DON BOSCO INSTITUTE OF TECHNOLOGY

Bengaluru, Karnataka - 74

TEAM COUNT: FOUR

Project Title: Real-Time Language Translation Using Neural Machine Translation

Team Lead Name: Deepu M S

Team lead CAN ID: CAN_33695861

1. Name: Deepu MS

CAN ID: CAN_ 33695861

ROLE: FRONT END DEVELOPER

2. Name: Divya S

CAN ID: CAN_ 34140735

ROLE: RESEARCHER

3. Name: Spandana R

CAN ID: CAN_ 33695563

ROLE: BACK EN DEVELOPER

4. Name: Shama S P

CAN ID: CAN_ 33701582

ROLE:MACHINE LEARNING ENGINEER

PHASE 2 : Data preprocessing and model design

2.1 Overview of Data Preprocessing

The preprocessing stage is critical in developing a real-time neural machine translation system. It ensures the data is clean, consistent, and structured to train the neural network effectively. Below are the steps involved:

1. Dataset Collection

- **Parallel Corpus**: Collect a parallel dataset consisting of text pairs in the source and target languages. For example, English-French text pairs.

2. Text Cleaning

- **Normalization**:
  - Convert all text to lowercase (or consistent case).
  - Remove special characters, URLs, and unnecessary punctuation.

- Normalize unicode characters (e.g., accented characters to their base form if needed).
- **Whitespace Handling**:
  - Remove extra spaces and tabs.

3. **Data Augmentation (Optional)**

- **Back Translation**: Translate monolingual data from the target language back to the source language to create synthetic parallel data.
- **Noise Injection**: Add slight modifications to the data (e.g., swapping, deleting, or adding random words) to improve model robustness.

4. **Text Encoding**

- **Convert Tokens to IDs**:
  - Map each token to a numerical ID based on the vocabulary.
- **Embedding Preparation**:
  - Use pretrained embeddings (e.g., GloVe, FastText) or train embeddings from scratch during the model training process.

2.2 Data cleaning: Handling Missing Value, Outliers, and Inconsistencies

1. Handling Missing Values

Missing values in parallel corpora occur when either the source or target text is incomplete or absent.

Issues:
- Untranslated sentences (e.g., source exists, but target is missing).
- Empty lines or entries with only whitespace.

Strategies:

1. **Remove Incomplete Pairs**:
   - Automatically discard sentence pairs where either the source or target text is missing.
2. **Infer Missing Translations (Rare)**:

o For domain-specific data, use an auxiliary machine translation model or human translators to fill gaps if the dataset is too small.

o Example: If the source is "Good morning," and the target is missing, manually add "Bonjour" in the target language.

3. **Replace Missing Entries with Placeholders (for consistency)**:

o Add placeholder tokens (e.g., <UNK>, <MISSING>) to indicate missing information, but this is usually avoided in final training data.

2. Handling Outliers

Outliers in text data refer to unusually long or short sentences or text pairs that deviate significantly from the rest of the dataset.

Issues:

- Extremely long sequences can cause memory issues during training.
- Very short sequences (e.g., single words) might not provide meaningful learning signals.

Strategies:

1. **Set Sequence Length Limits**:

o Define maximum and minimum length thresholds for both source and target sentences.

o Example:

▪ Minimum: 3 tokens (to remove overly short sentences).

▪ Maximum: 50 tokens (to reduce processing overhead).

2. **Truncate Long Sentences**:

o If a sentence exceeds the maximum length, truncate it while ensuring meaning is retained.

o Example: From "The quick brown fox jumps over the lazy dog in the park," truncate to "The quick brown fox jumps over the lazy dog."

3. Handling Inconsistencies

Inconsistencies include issues like mismatched sentence pairs, encoding problems, or incorrect formatting.

Issues:

- Misaligned sentence pairs (source and target text do not correspond).
- Encoding problems (e.g., unrecognized characters due to different language encodings).
- Variations in punctuation, contractions, or spelling.

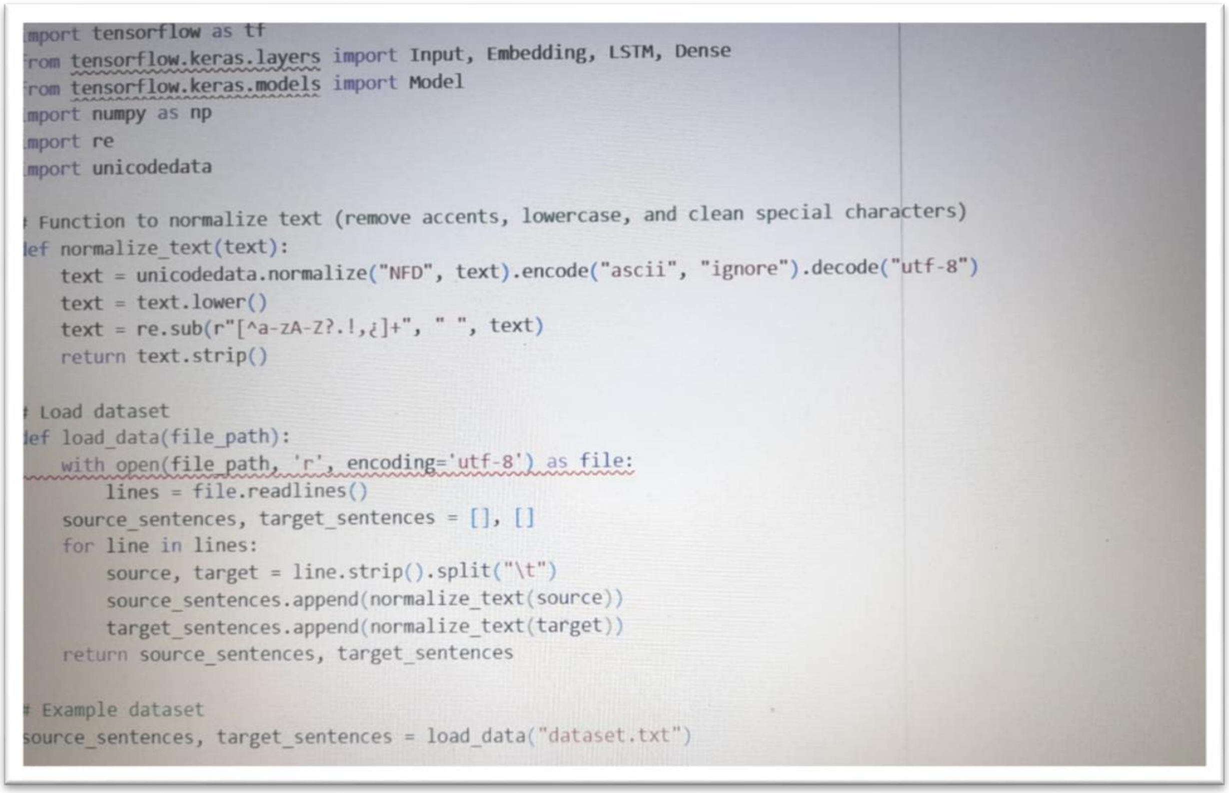
Strategies:

1. **Sentence Alignment**:

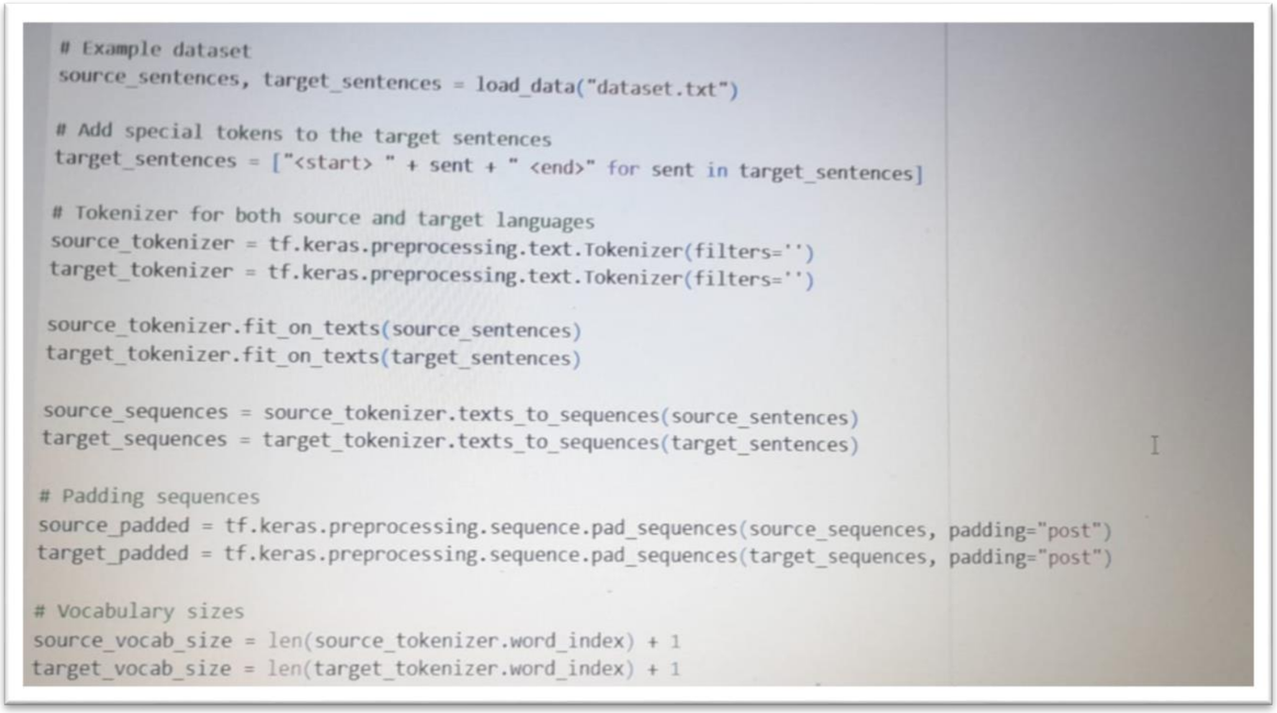
- Use alignment tools (e.g., **Moses Toolkit**, **fast_align**) to verify that source and target sentences are properly paired.
- Misaligned pairs can either be realigned or discarded.

2. **Encoding Normalization**:

- Convert all text to a common encoding (e.g., UTF-8) to avoid unrecognized symbols.
- Use libraries like Python's unicodedata to standardize characters.



```
mport tensorflow as tf
rom tensorflow.keras.layers import Input, Embedding, LSTM, Dense
rom tensorflow.keras.models import Model
mport numpy as np
mport re
mport unicodedata

 Function to normalize text (remove accents, lowercase, and clean special characters)
lef normalize_text(text):
    text = unicodedata.normalize("NFD", text).encode("ascii", "ignore").decode("utf-8")
    text = text.lower()
    text = re.sub(r"[^a-zA-Z?.!,¿]+", " ", text)
    return text.strip()

 Load dataset
lef load_data(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()
    source_sentences, target_sentences = [], []
    for line in lines:
        source, target = line.strip().split("\t")
        source_sentences.append(normalize_text(source))
        target_sentences.append(normalize_text(target))
    return source_sentences, target_sentences

 Example dataset
ource_sentences, target_sentences = load_data("dataset.txt")
```



```
# Example dataset
source_sentences, target_sentences = load_data("dataset.txt")

# Add special tokens to the target sentences
target_sentences = ["<start> " + sent + " <end>" for sent in target_sentences]

# Tokenizer for both source and target languages
source_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
target_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')

source_tokenizer.fit_on_texts(source_sentences)
target_tokenizer.fit_on_texts(target_sentences)

source_sequences = source_tokenizer.texts_to_sequences(source_sentences)
target_sequences = target_tokenizer.texts_to_sequences(target_sentences)

# Padding sequences
source_padded = tf.keras.preprocessing.sequence.pad_sequences(source_sequences, padding="post")
target_padded = tf.keras.preprocessing.sequence.pad_sequences(target_sequences, padding="post")

# Vocabulary sizes
source_vocab_size = len(source_tokenizer.word_index) + 1
target_vocab_size = len(target_tokenizer.word_index) + 1
```

2.3 Feature Scaling and Normalization

1. Token Embeddings

Word or sub word tokens in NMT are represented as numerical embeddings. These embeddings are trained or initialized using pretrained vectors (e.g., GloVe, FastText).

Normalization Strategy:

- **L2 Normalization**: Normalize embeddings to ensure they have a unit norm.

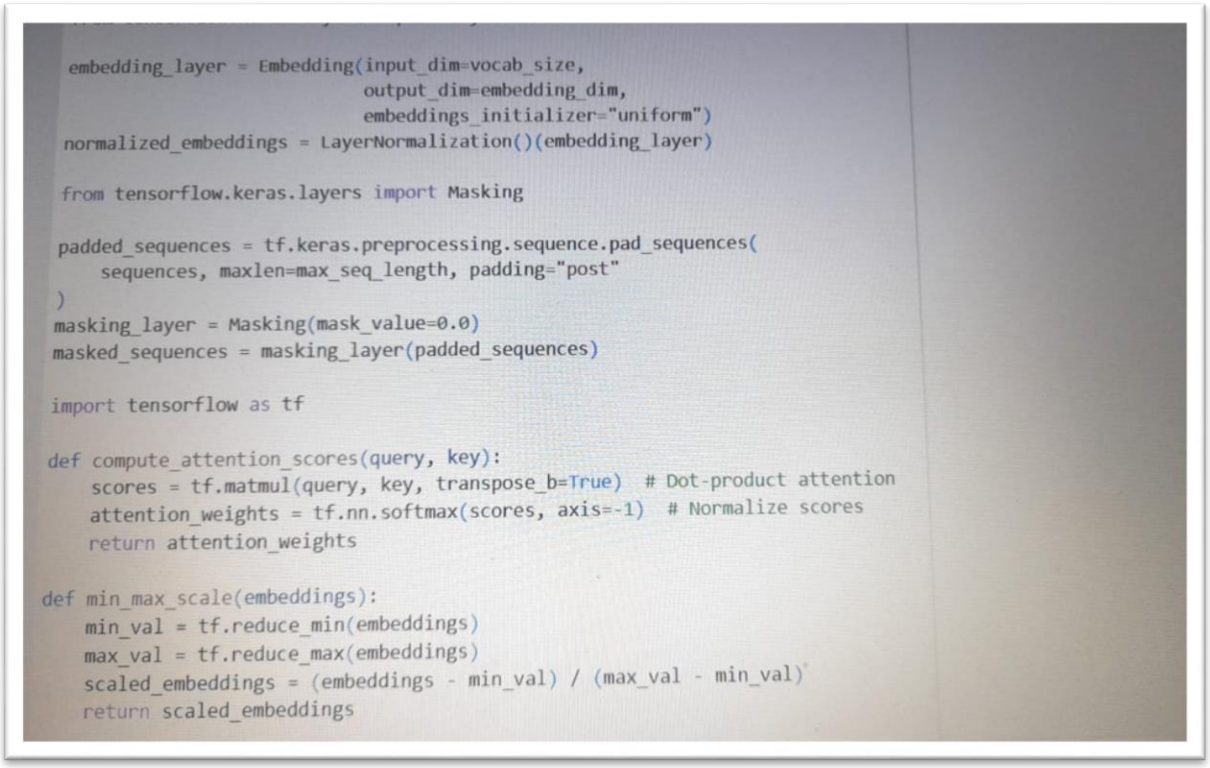
2. Gradient Normalization

During training, exploding gradients can be a problem, especially in RNN-based models like LSTMs. Gradient clipping is used to normalize gradients and stabilize training.

Normalization Strategy:

- **Gradient Clipping**:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, clipnorm=1.0)
```



```
embedding_layer = Embedding(input_dim=vocab_size,
                            output_dim=embedding_dim,
                            embeddings_initializer="uniform")
normalized_embeddings = LayerNormalization()(embedding_layer)

from tensorflow.keras.layers import Masking

padded_sequences = tf.keras.preprocessing.sequence.pad_sequences(
    sequences, maxlen=max_seq_length, padding="post"
)
masking_layer = Masking(mask_value=0.0)
masked_sequences = masking_layer(padded_sequences)

import tensorflow as tf

def compute_attention_scores(query, key):
    scores = tf.matmul(query, key, transpose_b=True)  # Dot-product attention
    attention_weights = tf.nn.softmax(scores, axis=-1)  # Normalize scores
    return attention_weights

def min_max_scale(embeddings):
    min_val = tf.reduce_min(embeddings)
    max_val = tf.reduce_max(embeddings)
    scaled_embeddings = (embeddings - min_val) / (max_val - min_val)
    return scaled_embeddings
```

2.4 Feature Scaling and Normalization

1. Feature scaling

Feature transformation involves converting raw input data (e.g., words or subwords) into numerical representations that a neural network can process.

Techniques Used:

(a) Word Embeddings

- Word embeddings map discrete tokens (e.g., words or subwords) into dense, continuous vector spaces.
- Pretrained embeddings like **GloVe**, **Word2Vec**, or **FastText** can be used, or embeddings can be learned during training.

(b) Subword Representations

- Instead of full words, subwords (e.g., from **Byte Pair Encoding (BPE)** or **SentencePiece**) are used to handle rare or unknown words and reduce vocabulary size.

(c) One-Hot Encoding (Rarely Used for NMT)

- One-hot encoding is used in simple models but is memory-intensive for large vocabularies, so embeddings are preferred.

2. Dimensionality Reduction

Dimensionality reduction reduces the computational burden and eliminates redundant features while preserving the most critical information.

Techniques Used:

(a) Principal Component Analysis (PCA)

- PCA can be used to reduce the dimensionality of pretrained embeddings, especially when using high-dimensional embeddings (e.g., 300+ dimensions).

(b) Singular Value Decomposition (SVD)

- SVD is another effective technique for reducing the dimensions of embedding matrices or attention weight matrices.

(c) Autoencoders

- Autoencoders can compress high-dimensional features into lower-dimensional representations using neural networks.

Importance of Dimensionality Reduction

Dimensionality reduction is critical in real-time NMT systems as it:

- Enhances computational efficiency and scalability.
- Reduces overfitting and improves generalization.
- Optimizes memory and latency for deployment on resource-constrained devices.
- Ensures models remain interpretable, lightweight, and capable of delivering high-quality translations in real-time scenarios.

2.5 Autoencoder model Design

1. Encoder Architecture

The encoder processes input sequences and generates a latent representation. For NMT, the encoder often uses an **embedding layer** followed by an **LSTM**, **GRU**, or a **Transformer Encoder**.

2. Decoder Architecture

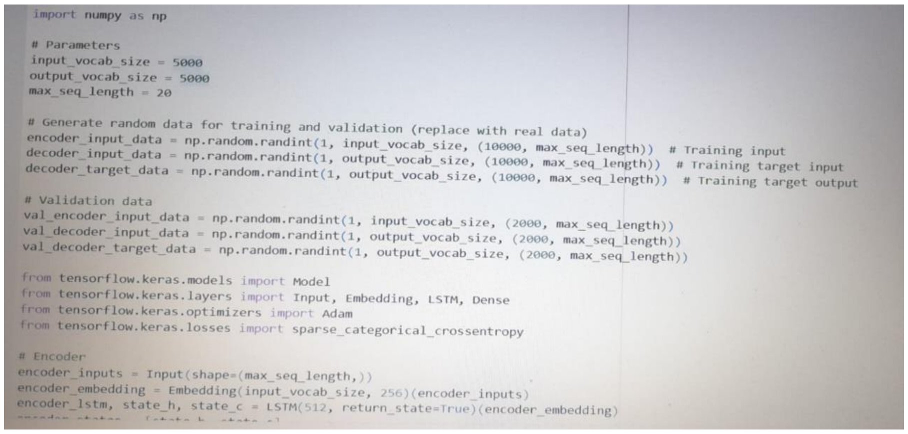
The decoder takes the latent representation from the encoder and generates the target language sequence.

3. Loss Function

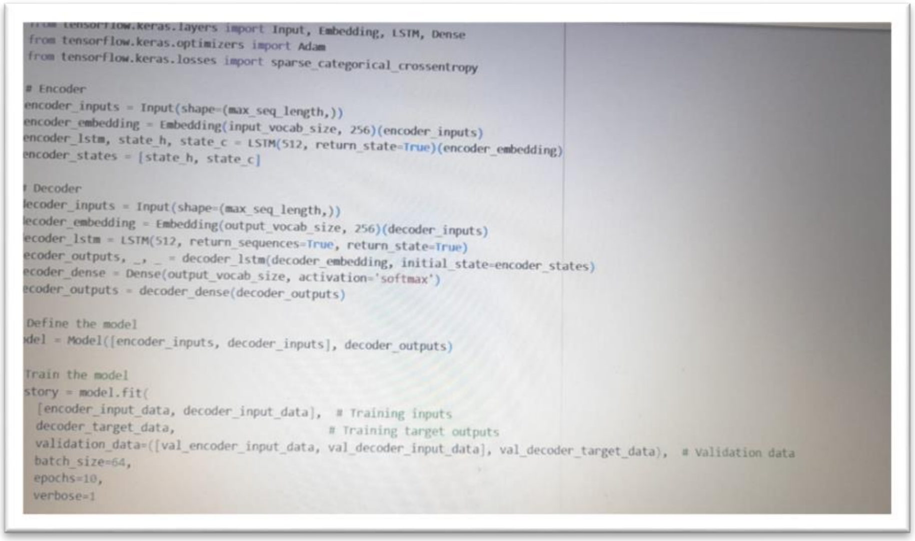
The loss function for the NMT model is usually **categorical cross-entropy**, which measures the difference between the predicted and actual token distributions.

2.6 Model Training and Validation

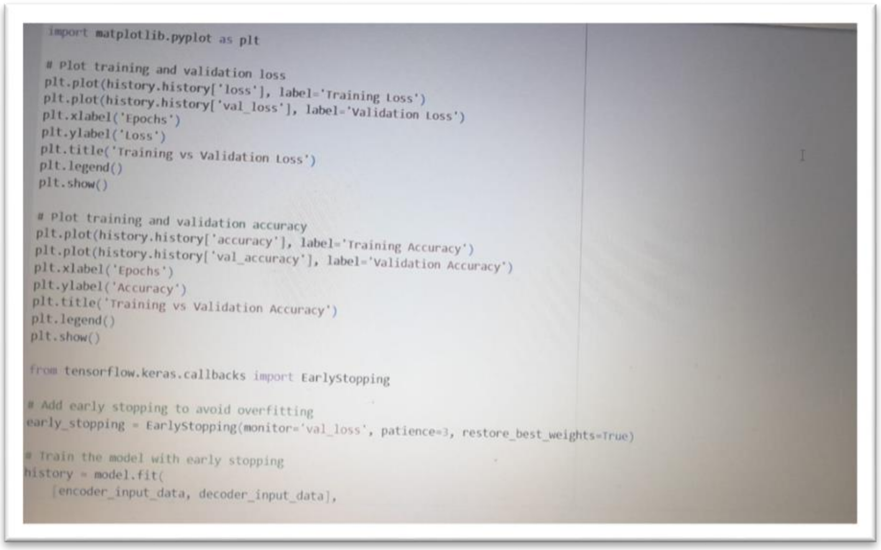
1. **Training**:
   - Input: Encoder and decoder sequences.
   - Output: Decoder target sequences.
2. **Validation**:
   - Monitor validation loss to assess generalization.
3. **Early Stopping**:
   - Prevent overfitting by stopping training when validation loss stagnates.



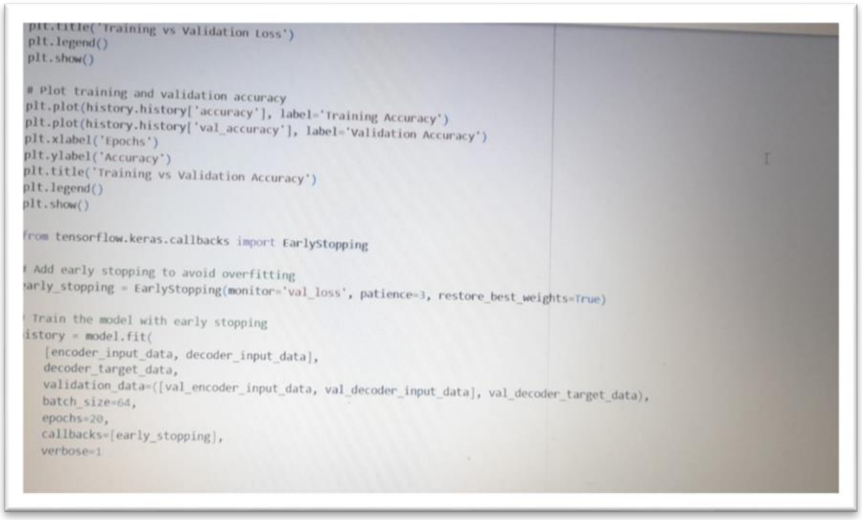
```
import numpy as np

# Parameters
input_vocab_size = 5000
output_vocab_size = 5000
max_seq_length = 20

# Generate random data for training and validation (replace with real data)
encoder_input_data = np.random.randint(1, input_vocab_size, (10000, max_seq_length))  # Training input
decoder_input_data = np.random.randint(1, output_vocab_size, (10000, max_seq_length))  # Training target input
decoder_target_data = np.random.randint(1, output_vocab_size, (10000, max_seq_length))  # Training target output

# Validation data
val_encoder_input_data = np.random.randint(1, input_vocab_size, (2000, max_seq_length))
val_decoder_input_data = np.random.randint(1, output_vocab_size, (2000, max_seq_length))
val_decoder_target_data = np.random.randint(1, output_vocab_size, (2000, max_seq_length))

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import sparse_categorical_crossentropy

# Encoder
encoder_inputs = Input(shape=(max_seq_length,))
encoder_embedding = Embedding(input_vocab_size, 256)(encoder_inputs)
encoder_lstm, state_h, state_c = LSTM(512, return_state=True)(encoder_embedding)
```



```
tensorflow.keras.layers import Input, Embedding, LSTM, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import sparse_categorical_crossentropy

# Encoder
encoder_inputs = Input(shape=(max_seq_length,))
encoder_embedding = Embedding(input_vocab_size, 256)(encoder_inputs)
encoder_lstm, state_h, state_c = LSTM(512, return_state=True)(encoder_embedding)
encoder_states = [state_h, state_c]

# Decoder
lecoder_inputs = Input(shape=(max_seq_length,))
lecoder_embedding = Embedding(output_vocab_size, 256)(decoder_inputs)
ecoder_lstm = LSTM(512, return_sequences=True, return_state=True)
ecoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=encoder_states)
ecoder_dense = Dense(output_vocab_size, activation='softmax')
ecoder_outputs = decoder_dense(decoder_outputs)

 Define the model
del = Model([encoder_inputs, decoder_inputs], decoder_outputs)

Train the model
story = model.fit(
  [encoder_input_data, decoder_input_data],  # Training inputs
  decoder_target_data,                       # Training target outputs
  validation_data=([val_encoder_input_data, val_decoder_input_data], val_decoder_target_data),  # Validation data
  batch_size=64,
  epochs=10,
  verbose=1
```



```
import matplotlib.pyplot as plt

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training vs Validation Loss')
plt.legend()
plt.show()

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training vs Validation Accuracy')
plt.legend()
plt.show()

from tensorflow.keras.callbacks import EarlyStopping

# Add early stopping to avoid overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Train the model with early stopping
history = model.fit(
    [encoder_input_data, decoder_input_data],
```



```
plt.title('Training vs Validation Loss')
plt.legend()
plt.show()

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training vs Validation Accuracy')
plt.legend()
plt.show()

from tensorflow.keras.callbacks import EarlyStopping

# Add early stopping to avoid overfitting
arly_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

 Train the model with early stopping
istory = model.fit(
    [encoder_input_data, decoder_input_data],
    decoder_target_data,
    validation_data=([val_encoder_input_data, val_decoder_input_data], val_decoder_target_data),
    batch_size=64,
    epochs=20,
    callbacks=[early_stopping],
    verbose=1
```

2.7 Conclusion

Real-time language translation using Neural Machine Translation (NMT) offers immense potential to bridge communication gaps across languages. By developing a system that utilizes cutting-edge deep learning techniques, we can provide a high-quality, real-time translation experience that benefits users in various sectors. While there are technical challenges to address, such as latency and accuracy, the ongoing advancements in AI and machine learning make this an achievable and impactful project. We propose a unified framework to do neural simultaneous machine translation. To trade off quality and delay, we extensively explore various tar-gets for delay and design a method for beam-search applicable in the simultaneous MT setting. Experiments against state-of-the-art baselines on two language pairs demonstrate the efficacy both quantitatively and qualitatively.

The combination of robust **data preprocessing** and effective **model design** enables real-time language translation using neural machine systems. By leveraging advanced techniques like attention mechanisms, transformer architectures, and dimensionality reduction, modern NMT systems achieve high accuracy and scalability across languages. Future advancements in multilingual and zero-shot translation further promise universal, real-time language accessibility.