Capstone Project: English to Hindi Language Translator Web App using TensorFlow and Streamlit

This capstone project details the development of an English to Hindi language translator using a sequence-to-sequence (Seq2Seq) deep learning model with Long Short-Term Memory (LSTM) units.

Project Overview

This capstone project details the development of an English to Hindi language translator using a sequence-to-sequence (Seq2Seq) deep learning model with Long Short-Term Memory (LSTM) units. The project encompasses the entire lifecycle, from data collection and meticulous preprocessing to building, training, and saving the LSTM-based encoder-decoder model. Finally, the report outlines the deployment readiness by saving the trained models and tokenizers for integration into a production-ready web application using Streamlit.

Project Goals

Data Acquisition and Preparation

Obtain a parallel corpus for English to Hindi translation and preprocess it by tokenizing, normalizing, and padding sequences.

Seq2Seq Model Construction

Build an encoder-decoder architecture from scratch using TensorFlow's Keras API, leveraging LSTM layers.

Model Training

Train the Seq2Seq model on the prepared dataset, monitoring performance.

Model Persistence

Save the trained encoder and decoder models, along with the tokenizers, for later use in prediction and deployment.

Prediction Functionality

Implement a function to perform real-time translation inference using the trained encoder and decoder models.

Deployment Readiness

Prepare the necessary model artifacts for integration into a Streamlit web application (though the Streamlit UI code itself is external to this report, the readiness is covered).

Project Workflow (High-Level Overview)

This project follows a systematic deep learning workflow, encompassing several key stages to build a functional language translation system:

Data Acquisition

Downloading a parallel English-Hindi dataset from a public source.

Data Preprocessing

- Extraction: Separating raw language pairs into distinct English (source) and Hindi (target) sentences.
- Tokenization: Converting text sentences into numerical sequences, creating vocabularies for both languages.
- Padding: Standardizing sentence lengths by adding padding tokens to create uniform input for the neural network.
- Target Output Preparation: Shifting and one-hot encoding the decoder's target sentences for model training.

Model Architecture Design

- Encoder: Building an LSTM-based encoder to convert English input sentences into a fixed-size context vector (sentence embedding).
- Decoder: Constructing an LSTM-based decoder, initialized with the encoder's context, to generate Hindi translations word by word.

Model Training

- Compiling the complete Seq2Seq model.
- Training the model on the preprocessed dataset, using "teacher forcing" for the decoder, to learn the English-to-Hindi mapping.

Model Persistence

- Saving the trained encoder and decoder models separately for efficient inference.
- Serializing the English and Hindi tokenizers to ensure consistent text-to-sequence conversion during prediction.

Translation Inference

- Developing a prediction function that utilizes the saved encoder to get the sentence context.
- Using the saved decoder in an autoregressive loop to generate the Hindi translation word by word.

Deployment Readiness

- Organizing the project directory for clarity and ease of integration.
- Outlining the structure for a Streamlit user interface that will load the saved models and tokenizers to perform live translations.

This sequential approach ensures that each component of the translation system is robustly built and integrated for an end-to-end solution.

Technologies Used and System Requirements

Technologies Used

- Programming Language: Python
- Deep Learning Framework: TensorFlow 2.x
- Data Handling: NumPy, zipfile, io, pickle
- Web Framework (for deployment readiness): Streamlit (implied for UI)

System Requirements

- Python environment (e.g., managed by Conda).
- TensorFlow installed with Keras.
- Standard Python libraries: NumPy, zipfile, io, pickle.
- Internet connection for data download.

Environment Setup

To begin, set up a dedicated Conda environment to manage project dependencies effectively (though the code primarily shows wget and unzip which are shell commands executed from a Python environment). The initial steps involve downloading and unzipping the dataset:

```
# Install TensorFlow if not already present
# pip install tensorflow
# Download the dataset using wget
!wget http://www.manythings.org/anki/hin-eng.zip --quiet
# Unzip the downloaded file
!unzip hin-eng.zip
```

Data Collection and Preprocessing

The project starts by acquiring a parallel corpus for English to Hindi translation from the **manythings.org Anki collection**. The hin-eng.zip file contains the raw text data.

1. Data Loading and Initial Inspection

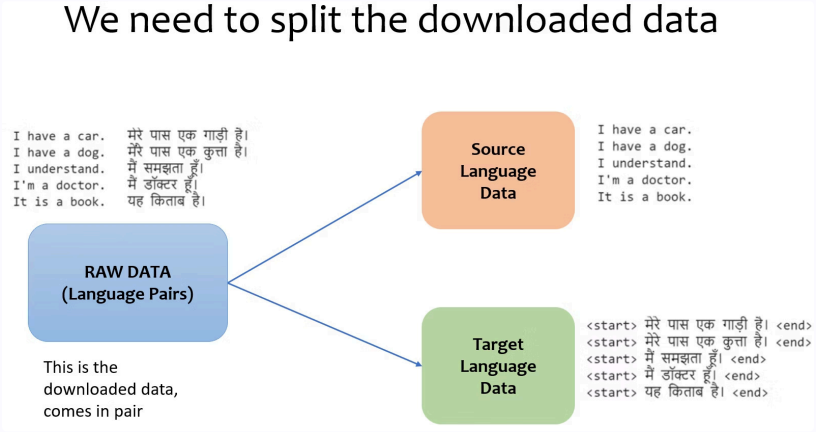
The hin-eng.zip file is read, and the hin.txt file within it is extracted. The content is then read line by line.

```
import zipfile
import io

# Read the zip file
zf = zipfile.ZipFile('hin-eng.zip', 'r')

# Extract data from zip file
data = ''
with zf.open('hin.txt') as readfile:
    for line in io.TextIOWrapper(readfile, 'utf-8'):
        data += line

# The 'data' variable now holds the entire text content.
# The length of the data can be inspected (e.g., len(data)).
```

2. Extracting Source and Target Language Pairs

The raw data, which contains English sentences, Hindi translations, and an auxiliary field (ignored), is split into individual lines. Each line is then parsed to separate the English (source) and Hindi (target) sentences. A crucial step here is the addition of <start> and <end> tokens to the target Hindi sentences. These tokens are vital for the decoder during both training and prediction, signaling the beginning and end of a translation.

```
data = data.split('\n')
encoder_text = [] # Initialize Source language list (English)
decoder_text = [] # Initialize Target language list (Hindi)

# Iterate over data to split into source and target pairs
for line in data:
    try:
        in_txt, out_txt, _ = line.split('\t')
        encoder_text.append(in_txt)
        # Add '' as start sequence and '' as end sequence in target
        decoder_text.append('<start>' + out_txt + '<end>')
    except:
        pass # Ignore lines that do not conform to the expected format
```



Extracting Source and Target Language Pairs

We split the raw data into English (source) and Hindi (target) pairs from each line, discarding the auxiliary field. Two separate lists store these languages. We add <start> and <end> tokens to Hindi sentences, which are essential for marking translation boundaries during both training and inference.

Tokenization and Sequence Preparation

3. Tokenization of Source Language Sentences (English)

A tf.keras.preprocessing.text.Tokenizer is used to convert English sentences into sequences of integers. This involves:

- Fitting the tokenizer on the encoder_text to build a vocabulary of English words and assign a unique integer ID to each word.
- Converting each English sentence into its corresponding sequence of integer IDs.
- Determining the maximum sentence length in the English corpus and the total vocabulary size.

```
# Tokenizer for source language (English)
encoder_t = tf.keras.preprocessing.text.Tokenizer(lower=True)
encoder_t.fit_on_texts(encoder_text) # Fit on Source sentences

# Convert English text to integer sequences
encoder_seq = encoder_t.texts_to_sequences(encoder_text)

# Calculate maximum sentence length for source language
max_encoder_seq_length = max((len(txt) for txt in encoder_seq))

# Calculate source language vocabulary size (+1 for potential 0-padding or OOV token)
encoder_vocab_size = len(encoder_t.word_index)
```

4. Tokenization of Target Language Sentences (Hindi)

Similarly, a tokenizer is created for the Hindi sentences (decoder_text). Crucially, the filters for this tokenizer are explicitly defined to *not* remove < and > characters, as these are part of our custom <start> and <end> tokens.

```
# Tokenizer for target language (Hindi)
# Filters are set to preserve '<' and '>' which are part of our  and  tokens
decoder_t = tf.keras.preprocessing.text.Tokenizer(filters='!"#$%&()*+,-./:;=?@[\\\]^_`{|}~\t\n')
decoder_t.fit_on_texts(decoder_text) # Fit on Target sentences
decoder_seq = decoder_t.texts_to_sequences(decoder_text) # Convert sentences to numbers

# Calculate maximum sentence length for target language
max_decoder_seq_length = max((len(txt) for txt in decoder_seq))

# Calculate target language vocabulary size (+1 for potential 0-padding or OOV token)
decoder_vocab_size = len(decoder_t.word_index)
```

Sequence Padding and Target Preparation

5. Padding Sentences

Neural networks typically require fixed-size inputs. Therefore, all English and Hindi sentence sequences are padded to a uniform length, which corresponds to the maximum sentence length found in each language. pad_sequences is used, with 'pre' padding for encoder inputs and 'post' padding for decoder inputs.

```
# Source sentences padding (pre-padding)
encoder_input_data = tf.keras.preprocessing.sequence.pad_sequences(encoder_seq,
maxlen=max_encoder_seq_length, padding='pre')

# Target sentences padding (post-padding)
decoder_input_data = tf.keras.preprocessing.sequence.pad_sequences(decoder_seq,
maxlen=max_decoder_seq_length, padding='post')
```

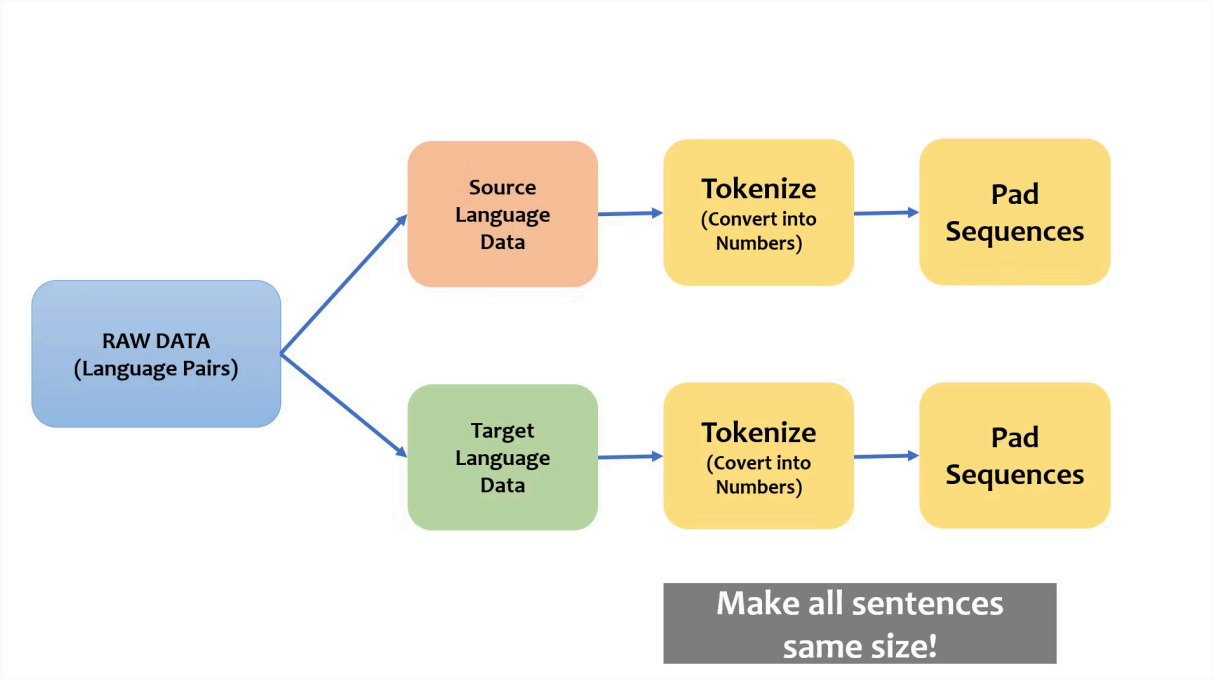
6. Preparing Decoder Output for Training

For training, the decoder's expected output needs to be prepared. This involves creating a target sequence that is essentially the decoder_input_data shifted by one time step to the left, effectively dropping the <start> token and aligning each word with the word it should predict. Additionally, this target data is then **one-hot encoded** to match the output shape of the decoder's softmax layer, which predicts probabilities over the entire vocabulary.

```
import numpy as np

# Initialize array for shifted decoder target data
decoder_target_data = np.zeros((decoder_input_data.shape[0], decoder_input_data.shape[1]))

# Shift Target output by one word (dropping )
for i in range(decoder_input_data.shape[0]):
    for j in range(1,decoder_input_data.shape[1]):
        decoder_target_data[i][j-1] = decoder_input_data[i][j]

# Initialize one-hot encoding array for decoder target
decoder_target_one_hot = np.zeros(((decoder_input_data.shape[0], decoder_input_data.shape[1],
len(decoder_t.word_index)+1)))

# Build one-hot encoded array
for i in range(decoder_target_data.shape[0]):
    for j in range(decoder_target_data.shape[1]):
        decoder_target_one_hot[i][j] = tf.keras.utils.to_categorical(decoder_target_data[i][j],
num_classes=len(decoder_t.word_index)+1)
```

An integer-to-word dictionary is also created for the decoder vocabulary, which will be essential for converting predicted word IDs back to human-readable Hindi.

```
# Create mapping from integer IDs to words for the decoder vocabulary
int_to_word_decoder = dict((i,c) for c, i in decoder_t.word_index.items())
```



Sequence Padding and Target Preparation

Model Architecture: Sequence-to-Sequence (Seq2Seq) with LSTM

The Seq2Seq model consists of an Encoder and a Decoder. The tf.keras.backend.clear_session() call ensures a clean slate before building the model.

1. Define Configuration Parameters

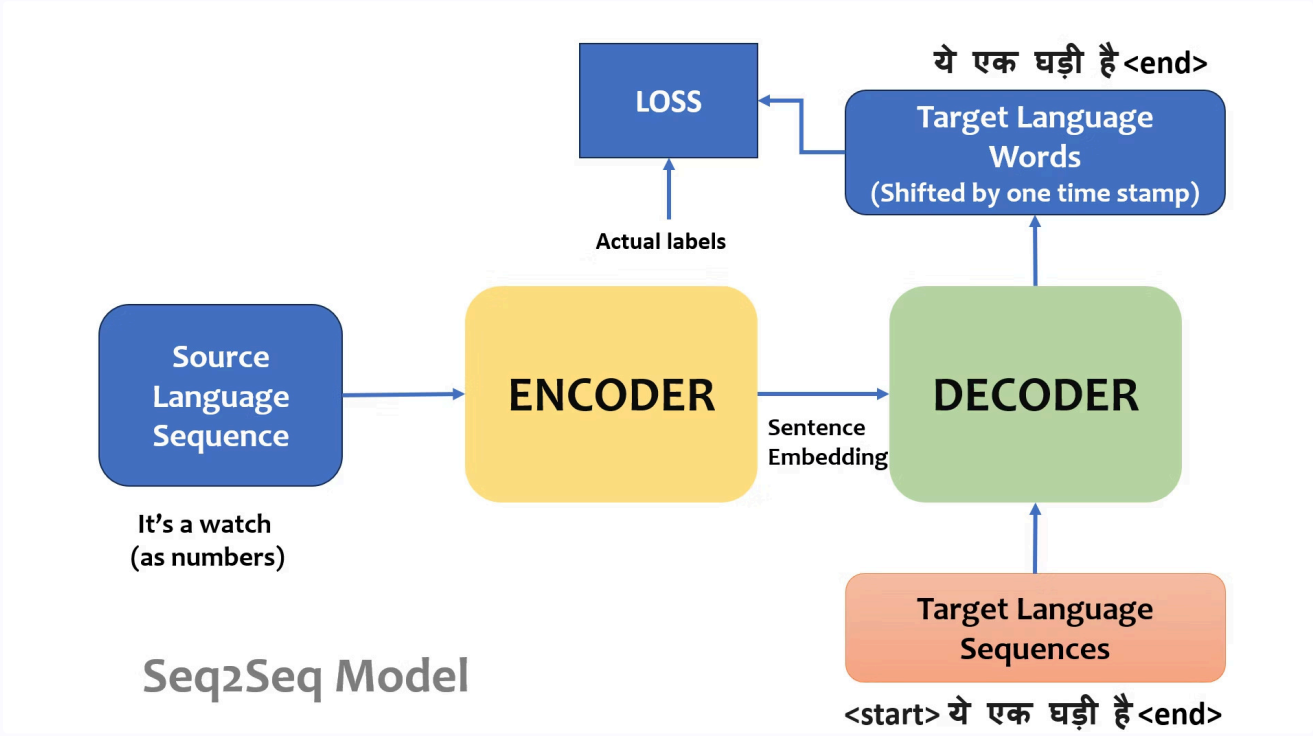
Key architectural parameters for the model are defined, including embedding sizes and the number of LSTM units (memory size).

```
encoder_embedding_size = 80 # Dimension for English word embeddings
decoder_embedding_size = 80 # Dimension for Hindi word embeddings
rnn_units = 100 # Number of LSTM units (hidden state size) for both encoder and decoder
```

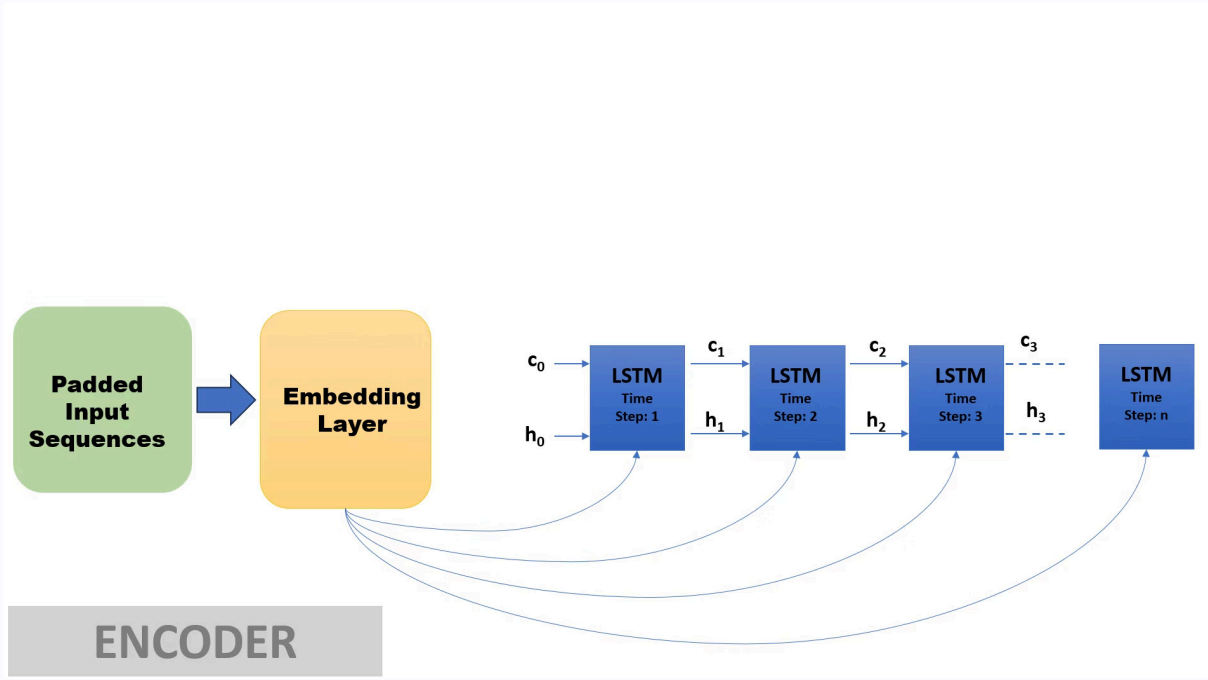
2. Build Encoder Model (Training)

The encoder takes the padded English sentence sequences as input. An Embedding layer converts word IDs into dense vector representations. An LSTM layer processes these embeddings. The return_state=True argument ensures that the final hidden state (state_h) and cell state (state_c) of the LSTM are returned. These states represent the "sentence embedding" or context vector, capturing the essence of the input English sentence, and are crucial for initializing the decoder. Dropout is applied for regularization.

```
tf.keras.backend.clear_session()

# Encoder Input Layer: Shape corresponds to max English sentence length
encoder_inputs = tf.keras.layers.Input(shape=(max_encoder_seq_length,))

# Encoder Embedding Layer: Converts input integer IDs to dense vectors
encoder_embedding = tf.keras.layers.Embedding(encoder_vocab_size + 1, encoder_embedding_size)
encoder_embedding_output = encoder_embedding(encoder_inputs)

# Encoder LSTM Layer: Processes embedded sequence and returns final states
# - return_state=True: to get the final hidden and cell states
# - dropout and recurrent_dropout for regularization
x, state_h, state_c = tf.keras.layers.LSTM(rnn_units, return_state=True, dropout=0.2, recurrent_dropout=0.3)
(encoder_embedding_output)

# Encoder states (context vector) to be passed to the decoder
encoder_states = [state_h, state_c]
```



Model Architecture



Encoder Model

Decoder Model and Training Process

3. Build Decoder Model (Training)

The decoder receives the padded Hindi input sequences (with <start> token) and is initialized with the encoder's final states. It also uses an Embedding layer. The LSTM layer here has return_sequences=True because it needs to output a prediction for *each time step* (word) in the target sequence. A Dense layer with softmax activation at the end outputs a probability distribution over the entire Hindi vocabulary for each predicted word.

```
# Decoder Input Layer: Shape corresponds to max Hindi sentence length
decoder_inputs = tf.keras.layers.Input(shape=(max_decoder_seq_length,))

# Decoder Embedding Layer
decoder_embedding = tf.keras.layers.Embedding(decoder_vocab_size + 1, decoder_embedding_size)
decoder_embedding_output = decoder_embedding(decoder_inputs)

# Decoder LSTM Layer: Initialized with encoder states, returns sequences for output
# - return_sequences=True: to get output at each time step for prediction
decoder_rnn = tf.keras.layers.LSTM(rnn_units, dropout=0.2, recurrent_dropout=0.3, return_sequences=True,
return_state=True)

# Decoder RNN output, initialized with encoder's final states
all_hidden_states_d, last_hidden_state_d, last_cell_state_d = decoder_rnn(
    decoder_embedding_output, initial_state=encoder_states)

# Decoder Dense Output Layer: Predicts probabilities over the target vocabulary
decoder_dense = tf.keras.layers.Dense(decoder_vocab_size + 1, activation='softmax')
decoder_outputs = decoder_dense(all_hidden_states_d)
```

4. Build Full Seq2Seq Training Model

The complete training model takes both the encoder inputs and decoder inputs as inputs, and outputs the decoder_outputs (the predicted probability distributions for each word in the target sequence). The model is compiled with the Adam optimizer and categorical_crossentropy loss, suitable for multi-class classification (predicting the next word).

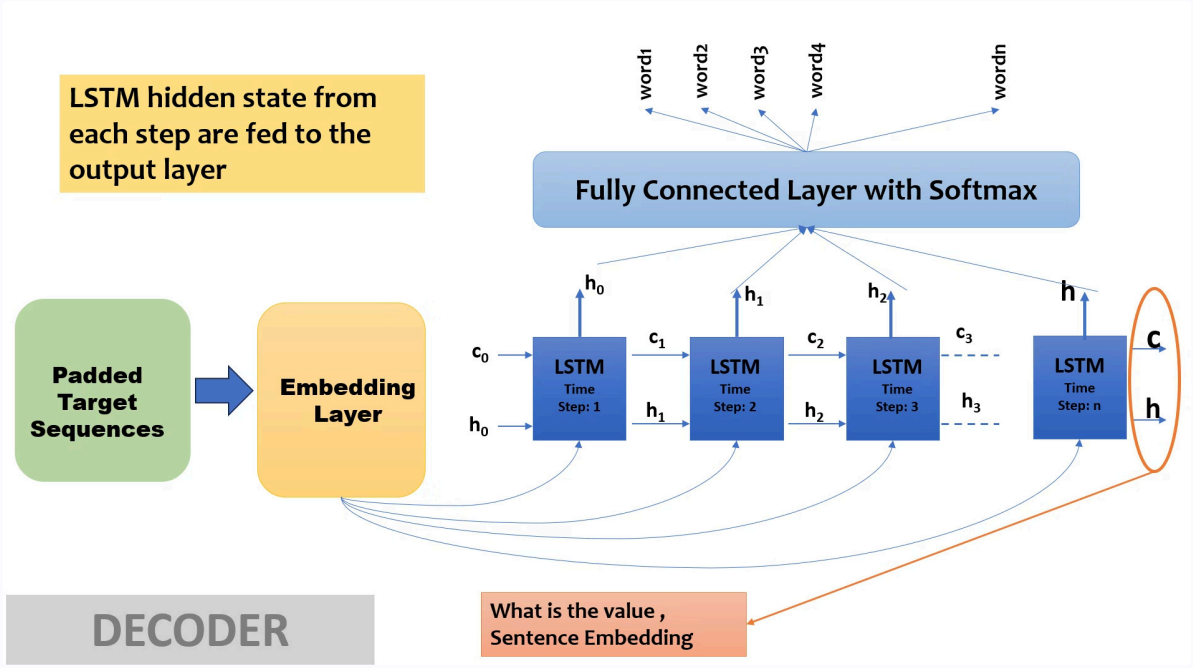
```
# Build a Seq2Seq model -> Encoder + Decoder
model = tf.keras.models.Model([encoder_inputs, decoder_inputs], decoder_outputs)

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

# Display model summary
model.summary()
```

Training and Fine-tuning

The model is trained using the fit method with the prepared encoder_input_data and decoder_input_data as features, and decoder_target_one_hot as the labels. A validation split is used to monitor performance on unseen data during training, helping to prevent overfitting.

```
# Train the model
model.fit([encoder_input_data, decoder_input_data], decoder_target_one_hot, batch_size=64, epochs=10,
validation_split=0.2)
```

During training, the decoder operates using **teacher forcing**, meaning it receives the true target word at each time step (shifted by one position). This allows the model to learn the mapping efficiently.



Decoder Model

Prediction (Inference) Process

Translating a new English sentence into Hindi with the trained Seq2Seq model follows a multi-step inference process that differs significantly from the training phase. Unlike training, where the decoder is "taught" using the actual target sequence (teacher forcing), during prediction, the decoder must generate the output sequence entirely on its own, word by word.

1. Build Encoder Model for Inference

This model takes the English input sequence and outputs only the final hidden and cell states of the encoder's LSTM. These states encapsulate the context of the input sentence.

```
# Build the Encoder Model to predict Encoder States
encoder_model = tf.keras.models.Model(encoder_inputs, encoder_states)
encoder_model.summary()
```

2. Build Decoder Model for Inference

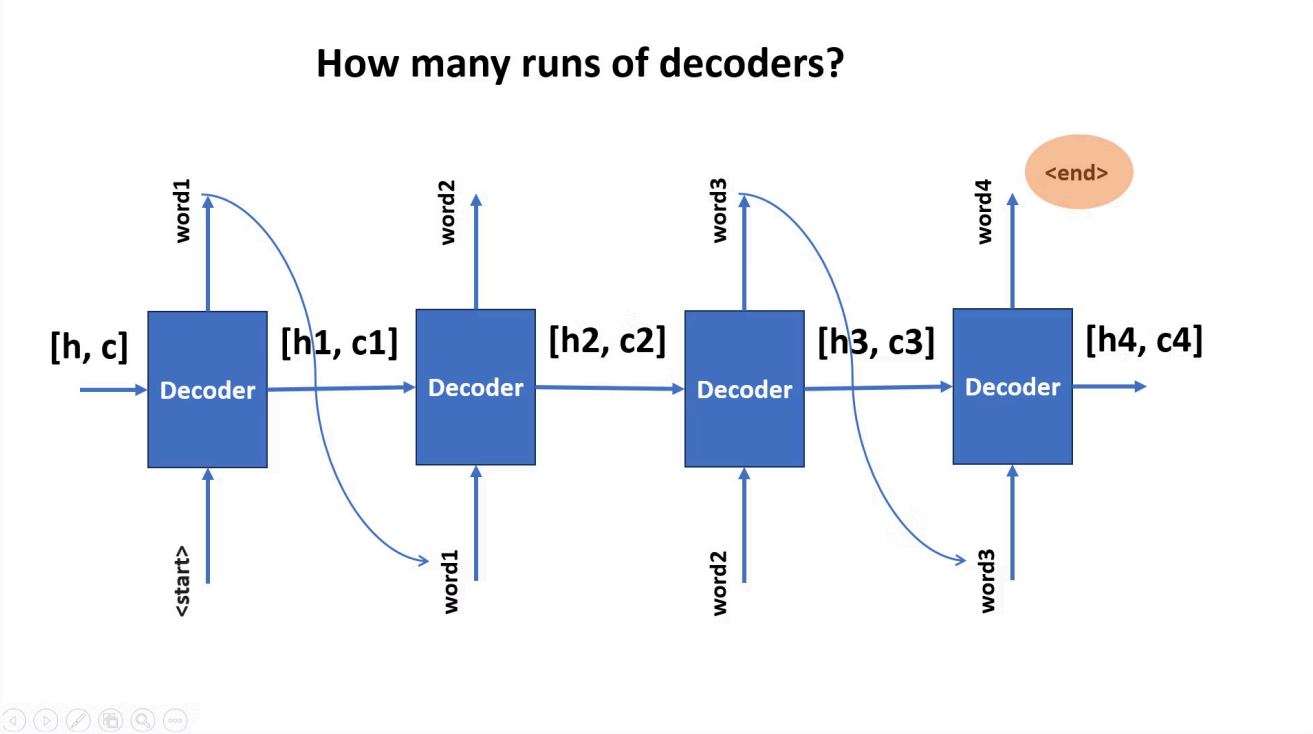
The inference decoder model is designed to perform one-step prediction. It takes:

- An input representing the previously predicted word (or <start> token initially).
- The initial hidden and cell states (from the encoder's output for the first step, or from its own previous step's output for subsequent steps).

It outputs:

- The probability distribution over the vocabulary for the next word.
- Its own updated hidden and cell states for the next iteration.

```
# Define Input for both 'h' state and 'c' state initialization for the decoder
decoder_state_input_h = tf.keras.layers.Input(shape=(rnn_units,))
decoder_state_input_c = tf.keras.layers.Input(shape=(rnn_units,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

# Get Embedding layer output for the single input word
x = decoder_embedding(decoder_inputs)
# Note: decoder_inputs here refers to the input layer defined previously

# Use the trained decoder_rnn layer to get outputs and updated states
# Initial states are from decoder_states_inputs
rnn_outputs, state_h, state_c = decoder_rnn(x, initial_state=decoder_states_inputs)

# The updated decoder states for the next iteration
decoder_states = [state_h, state_c]

# Get Decoder Dense layer output for the predicted word
decoder_outputs = decoder_dense(rnn_outputs)

# Build Decoder Model for inference
decoder_model = tf.keras.models.Model(
    [decoder_inputs] + decoder_states_inputs,  # Inputs: current word + previous states
    [decoder_outputs] + decoder_states)  # Outputs: predicted probabilities + new states
decoder_model.summary()
```



Decoder Model for Predictions

Prediction Function and Deployment

3. Prediction Function (decode_sentence)

This function orchestrates the word-by-word prediction process:

1. **Encoder Prediction:** The encoder_model is used to get the initial decoder_initial_states_value (context vector) from the English input sentence.
2. **Initial Decoder Input:** A target_seq is created, initialized with the <start> token ID.
3. **Autoregressive Loop:** The while loop continues until an <end> token is predicted or a maximum length is reached.

```
def decode_sentence(input_sequence):
    # Get the encoder state values (sentence embedding)
    decoder_initial_states_value = encoder_model.predict(input_sequence)

    # Build a sequence with '' - starting sequence for Decoder
    target_seq = np.zeros(((1,1)))
    target_seq[0][0] = decoder_t.word_index['']

    stop_loop = False
    predicted_sentence = ''
    num_of_predictions = 0

    # Start the autoregressive decoding loop
    while not stop_loop:
        # Predict the next word and get new states
        predicted_outputs, h, c = decoder_model.predict([target_seq] + decoder_initial_states_value)

        # Get the predicted word index with highest probability
        predicted_output = np.argmax(predicted_outputs[0, -1,:])

        # Get the predicted word from its index
        predicted_word = int_to_word_decoder[predicted_output]

        # Check for termination conditions
        if(predicted_word == '' or num_of_predictions > max_decoder_seq_length):
            stop_loop = True
            continue

        num_of_predictions += 1

        # Update predicted sentence
        if (len(predicted_sentence) == 0):
            predicted_sentence = predicted_word
        else:
            predicted_sentence = predicted_sentence + ' ' + predicted_word

        # Update target_seq to be the predicted word index for the next step
        target_seq[0][0] = predicted_output

        # Update initial states value for decoder with the new states
        decoder_initial_states_value = [h,c]

    return predicted_sentence
```

Model and Tokenizer Persistence

To enable deployment of the trained model without retraining, both the encoder and decoder models are saved in H5 format, and the tokenizers are serialized using pickle. These saved artifacts are crucial for the Streamlit application to load and perform translations.

```
# Compile models to avoid potential errors during saving (though often not strictly necessary for H5 saving)
encoder_model.compile(optimizer='adam',loss='categorical_crossentropy')
decoder_model.compile(optimizer='adam',loss='categorical_crossentropy')

# Save the models
encoder_model.save('seq2seq_encoder_eng_hin.h5')  # Encoder model
decoder_model.save('seq2seq_decoder_eng_hin.h5')  # Decoder model

import pickle
# Save the tokenizers
pickle.dump(encoder_t, open('encoder_tokenizer_eng', 'wb'))
pickle.dump(decoder_t, open('decoder_tokenizer_hin', 'wb'))
```

Deployment and Future Enhancements

Directory Structure

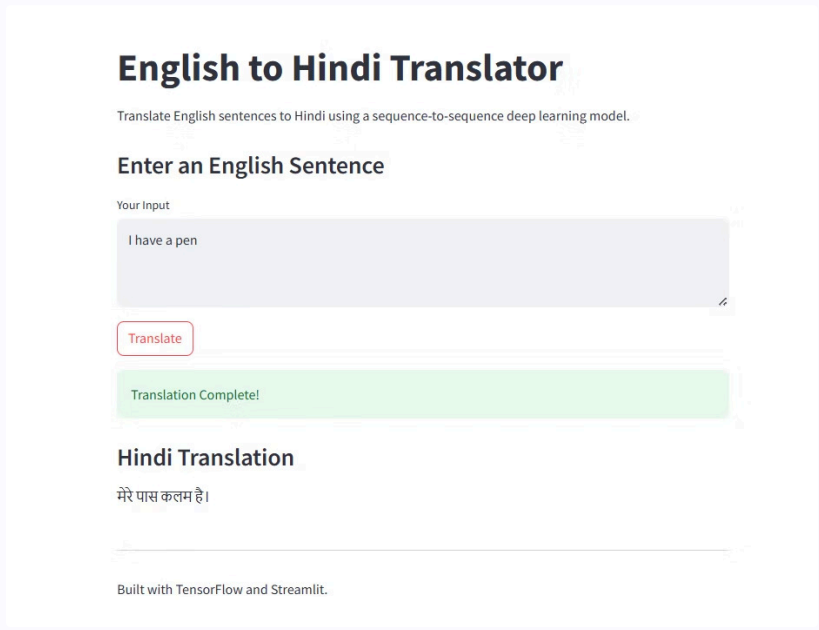
A well-organized directory structure is vital for project maintainability and scalability.

```
language_translator_app/
├── app.py # Streamlit web application script
├── data/ # Directory for datasets
│   ├── hin-eng.zip # Original dataset
│   └── (other processed data files)
├── models/ # Directory for model weights
│   ├── seq2seq_encoder_eng_hin.h5
│   ├── seq2seq_decoder_eng_hin.h5
│   ├── encoder_tokenizer_eng
│   └── decoder_tokenizer_hin
├── requirements.txt # Python dependencies
├── README.md # Project description
└── .gitignore # Files to ignore
```

Streamlit UI Components

The Streamlit app.py script would leverage the saved models and tokenizers to create a user-friendly interface for translation.

- Title and Application Description
- Text Input Box for English sentences
- Translation Trigger Button
- Output Display for Hindi translation
- Styled Layout with columns and containers
- Loading Indicators during translation
- Error Handling for invalid inputs



Deployment Suggestions

Once the application is developed and tested, several options exist for deployment:

Local Execution

The simplest way to run the application is locally with the command: streamlit run app.py

Streamlit Cloud

For easy sharing and public access, Streamlit Cloud offers a convenient platform to deploy Streamlit applications directly from a GitHub repository.

Render.com

A versatile platform that supports deploying various types of applications, including Python web apps. It provides features like continuous deployment and custom domains.

Hugging Face Spaces

A growing platform for hosting machine learning demos, offering free tiers and easy integration with Hugging Face models.

Possible Enhancements

This project lays a strong foundation for future improvements and expansions:

Add Attention Mechanism

Integrate an attention mechanism into the decoder to allow the model to focus on relevant parts of the input sentence during translation, significantly improving quality.

Visualize Attention Weights

If attention is implemented, visualize these weights in the Streamlit UI to offer insights into the model's decision-making process.

Support for Multiple Language Pairs

Extend the model to translate between other language pairs by collecting and training on corresponding datasets.

Bidirectional Encoder

Use Bidirectional LSTMs in the encoder to capture context from both forward and backward directions of the input sequence, leading to richer sentence embeddings.

Larger Datasets and Transfer Learning

Train on larger, more diverse datasets, or leverage pre-trained embedding models (e.g., Word2Vec, GloVe) or even more advanced pre-trained transformer models for fine-tuning.

Speech Integration

Allow users to input English sentences via voice and have the translated Hindi output spoken aloud.

Conclusion

Project Achievement Summary

This capstone project demonstrates the end-to-end development of a sophisticated English to Hindi translation system leveraging the power of Sequence-to-Sequence LSTM architecture in TensorFlow. Through detailed exploration of critical stages—from data preprocessing and tokenization to model construction and deployment—we've illustrated the practical application of neural machine translation principles.

Model Performance and Implementation

Despite training on a relatively modest dataset, the model achieved impressive translation results, confirming the effectiveness of our architectural choices and training methodology. The careful implementation of encoder-decoder components, attention to tokenizer preservation, and streamlined inference process ensures the system is not only academically sound but also deployment-ready.

Future Applications and Significance

This project serves as both a technical portfolio showcase and a practical foundation for further research, whether exploring attention mechanisms, implementing bidirectional encoders, or expanding to multiple language pairs. Ultimately, this work bridges theoretical machine learning concepts with real-world application development, providing valuable insights into the fascinating intersection of deep learning and language processing.