \mathbb{R}^{v*d}$

$$e_t = W^E . v_t$$

Here, e_t is the input embedding of dimension d.

Sequence Encoding with LSTM

The LSTM processes the input sequence $e_1, e_2, \ldots, e_T$in a step-by-step manner. At each timestep t, the LSTM updates its hidden state $h_t^E$and cell state c_t^E based on current input embedding e_t, the previous hidden state h_{t-1}^E , and the previous cell state c_{t-1}^E.

The computations at each timestep t are as follows:

1. **Forget gate:**

The forget gate determines which parts of the previous cell state c_{t-1}^{E} to retain as

$$f_t = \sigma\left(W_f . [h_{t-1}^{E}, e_t] + b_f\right) \qquad [10]$$

2. Input Gate:

The input gate decides which parts of the new candidate state $c_t^{\sim}$ to incorporate. It can be described by mathematical relation as

$$i_t = \sigma(W_i . [h_{t-1}^{E}, e_t] + b_i)$$

$$c_t' = tanh(W_c . [h_{t-1}^{E}, e_t] + b_c) \qquad [10]$$

3. Cell state update:

The new cell state ctE combines retained information from the previous state and new candidate values.

$$c_t^{E} = f_t \odot c_{t-1}^{E} + i_t \odot c_t \sim [10]$$

4. Output Gate:

The output gate determines the hidden state htE that will be passed to the next timestep.

$$o_t = \sigma(W_o . [h_{t-1}^{E}, e_t] + b_o)$$

$$h_t^{E} = o_t \bigodot tanh(c_t^{E})$$

Here σ is the sigmoid activation function, tanh is the hyperbolic tangent function, $\odot$ is the element-wise multiplication, and W_f, W_i , W_c , W_o are weight matrices, while b_f, b_i, b_c, b_o are biases. At the end of the input sequence (t = T), The encoder produces the final hidden state h_t^{E} , which acts as the context vector summarizing the entire input sequence:

$$ContextVector: h_T^{E} = h_o^{D}$$

This context vector is passed to the decoder as the initial hidden state h_o^{D}, enabling the decoder to generate the target sentences.

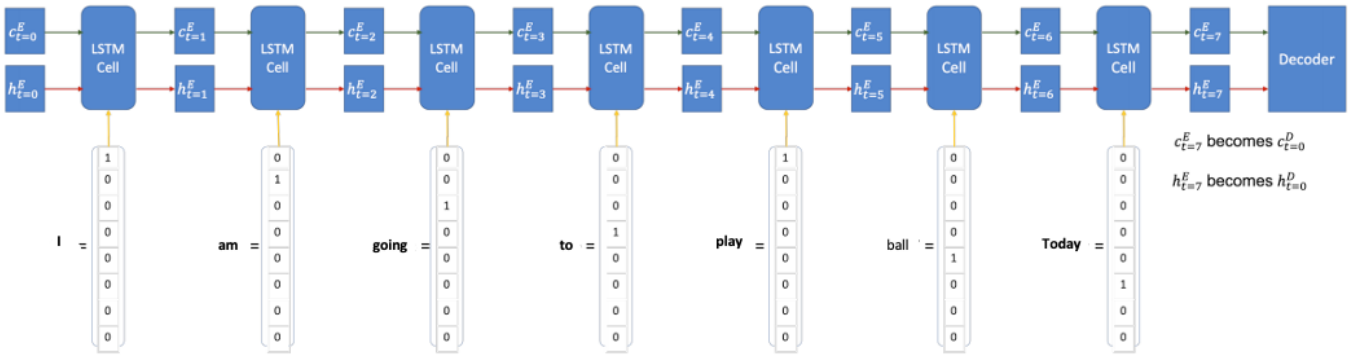


Figure 4.6: Encoder Part of English to Nepali Translation Model

In the decoder portion of the architecture, the training process involves iteratively updating the model parameters to minimize the CrossEntropy Loss using the Stochastic Gradient Descent (SGD) optimization algorithm. Here's how this process works in conjunction with the decoder. At each timestep t, the decoder takes the hidden state h^D_{t-1} (from the previous timestep or initialized using the encoder's context vector at t = 0) and the embedding of the current token x_t to produce the updated hidden state h^D_t. This can be represented as:

$$h^D_t = DecoderLSTM(x_t, h^D_{t-1})$$

Here, x_t is the embedding of the token from the vocabulary.

The updated hidden state h^D_t is passed through a linear transformation (a weight matrix W_{out} and bias b_{out}) followed by a softmax function to compute the probability distribution

$$y'_t = softmax(W_out.h^D_t + b_out)$$

The most probable token y'_t is selected during inference, but during training, the true token y_t (from the ground truth sequence) is fed as input for the next time step using the teacher forcing mechanism. The CrossEntropy Loss between the predicted probability distribution y'_t and the true target token y_t is calculated as

$$l_t = -\sum_{k=1}^{K} y^k_t . log(y_t)'^k \text{ [11]}$$

Here, K is the vocabulary size, y_t^k is the true label (1 if the token is correct, 0 otherwise), and $y_t'^K$ is the predicted probability for token k. The total loss L for the sequence is the sum of the losses at all timesteps

$$L = \sum_{t=1}^{T} l_t$$

Where, T is the length of the target sequence

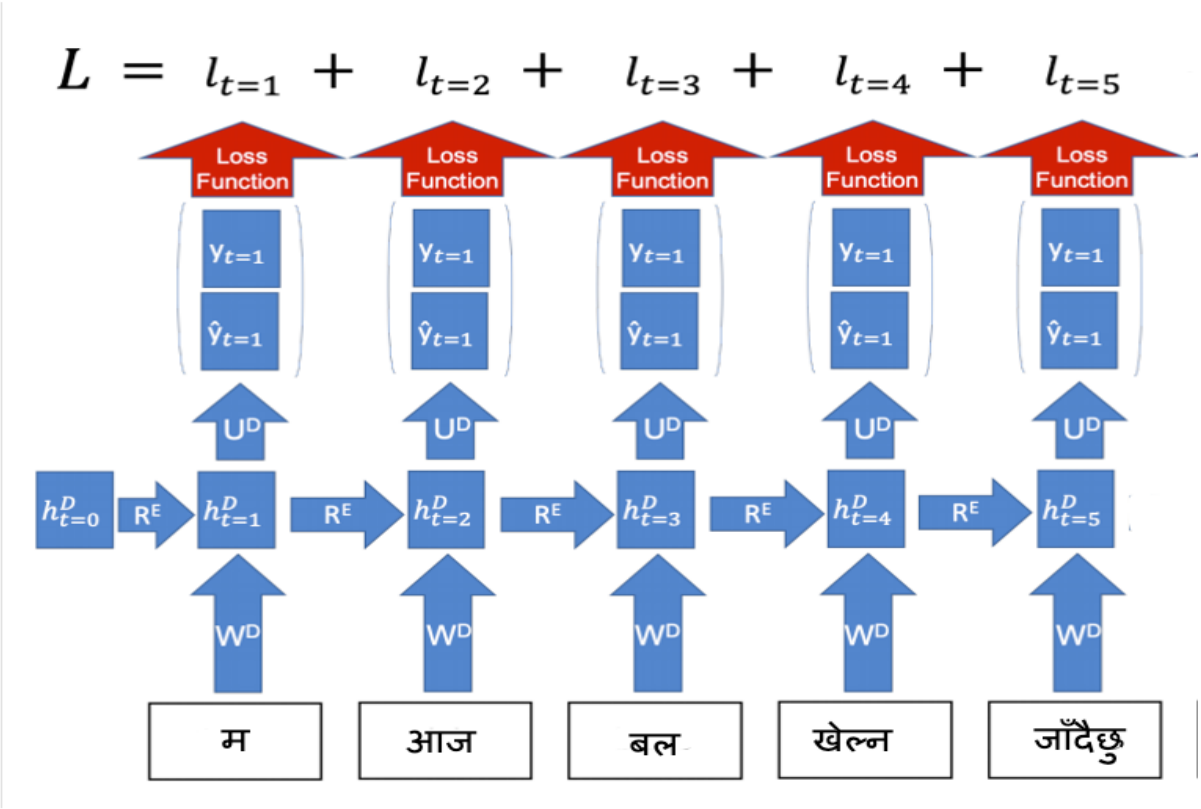


Figure 4.7: Decoder Part of English to Nepali Translation Model

SGD for weight Updates

SGD is used to minimize the loss L by updating the model's trainable parameters (weights and biases) based on the computed gradients. Gradients of the loss L with respect to the trainable parameters W and b are calculated using backpropagation through time (BPTT) like $\frac{\partial L}{\partial w_i^E}, \frac{\partial L}{\partial w_f^E}, \frac{\partial L}{\partial w_o^E}, \frac{\partial L}{\partial w_i^D}, \frac{\partial L}{\partial w_o^D}$. These gradients indicate how much each parameter contributes to the loss and in which direction they need to be adjusted.

Once gradients are computed, SGD updates the parameters using the following formula

$$\theta \leftarrow \theta - \eta . \frac{\partial L}{\partial \theta}$$

Here,

θ represents the trainable parameter like $W_i^E, W_i^D, W_o^E, W_o^D$, etc.

η is the learning rate

$\frac{\partial L}{\partial \theta}$ is the gradient of the loss with respect to the parameter.

The process of gradient computation and weight updates is repeated for multiple epochs (iterations over the entire dataset) until the model converges, i.e., the loss L reaches a minimum.

Global Attention mechanism

The attention mechanism computes a context vector at each decoder timestep t, which summarizes the relevant information from all encoder hidden states. This allows the decoder to consider the most relevant parts of the input sequence when generating a target token. [3] The encoder processes the input sequence and outputs a sequence of hidden states

$$h_t^E \quad for\ t = 1,2, \dots \dots, T$$

Where T is the length of the input sequence. These hidden states represent contextual information for each word in the input. At each decoder timestep t, the attention scores are computed using the dot product between the current decoder hidden state h_t^D and each encoder hidden state h_t^E

$$score(h_t^D, h_i^E) = h_t^D . h_i^E$$

The score measures how relevant each encoder hidden state h_i^E is to the current decoder state h_t^D . The scores are normalized using the softmax function to compute the attention weights:

$$a_{t,i} = exp\left(score(h_t^D, h_i^E)\right) / \sum_{j=1}^{T} exp\left(score\left(h_t^D, h_j^E\right)\right) \qquad [12]$$

Here,

- $a_{t,i}$ represents how much attention the decoder should pay to the i-th encoder hidden state at timestep t .
- The weights $a_{t,i}$sum to 1 across all encoder hidden states:

The context vector c_t^D is computed as the weighted sum of all encoder hidden states using the attention weights. This context vector encapsulates the most relevant information from the input sequence for timestep t. The context vector c_t^D is concatenated with the decoder hidden state h_t^D2 and passed through a feedforward layer (often followed by a softmax) to predict the next token.

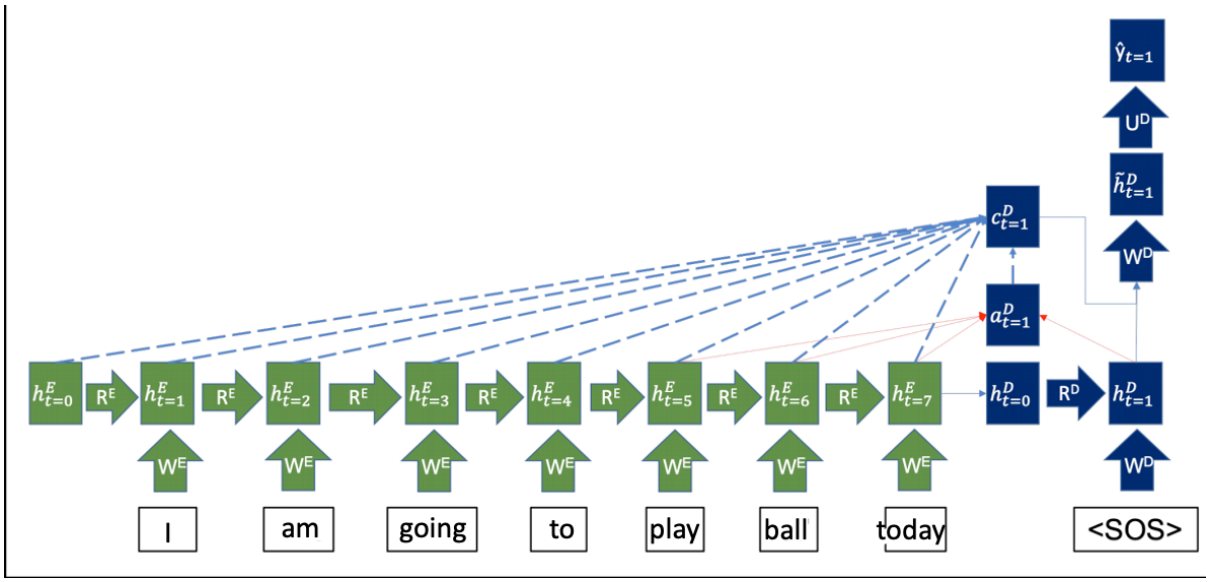


Figure 4.8: First step of Decoder with Global Attention Model

CHAPTER 5: IMPLEMENTATION AND TESTING

5.1. Implementation

5.1.1. Tools Used

In this section, we provide a detailed explanation of the tools, technologies, and methods used to implement the English to Nepali translation system. The chapter also explains the individual modules, classes, procedures, functions, and algorithms that were implemented. The following tools and technologies were used during the implementation phase of the English to Nepali translation system:

1. **IDE and Development Tools:**
    - **VS Code:** The primary development environment used for coding and debugging the project. VS Code is equipped with various extensions like Python, PyTorch, and Django support, making it an ideal tool for development and testing.
2. **Programming Languages:**
    - **Python:** The core programming language for implementing the machine learning model. Python libraries such as PyTorch and Matplotlib were used for building and training the LSTM-based encoder-decoder model for machine translation and visualizing the training process.
    - **HTML/CSS/JavaScript:** Used for developing the web-based frontend where users can interact with the system by inputting text and receiving translations.
    - **Django:** A high-level Python web framework used for developing the backend of the web application. It facilitates handling user requests, loading the trained model, and providing responses.
3. **Libraries and Frameworks:**
    - **PyTorch:** Used to define, train, and evaluate the LSTM-based sequence-to-sequence model. It offers powerful tools for deep learning, which is essential for building a translation model.

- **NLTK:** The Natural Language Toolkit is used for text processing tasks such as tokenization and lemmatization, especially for English text.
- **Matplotlib:** Used for visualizing training metrics like loss curves and WER scores during the model training process.
- **NumPy:** This library was used extensively for numerical operations, such as handling matrices and arrays, which are common when dealing with large datasets and machine learning models. Specifically, NumPy was used to manage and perform operations on the data in array form, particularly when manipulating the tokenized data, which often requires handling sequences of integers in a structured way (e.g., padding sequences to the same length).
- **Pandas:** Pandas was used for managing and processing structured data. It provides efficient data structures like DataFrames, which were particularly useful for storing, cleaning, and transforming the raw parallel English and Nepali sentences into the necessary formats. It also facilitated reading and writing data from and to CSV, JSON files, which is essential for handling large datasets in this project.

5.1.2. Implementation Details of Modules

The system consists of multiple modules that interact with each other to form the final translation pipeline. Below are the key modules, classes, and functions implemented for the translation system:

- **normalizeString(s)**

  This function performs text normalization for any input string s. It removes unwanted special characters (such as punctuation, currency symbols, emojis, and other non-standard characters) from the string.

- **filterPair(p, max_length)**

  This function filters sentence pairs based on their length. It ensures that both the input and output sentences in a pair do not exceed a specified maximum length (max_length).

This is crucial for handling longer sequences that may disrupt training or model performance.

- **filterPairs(pairs, max_length)**

  This function processes a list of sentence pairs, applying the filterPair function to filter out any pair where one or both sentences exceed the specified length. It returns a list of valid pairs.

- **Lang Class**

  The Lang class is designed to store and manage the vocabulary of a specific language (either the input or output language for translation). It includes methods to build and manage the vocabulary, count word frequencies, and handle the addition of special tokens (such as SOS, EOS, and UNK).

  Key Methods:

  countSentence(sentence): Counts the frequency of words in a given sentence.

  countWords(word): Updates the frequency of individual words.

  createCutoff(max_vocab_size): Establishes a cutoff point for vocabulary size based on word frequencies.

  addSentence(sentence): Adds a sentence to the vocabulary, converting each word into a unique index.

  addWord(word): Adds a word to the vocabulary if it meets the frequency threshold and assigns it a unique index.

  Attributes:

  - word_to_index: A dictionary mapping words to unique indices.
  - word_to_count: A dictionary mapping words to their frequency counts.
  - index_to_word: A dictionary mapping indices back to words.
  - vocab_size: The total size of the vocabulary.

- **prepareLangs(lang1, lang2, file_path, reverse=False)**

  This function prepares the Lang objects for both languages (lang1 and lang2). It reads sentence pairs from the given file paths, normalizes the sentences, and creates a list of pairs. If the reverse flag is set to True, the source and target languages are swapped.

- **prepareData(lang1, lang2, file_path, max_vocab_size=50000, reverse=False, trim=10, perc_train_set=0.8, print_to=None)**

  This function prepares both input and output languages, counts word frequencies, applies a vocabulary size cutoff, filters pairs, and splits the data into training and test sets. It returns the processed data along with vocabulary details.

  Key Steps:

  - Calls prepareLangs to get the Lang objects and sentence pairs.
  - Filters the pairs based on the specified trim value.
  - Counts word frequencies for both languages.
  - Applies a cutoff to the vocabulary size using createCutoff.
  - Splits the dataset into training and testing pairs based on the percentage specified (perc_train_set).

- **indexesFromSentence(lang, sentence)**

  This function converts a sentence into a list of indices corresponding to the words in the sentence. If a word is not found in the vocabulary, it is replaced with the index of the unknown token (<UNK>).

  Key Steps:

  - Splits the sentence into words.
  - Maps each word to its index in the vocabulary.
  - If the word is not in the vocabulary, it uses the <UNK> token index.

- **tensorFromSentence(lang, sentence)**

  This function converts a sentence into a tensor of indices, with an additional EOS (End of Sentence) token appended at the end. The resulting tensor is used for training the model. It also handles the transfer of the tensor to GPU if CUDA is enabled.

Key Steps:

- Converts the sentence into word indices using indexesFromSentence.
- Appends the EOS token index to signal the end of the sentence.
- Converts the list of indices into a PyTorch tensor.

- **tensorsFromPair(input_lang, output_lang, pair)**

  This function converts a pair of sentences (input and target) into PyTorch tensors suitable for training. It uses the tensorFromSentence function to convert both the input and output sentences.

  Key Steps:

  - Converts both input and output sentences into tensors using tensorFromSentence.

- **sentenceFromTensor(lang, tensor)**

  This function converts a tensor of indices back into a human-readable sentence. It maps the indices in the tensor to their corresponding words and returns the sentence.

  Key Steps:

  - Maps each index in the tensor to a word using the index_to_word dictionary.
  - Joins the words into a sentence.

- **batchify(data, input_lang, output_lang, batch_size, shuffle_data=True)**

  This function divides the dataset into batches of a specified size (batch_size). It optionally shuffles the data and computes the longest input and target sequences for each batch.

- **pad_batch(batch)**

  This function pads the input and target sequences in a batch to ensure that all sequences in the batch have the same length. Padding is done using the EOS token.

- **EncoderRNNManual Class**

Purpose: This class defines the encoder part of the sequence-to-sequence model, which processes the input sequence and returns hidden states that the decoder will use. It is based on a manual LSTM (Long Short-Term Memory) implementation.

Constructor (__init__):

- Parameters:
  - input_size: The number of features in the input.
  - hidden_size: The size of the hidden state in the LSTM.
  - bidirectional: A boolean that determines whether the encoder LSTM is bidirectional.
  - layers: Number of LSTM layers.
  - dropout: Dropout rate applied to the input to prevent overfitting.
- Attributes:
  - Initializes an embedding layer for input, a dropout layer, and an LSTM cell for sequence processing.
  - Sets up a linear layer to combine outputs from the bidirectional LSTM.
- Methods:
  - __init__: Initializes the encoder with the given parameters.
    - forward(input_data, h_hidden, c_hidden):
    - Takes the input sequence and the initial hidden states (h, c) of the LSTM.
    - Passes the input through the embedding and dropout layers, then through the LSTM.
    - Returns the hidden states and the LSTM outputs.
    - create_init_hiddens(batch_size):
    - Initializes the hidden and cell states for the LSTM to zeros, and optionally moves them to GPU if available.

- **DecoderAttnManual Class**

Purpose: This class defines the decoder with attention mechanisms, responsible for generating output sequences based on the encoded input sequence. It uses an LSTM and

calculates attention scores to focus on relevant parts of the encoder's hidden states during decoding.

Constructor (__init__):

- Parameters:
  - hidden_size: Size of the hidden state for the decoder's LSTM.
  - output_size: Size of the output vocabulary.
  - layers: Number of LSTM layers.
  - dropout: Dropout rate applied to the input to prevent overfitting.
  - bidirectional: Determines whether the decoder's LSTM is bidirectional.
- Attributes:
  - Similar to the encoder, the decoder has an embedding layer, dropout, LSTM cell, a score learner for attention, a context combiner to combine attention context and LSTM output, and output layers for prediction.

Methods:

- __init__: Initializes the decoder with the specified parameters.
- forward(input_data, h_hidden, c_hidden, encoder_hiddens):
  - Takes the current input token, the hidden states from the previous timestep, and the encoder's hidden states.
  - Runs the LSTM cell and computes attention scores based on the encoder's outputs.
  - Uses the attention mechanism to compute a context matrix and combines it with the LSTM output to form the final hidden state.
  - Computes the predicted token probabilities using a softmax and returns them.

Training Functions

train_batch () Function:

Purpose: This function is responsible for training on a single batch of data, computing the loss, performing backpropagation, and updating the model parameters.

Parameters:

- input_batch: The batch of input data for the encoder.

- target_batch: The batch of target data for the decoder.
- encoder: The encoder model.
- decoder: The decoder model.
- encoder_optimizer: Optimizer for the encoder.
- decoder_optimizer: Optimizer for the decoder.
- criterion: The loss function (CrossEntropyLoss).
- device: The device to run the model on ('cuda' or 'cpu').

Steps:

1. Moves the input and target batches to the device (GPU/CPU).
2. Zeroes the gradients of the encoder and decoder optimizers.
3. Initializes the encoder hidden states and runs the input through the encoder.
4. For each time step of the target sequence, performs a forward pass through the decoder using teacher forcing (where the target output is used as the next input).
5. Computes the loss and adds it to the total loss.
6. Performs backpropagation and clips the gradients to prevent exploding gradients.
7. Updates the model parameters using the optimizer.

Train () Function:

Purpose: This function trains the model for a full epoch (over multiple batches) and computes the average loss.

Parameters:

- train_batches: A list of training batches.
- encoder: The encoder model.
- decoder: The decoder model.
- encoder_optimizer: The encoder's optimizer.
- decoder_optimizer: The decoder's optimizer.
- criterion: The loss function.
- output_lang: The output language object (used to fetch the "SOS" token).
- device: The device to run the model on ('cuda' or 'cpu').

Steps:

1. Loops over each training batch, padding the batch and moving it to the specified device.

2. Computes the batch loss using the train_batch function.
3. Accumulates the total loss for the epoch and returns the average loss.

5.2. Testing

Testing is the process of determining whether the system works effectively and efficiently. Testing does not only include debugging. It also checks for quality assurance, validation and verification, reliability and estimation.

5.2.1. Test Cases for Unit Testing

Unit Testing is the process of checking small piece of code to ensure that the individual parts of a program work properly on their own. Unit Testing result is given below:

Table 5.1: Test Cases for Unit Testing of English to Nepali Translation Model

Function	Input	Expected result	Result
normalizeString	Why didn't <> it renew? किन भएन नवीकरण?	why didnt it renew किन भएन नवीकरण	why didnt it renew किन भएन नवीकरण
prepareLangs	lang1, lang2, file_path	Return input and out lang object with sentence pairs	Input_lan, output_lan, pairs were returned
prepareData	lang1, lang2, file_path, max_vocab_size, trim, perc_train_set	Read sentence pair and split into train and test	Train and test sets get splitted

tensorFromSentence	I am a boy	One hot encoding of input	Sentences were represented using One-Hot vectors
sentenceFromTensor	tensor of one hot encoding vector	converts from tensor indices to sentence	Sentences were generated from tensors
batchify	data, batch_size, shuffle_data	separates data into batches of size batch_size	Pairs were divided into batch of batch_size
pad_batch	batch	pads batches to allow for sentences of variable lengths to be computed in parallel	Variable lengths of sentences in a batch were padded into the same length of size = len(longest sentence).
calculate_wer	predicted_word = "मलाई मन पर्ने नेपाल " reference_words="मलाई नेपाल मन पर्छ”	WER for the sentence:0.75	WER for the sentence: 0.75
save_checkpoint	epoch, encoder, decoder, encoder_optimizer, decoder_optimizer,path	Save checkpoint at certain epoch during training	The checkpoints were saved at every interval of stated epochs

load_checkpoint	checkpoint = torch.load(path, map_location=device)	(if model was saved til epoch 30) Resuming from epoch 30	Resuming from epoch 30
evaluate_randomly	encoder, decoder, pairs, trim	(random evaluation in test set) उहाँले सचेत गर्नुपर्छ भन्ने सचेत गर्नुपर्छ	उहाँले सचेत गर्नुपर्छ भन्ने सचेत गर्नुपर्छ

5.2.2. Test Cases for System Testing

System testing is a testing that validates the complete and full integrated system product. Its purpose is to evaluate the end to end system specification. System testing results are given:

Table 5.2: Test Cases for System Testing of English to Nepali Translation Model

Input	Expected translation	Actual Translation	WER
i can do much better than this	म यो भन्दा धेरै राम्रो गर्न सक्छु	म यो भन्दा धेरै राम्रो गर्न सक्छु	0.0
i love you	म तिमीलाई माया गर्छु	म तिमीलाई माया गर्छु	0.0

Please e-mail us your suggestions and bugs	कृपया हामीलाई आफ्नो सुझाव र बगहरू इमेल गर्नुहोस्	कृपया हामीलाई तपाईंको सुझाव र <UNK> हामीलाई ईमेल गर्नुहोस्	0.5
All responsibilities are of election officer.	सबै जिम्मेवारी निर्वाचन अधिकृतको हो	सबै जिम्मेवारी निर्वाचन अधिकारी हुन्	0.4
what are you doing here	तपाईं यहाँ के गर्दै हुनुहुन्छ	यहाँ के गर्दै हुनुहुन्छ यहाँ के गर्दै हुनुहुन्छ	0.8
This means that vegetarian diets are low in iron.	यसको मतलब शाकाहारी भोजनमा आइरन कम हुन्छ	यसको अर्थ <UNK> <UNK> <UNK> कम हुन्छ भन्ने अर्थ हुन्छ	1.0
Click here to view and download PDF	PDF हेर्न र डाउनलोड गर्न यहाँ क्लिक गर्नुहोस्	क्लिक गर्नुहोस् र डाउनलोड <UNK> लागि यहाँ क्लिक गर्नुहोस्	0.5

5.3. Result Analysis

We compared various hyperparameters during experimentation and found the following configuration to be the most effective for achieving optimal results in training our Neural Machine Translation (NMT) model for English-to-Nepali translation. The model employs a 3-layer architecture for both the encoder and decoder, which effectively balances depth and computational efficiency. Each layer has a hidden size of 728, ensuring sufficient capacity to capture complex linguistic patterns in the data. To mitigate overfitting, a dropout rate of 0.1 is applied across both the encoder and decoder layers.

The model was trained with a batch size of 64 for both the training and test sets, which offered a balance between computational performance and gradient stability. Training spanned 30

epochs, with an initial learning rate of 1, enabling the model to make significant updates in the early stages. A learning rate schedule was implemented, progressively reducing the learning rate by a factor of 100 at the 10th, 20th, and 25th epochs to ensure finer weight adjustments in later stages of training. For loss computation, we used the CrossEntropyLoss criterion, a standard choice for sequence-to-sequence tasks, which penalizes incorrect token predictions effectively. This configuration provided stable convergence and minimized overfitting, enabling the model to generalize well across the diverse and context-rich Nepali language.

The Nepali-to-English translation model has shown good performance on the training data but faces challenges when translating unseen data.This plot below provides a detailed view of the training and evaluation metrics for our Neural Machine Translation (NMT) model, which employs LSTM cells with a global attention mechanism for translating English to Nepali. The top sub-plot illustrates the training and test loss over time (in minutes), showing how the model's predictions improve during training. Initially, the loss is high due to random weight initialization and the complex nature of the Nepali language. However, as training progresses, both the train and test losses decrease significantly, indicating that the model is learning the underlying patterns of translation. The gap between train and test loss narrows but doesn't completely close, which may reflect some degree of overfitting or the intrinsic challenge of handling the variability in Nepali translations. The second sub-plot presents the Word Error Rate (WER) on the test set, showing a general decline as the model becomes more accurate, though fluctuations indicate sensitivity to complex or rare sentence structures. WER provides a quantitative metric to evaluate translation quality, although its limitations in capturing semantic equivalence are notable.

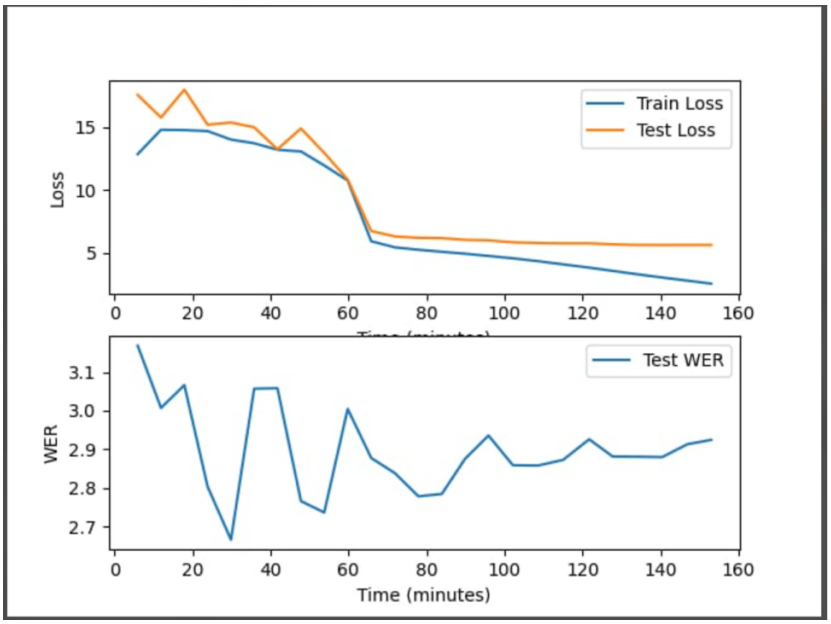


Figure 5.1: Plot of Train and Test loss of English to Nepali Translation Model

It is important to emphasize that Nepali, being a linguistically rich language, often has multiple valid translations for a single word or phrase, depending on tone, context, or regional variations. For example, the English word "come" could be translated as "आउ" (casual), "आउनुहोस्" (formal), or "आउदैछु" (progressive), each suited to different contexts. Automatic evaluation metrics like WER and loss may penalize the model for providing one correct translation while ignoring others, leading to an incomplete assessment of performance. Thus, incorporating human evaluation, where bilingual experts assess translation quality based on context, fluency, and meaning, is essential for a holistic understanding. For instance, while the model might translate "How are you?" as "तपाईंलाई कस्तो छ?", human evaluators might note its appropriateness depending on whether the tone should be formal or informal. Human evaluation complements automated metrics, ensuring a nuanced and realistic evaluation of translation quality in the context of a language as diverse as Nepali.

1. Performance on Seen Data:

- Good Accuracy for Known Sentences: The model works well on sentences it has seen during training. It translates these sentences accurately, meaning it has learned the relationship between Nepali and English sentences effectively.

2. Challenges on Unseen Data:

- **Poor Generalization:** When the model is tested with new sentences (unseen data), its performance drops. This shows that the model is having trouble applying what it learned to sentences it has not seen before. The new sentences may contain words, phrases, or sentence structures that the model hasn't learned, causing poor translations.
- **Out-of-Vocabulary (OOV) Problems**: If the model encounters words that were not in the training data, it struggles to translate them. Even though word embeddings like Word2Vec or FastText help with new words, the model still has trouble if these words weren't handled correctly during training.
- **Struggles with Complex Structures:** The model also has difficulty translating sentences with more complex structures that differ from the simpler patterns seen in the training data.

3. Reasons for Poor Translation on Unseen Data:

- **Limited Data Variety:** The training data may not have enough variety in terms of sentence structure, vocabulary, and topics. This makes it hard for the model to handle new, unseen sentences.
- **Out-of-Vocabulary (OOV) Problems:** Due to small vocabulary size, If the model encounters words that were not in the training data, it struggles to translate them.

CHAPTER 6: CONCLUSION AND FUTURE RECOMMENDATIONS

6.1. Conclusion

In this project, a Nepali-to-English translation system was developed using deep learning, specifically an LSTM-based encoder-decoder model. The model performs well on sentences encountered during training, accurately translating Nepali sentences into English. However, its performance decreases when translating new, unseen sentences.

This drop in performance can be attributed to factors such as overfitting to the training data, limited diversity in the training set, and difficulties in handling out-of-vocabulary (OOV) words and complex sentence structures. The attention mechanism, which is designed to focus on relevant parts of the input sentence, works well for familiar sentences but struggles with unseen sentences, resulting in inaccuracies.

Despite these challenges, the project demonstrates the potential of deep learning models for translation tasks, particularly when trained on large, diverse datasets. The model's success with seen data is promising, but further improvements are needed to enhance its ability to generalize and translate unseen sentences effectively.

6.2. Future Recommendations

We will focus on several strategies to enhance the Nepali-to-English translation model and improve its performance, especially when dealing with unseen data.

- **Fine-Tuning Hyperparameters:** We will fine-tune hyperparameters like learning rate, batch size, and the number of epochs to optimize the model's performance. By experimenting with different values, we aim to find the best configuration that improves the model’s ability to generalize, particularly on unseen data.
- **Expanding the Training Dataset:** We will expand the training dataset to include a broader range of sentence structures, topics, and domains. This will help the model perform better on unseen data and improve its overall versatility.

- **Enhanced Attention Mechanism:** We will work on improving the attention mechanism by integrating advanced attention models, such as multi-head attention. This will allow the model to focus better on the relevant parts of the input sentence, improving translation accuracy, particularly for complex or unfamiliar sentence structures.
- **Data Augmentation:** We will increase the diversity of the training dataset to help the model generalize better. Techniques such as back translation (translating sentences into another language and back) and paraphrasing will be used to generate more varied examples, making the model capable of handling a wider range of sentence structures and vocabulary.

REFERENCES

[1] Y. A. M. C. O. Hanımnur Mercan, "The Evolution of Machine Translation: A Review Study," *Journal of English Education,* 2024.

[2] B. v. M. D. B. B. Y. B. Kyunghyun Cho, "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," 2020.

[3] A. F. Raj Dabre, "Softmax Tempering for Training Neural Machine Translation Models," 2020.

[4] A. S. A. S. R. K. Gaurav Tiwari, "English-Hindi Neural Machine Translation-LSTM Seq2Seq and ConvS2S," *2020 International Conference on Communication and Signal Processing (ICCSP),* 2020.

[5] B. K. B. P. A. Shabdapurush Poudel, "Bidirectional English-Nepali Machine Translation(MT) System for Legal Domain," *ELRA and ICCL,* Vols. Proceedings of the 3rd Annual Meeting of the Special Interest Group on Under-resourced Languages @ LREC-COLING 2024, 2024.

[6] W. Z. H. L. M. W. Dan Guo, "Hierarchical LSTM for Sign Language Translation," *Thirty-Second AAAI Conference on Artificial Intelligence ,* vol. 32, 2020.

[7] B. K. B. Praveen Acharya, "A Comparative Study of SMT and NMT: Case Study of English-Nepali Language Pair," *6th Workshop on Spoken Language Technologies for Under-Resourced Languages (SLTU 2021),* Vols. 29-31, 2021.

[8] S. K. S. N. E. N. L. Angel Shrestha, "A Reflection on Machine Translation Process from Nepali to English," *ICCTSAI 202,* 2021.

[9] K. a. S. S. Nemkul, ""Low resource English to Nepali sentence translation using RNN—long short-term memory with attention."," *Proceedings of International Conference on Sustainable Expert Systems: ICSES 2020.,* 2020.

[10] X. S. C. H. J. Z. Yong Yu, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures," *Neural Computation (2020) ,* 2020.

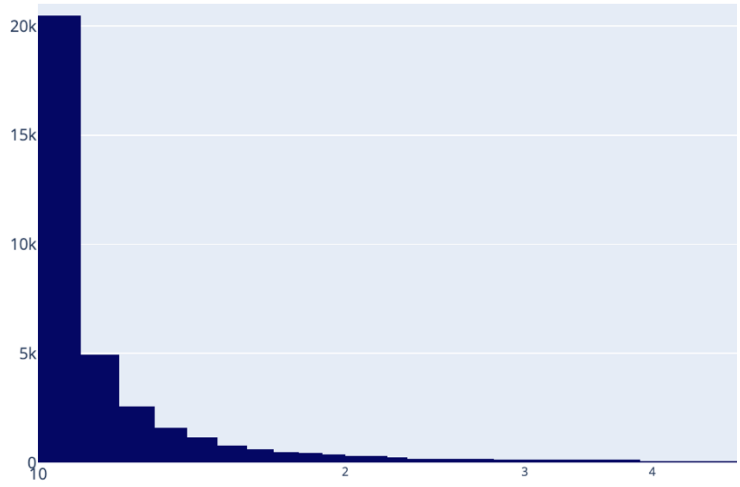
[11] M. M. Y. Z. Anqi Mao, "Cross-Entropy Loss Functions: Theoretical Analysis and Applications," *Proceedings of the 40th International Conference on Machine Learning,* 2023.

[12] Y. Jia, "Attention Mechanism in Machine Translation," *Journal of Physics: Conference Series,* 2020.

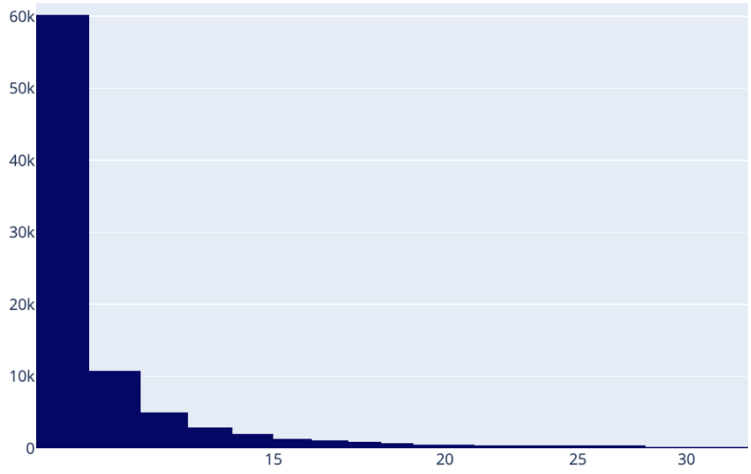
[13] K. a. S. S. Nemkul, ". "Low resource English to Nepali sentence translation using RNN—long short-term memory with attention."," *Proceedings of International Conference on Sustainable Expert Systems: ICSES 2020. Springer Singapore, 2021,* 2020.

APPENDICES

English word Frequency Counts



Nepali word Frequency Counts



```
"""HYPERPARAMETERS"""

"""number of layers in both the Encoder and Decoder"""
layers = 3

"""Hidden size of the Encoder and Decoder"""
hidden_size = 728

"""Dropout value for Encoder and Decoder"""
dropout = 0.1

"""Training set batch size"""
batch_size = 64

"""Test set batch size"""
test_batch_size = 64

"""number of epochs (full passes through the training data)"""
epochs = 30

"""Initial learning rate"""
learning_rate= 1

"""Learning rate schedule. Signifies by what factor to divide the learning rate
at a certain epoch. For example {5:10} would divide the learning rate by 10
before the 5th epoch and {5:10, 10:100} would divide the learning rate by 10
before the 5th epoch and then again by 100 before the 10th epoch"""
lr_schedule = {10:100, 20:100,25:100}

"""loss criterion, see https://pytorch.org/docs/stable/nn.html for other options"""
criterion = nn.CrossEntropyLoss()
```

Hyperparameters of model during training

```
EncoderRNNManual(
  (embedder): Embedding(38604, 728)
  (dropout): Dropout(p=0.1, inplace=False)
  (lstm_cell): LSTM(728, 728, num_layers=3, dropout=0.1)
  (fc): Linear(in_features=728, out_features=728, bias=True)
)
```

Encoder Evaluation

```
DecoderAttnManual(
  (embedder): Embedding(31228, 728)
  (dropout_layer): Dropout(p=0.1, inplace=False)
  (score_learner): Linear(in_features=728, out_features=728, bias=True)
  (lstm_cell): LSTM(728, 728, num_layers=3, dropout=0.1)
  (context_combiner): Linear(in_features=1456, out_features=728, bias=True)
  (tanh): Tanh()
  (output): Linear(in_features=728, out_features=31228, bias=True)
  (soft): Softmax(dim=1)
  (log_soft): LogSoftmax(dim=1)
)
```

Decoder Evaluation

```python
import torch
from torch import nn
from torch.autograd import Variable

class EncoderRNNManual(nn.Module):
    def __init__(self, input_size, hidden_size, bidirectional, layers, dropout):
        super(EncoderRNNManual, self).__init__()

        # Set directions for bidirectionality
        self.directions = 2 if bidirectional else 1
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = layers
        self.dropout_rate = dropout

        # Initialize embedding layer and dropout
        self.embedder = nn.Embedding(input_size, hidden_size)
        self.dropout = nn.Dropout(dropout)

        # Replace LSTM with custom LSTMCell
        self.lstm_cell = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size,
            num_layers=layers,
            dropout=dropout,
            bidirectional=bidirectional,
            batch_first=False
        )
        self.fc = nn.Linear(hidden_size * self.directions, hidden_size)

    # Actual forward code
    def forward(self, input_data, h_hidden, c_hidden):
        embedded_data = self.embedder(input_data)
        embedded_data = self.dropout(embedded_data)
        hiddens, outputs = self.lstm_cell(embedded_data, (h_hidden, c_hidden))
```

```python
def create_init_hiddens(self, batch_size):
    # Create initial hidden and cell states for the encoder
    h_hidden = Variable(torch.zeros(self.num_layers * self.directions, batch_size, self.hidden_size))
    c_hidden = Variable(torch.zeros(self.num_layers * self.directions, batch_size, self.hidden_size))

    if torch.cuda.is_available():
        return h_hidden.cuda(), c_hidden.cuda()
    else:
        return h_hidden, c_hidden
```

Encoder_RNN

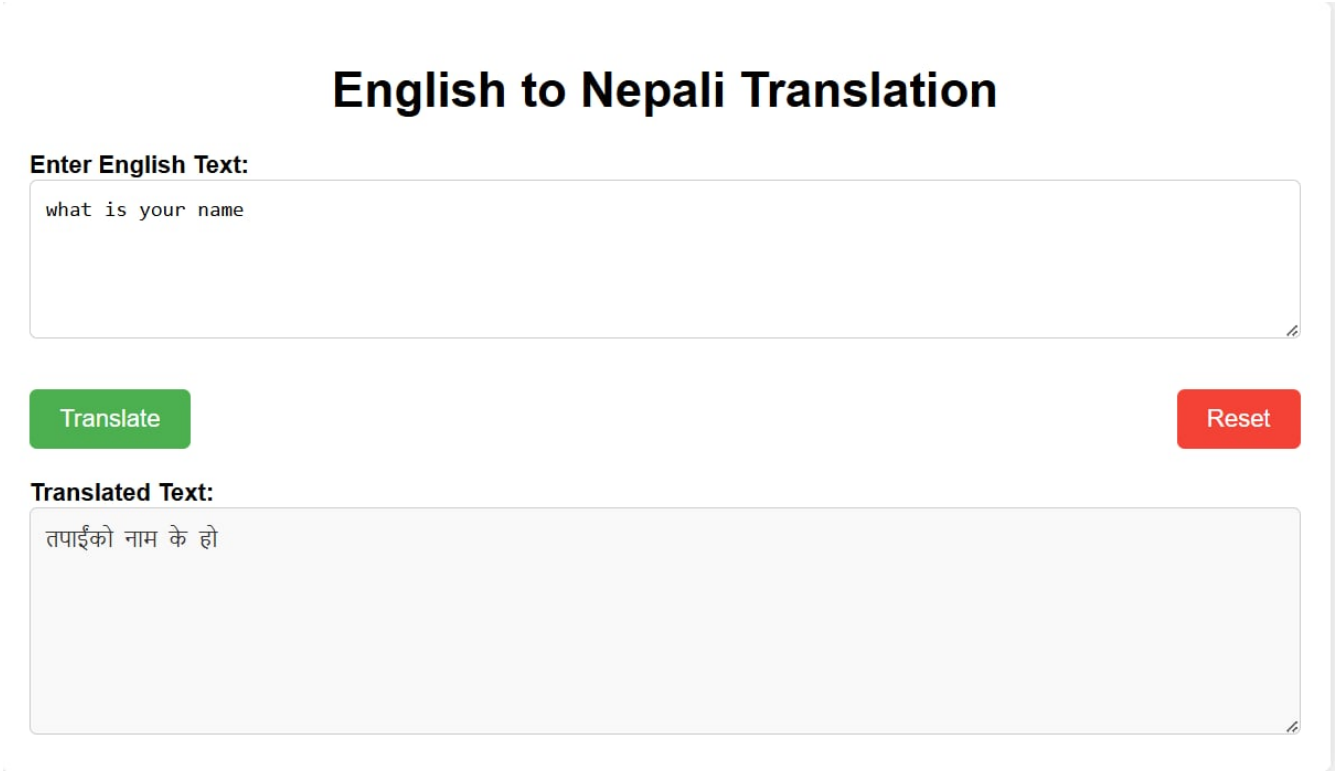
```python
class DecoderAttnManual(nn.Module):
    def __init__(self, hidden_size, output_size, layers, dropout, bidirectional):
        super(DecoderAttnManual, self).__init__()

        # Attributes and embeddings initialization
        self.directions = 2 if bidirectional else 1
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.num_layers = layers
        self.dropout = dropout
        self.embedder = nn.Embedding(output_size, hidden_size)
        self.dropout_layer = nn.Dropout(dropout)
        self.score_learner = nn.Linear(hidden_size * self.directions, hidden_size * self.directions)
        self.lstm_cell = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size,
            num_layers=layers,
            dropout=dropout,
            bidirectional=bidirectional,
            batch_first=False
        )

        # Additional layers
        self.context_combiner = nn.Linear((hidden_size * self.directions) + (hidden_size * self.directions), hidden_size)
        self.tanh = nn.Tanh()
        self.output = nn.Linear(hidden_size, output_size)
        self.soft = nn.Softmax(dim=1)
        self.log_soft = nn.LogSoftmax(dim=1)

    def forward(self, input_data, h_hidden, c_hidden, encoder_hiddens):
    # Embedding the input token
        embedded_data = self.embedder(input_data)
        embedded_data = self.dropout_layer(embedded_data)
        batch_size = embedded_data.shape[1]
```

```python
    outputs, (hiddens, c_hiddens) = self.lstm_cell(embedded_data, (h_hidden, c_hidden))

# Compute attention scores
    prep_scores = self.score_learner(encoder_hiddens.permute(1, 0, 2))
    scores = torch.bmm(prep_scores, outputs.permute(1, 2, 0))
    attn_scores = self.soft(scores)

# Compute context matrix and combined hidden state
    con_mat = torch.bmm(encoder_hiddens.permute(1, 2, 0), attn_scores)
    h_tilde = self.tanh(
        self.context_combiner(torch.cat((con_mat.permute(0, 2, 1), outputs.permute(1, 0, 2)), dim=2))
    )

    # Final prediction (shape: [batch_size, 1, vocab_size])
    pred = self.output(h_tilde)

    # Squeeze to remove the unnecessary dimension (shape: [batch_size, vocab_size])
    pred = pred.squeeze(1)

    # Log softmax for prediction
    pred = self.log_soft(pred)

    return pred, (hiddens, c_hiddens)
```

Decoder_RNN

```
train_and_test(epochs, test_eval_every, plot_every,
               learning_rate, lr_schedule, train_pairs,
               test_pairs, input_lang, output_lang,
               batch_size, test_batch_size, encoder,
               decoder,trim, save_interval, path)
```

Train and Test



UI for Translation WebApp