Post Graduate Program *in*

Artificial Intelligence & Machine Learning

(Great Learning & UT Austin)

Capstone Project Report

Machine Translation

By:

NLP 2 Group 11

Bipin Kumar

Jeffry Jones

Rajat Agrawal

Rishi Barve

Soumit Kundu

Table of Contents

Introduction

Language is the primary means of human communication, relying on both grammar and vocabulary to convey meaning. In contrast, computers operate using Machine Language, which consists of numeric codes in the form of algorithms—strings of 0s and 1s known as bits—that enable them to execute operations directly. However, computers cannot understand human language in its natural form.

To bridge this gap between humans and machines, Natural Language Processing (NLP) was developed. NLP allows computers to process and interpret human speech, making it possible to analyze and respond to what users say. By combining computational linguistics with Artificial Intelligence (AI), NLP enables effective communication between humans and computers. This technology now powers everyday applications like Alexa and Siri.

For a computer program to engage in meaningful communication with humans, it must understand grammatical syntax, word meanings (semantics), correct tense usage (morphology), and conversational context (pragmatics). The complexity of these requirements has historically posed significant challenges for NLP, leading to the failure of earlier approaches. However, modern NLP has shifted from rule-based methods to pattern-learning-based programming, resulting in more effective language processing.

In this project, we explore the development of a Machine Translation model designed to translate text between languages. Specifically, we have focused on translating sentences between German and English. The goal of this project is to facilitate communication and idea exchange between people from different countries, breaking down language barriers and fostering global collaboration.

Problem Statement

The task is to design a Sequence-to-Sequence model for Machine Translation, that can be used to translate sentences from German to English language

The dataset comes from ACL2014 Ninth workshop on Statistical Machine Translation. This workshop mainly focused on language translation between European language pairs. The idea behind the workshop was to provide the ability for two parties to communicate and exchange the ideas from different countries.

Three datasets are used here –

- Common Crawl Dataset: General collection of parallel German-English sentences.
- Europarl Dataset: Collection of text from the European Parliament proceedings.
- News Commentary Dataset: Collection of text from news articles.

We will use sequence-to-sequence models like RNN and LSTM models, build different variants of these models, to find the algorithm best suited for the machine translation task.

Milestone 1
Data Exploration, Cleaning, Pre-processing and Building Simple Sequence-to-Sequence Models

1. Import and Read Files

We have 3 sets of files in English and German each.

- Common Crawl and contain about 2 million sentences.

```
Length of Common Crawl English Sentences File: 2399123
Length of Common Crawl German Sentences File: 2399123

English Sentence: iron cement is a ready for use paste which is laid as a fillet by putty

German Sentence: iron cement ist eine gebrauchs-fertige Paste, die mit einem Spachtel oder


Length of Europarl English Sentences File: 1920209
Length of Europarl German Sentences File: 1920209

English Sentence: Resumption of the session

German Sentence: Wiederaufnahme der Sitzungsperiode
```

- News Commentary file contain about 200 K sentences

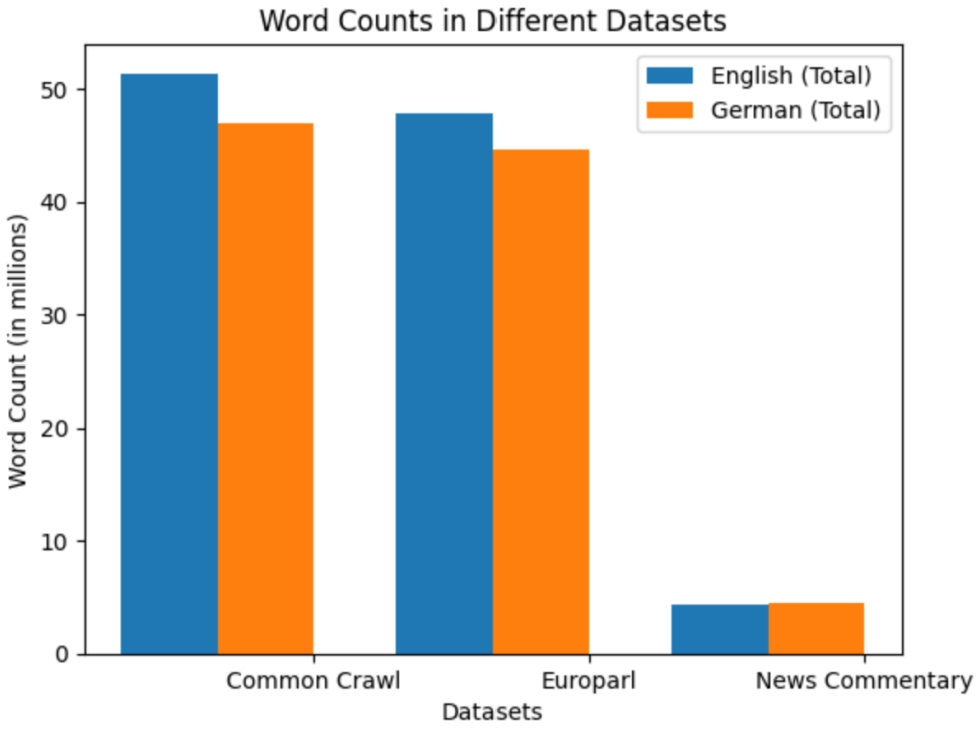
```
Length of News Commentary English Sentences File: 201995
Length of News Commentary German Sentences File: 201854

English Sentence: $10,000 Gold?

German Sentence: Steigt Gold auf 10.000 Dollar?
```

News Commentary English File has 141 more sentences than the German File

2. Exploratory Data Analysis

Check Vocabulary Size

Total Word Count



Observations

- Both Common Crawl and Europarl have a huge total word count in English and German (~50 million), which is expected considering the no of sentences in both.
- News Commentary has a much smaller total word count of ~5 million
- The total word count for English is slightly higher than German for both Common Crawl and Europarl, but almost equal across for News Commentary

Unique Word Count

```
Common Crawl:
English (Unique)- 1532712
German (Unique)- 2557806

Europarl:
English (Unique)- 271951
German (Unique)- 616947

News Commentary:
English (Unique)- 139001
German (Unique)- 139001
```

Observations:

- Even though Common Crawl and Europarl have similar file and vocab size, the unique word count differs considerably, with Common Crawl having unique count of 1.5 – 2.5 million, while Europarl having unique count in 200-600 K.
- In both of these, Unique word count in German is much higher than that in English.
- Interestingly, in News Commentary, both English and German have same unique word count

Check and Remove Blank Lines

In the dataset, we see blank lines as well. It seems that in most cases, 2 lines seem to have been combined in one line, leaving the next line as blank.

To handle these blank lines, we

- Check for the blank lines in each dataset's German and English Language files and store the corresponding indices
- Combine indices of blank lines in both German and English Files
- Remove lines with these combined indices from both German and English files, so that same lines are removed from each file and rest of the lines remain aligned

Function to check for blank lines

```
# Define function to check for blank lines in given file

def check_blank(list_name):
    blank_lines_index = []
    for i in range(len(list_name)):
        if list_name[i] == '\n':
            blank_lines_index.append(i)

    return blank_lines_index
```

Results

```
No of blank lines in Common Crawl English File: 0
No of blank lines in Common Crawl German File: 0
No of blank lines in Europarl English File: 8366
No of blank lines in Europarl German File: 2923
No of blank lines in News Commentary English File: 322
No of blank lines in News Commentary German File: 224
```

```
Europarl English File Length, after removing blank lines: 1908920
Europarl German File Length: 1908920

News Commentary English File Length, after removing blank lines: 201449
News Commentary German File Length,after removing blank lines: 201309
```

There are no blanks in Common Crawl files.

After removing blank lines from News Commentary, we still have diff of 140 lines between English and German files

3. Merging Datasets

Because of the huge size of datasets, it is not feasible to use entire datasets for training and testing, due to amount of computing resource and training time which will be required.

So, for training, validation and testing, 10000 sentences from each of the 3 datasets are taken and merged to form common English and German datasets

```
# Merge the three lists
en_sentences = common_crawl_en[:10000] + europarl_en[:10000] + news_commentary_en[:10000]
de_sentences = common_crawl_de[:10000] + europarl_de[:10000] + news_commentary_de[:10000]

# Print the lengths of the merged lists
print('Length of Merged English File:', len(en_sentences))
print('Length of Merged German File:', len(de_sentences))
```

```
Length of Merged English File: 30000
Length of Merged German File: 30000
```

For ease of viewing, data cleaning and visualization, we create a dataframe

```
# Create a DataFrame from the merged lists
df = pd.DataFrame({'english': en_sentences, 'german': de_sentences})

df.head()
```

	english	german
0	iron cement is a ready for use paste which is ...	iron cement ist eine gebrauchs-fertige Paste, ...
1	iron cement protects the ingot against the hot...	Nach der Aushärtung schützt iron cement die Ko...
2	a fire restant repair cement for fire places, ...	feuerfester Reparaturkitt für Feuerungsanlagen...
3	Construction and repair of highways and...\n	Der Bau und die Reparatur der Autostraßen...\n
4	An announcement must be commercial character.\n	die Mitteilungen sollen den geschäftlichen kom...

4. Data Cleaning and Pre-Processing

Previously, we have already checked and removed all blank lines from each of the files, before selecting the sample data from each file and merging them.

As part of data cleaning, we will now check for any NULL values, remove punctuation marks and covert the text to lowercase

Check for NULL values

As blank lines were removed, no NULL values are remaining

```
# Check for any NULL values
df.isnull().sum()
```

	0
english	0
german	0

Remove Punctuation Marks and Convert to Lowercase

We define a function to perform required operations on English and German Sentence columns in dataframe and add cleaned columns

```
import string

# Define function to remove puntuation marks and numbers, and then convert to lowercase
def clean_text(text):
    text = text.translate(str.maketrans('', '', string.punctuation)).lower()
    return text
```

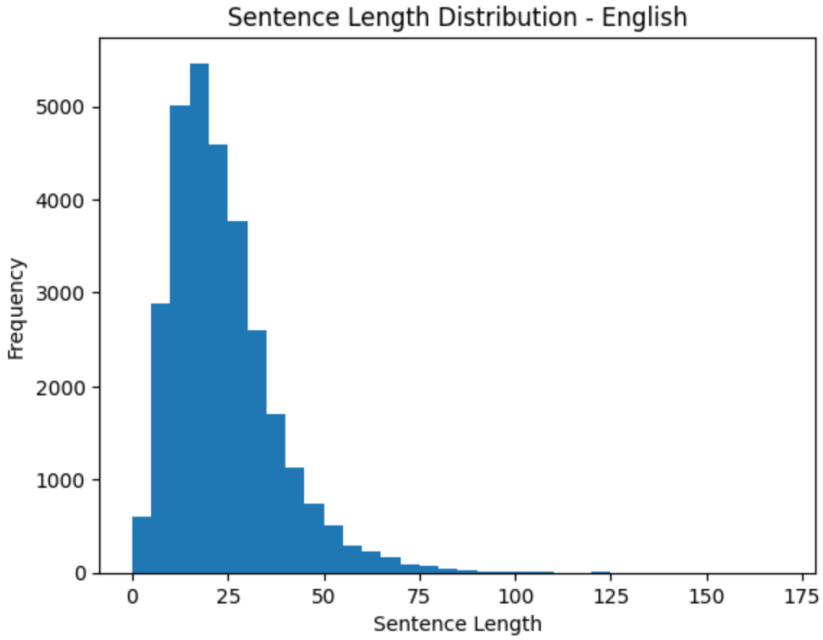
```
df['english_cleaned'] = df['english'].apply(lambda x: clean_text(x))
df['german_cleaned'] = df['german'].apply(lambda x: clean_text(x))

df.tail()
```

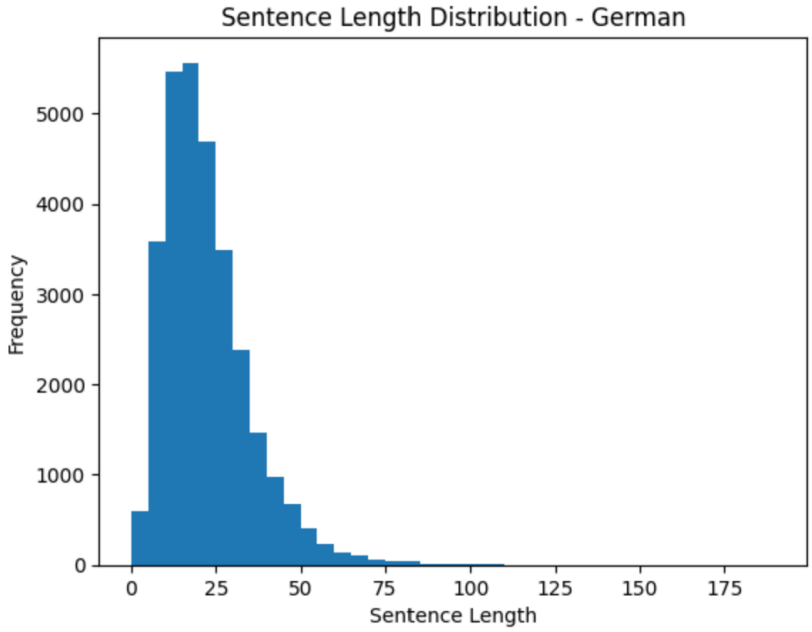
	english	german	english_cleaned	german_cleaned
29995	If the developed world is able to pay trillion...	Wenn die Industrieländer in der Lage sind, meh...	if the developed world is able to pay trillion...	wenn die industrieländer in der lage sind mehr...
29996	Clearly this is not about the availability of ...	Es geht dabei eindeutig nicht um das Vorhanden...	clearly this is not about the availability of ...	es geht dabei eindeutig nicht um das vorhanden...
29997	It is about the inappropriate priorities in ho...	Es geht um unangemessene Prioritäten und darum...	it is about the inappropriate priorities in ho...	es geht um unangemessene prioritäten und darum...
29998	It is about moral values that make it appropri...	Es geht um moralische Werte, die es angemessen...	it is about moral values that make it appropri...	es geht um moralische werte die es angemessen ...
29999	I cannot believe that people in developed coun...	Ich kann nicht glauben, dass die Menschen in d...	i cannot believe that people in developed coun...	ich kann nicht glauben dass die menschen in de...

We will now use these cleaned columns for further processing

Check Sentence Length Distribution and Max Sentence Length

We now check the sentence length distribution of cleaned English and German Sentences





Observations

- Distribution of English and German sentence lengths are similar most sentences made up of 20-25 words.
- There are a few outliers with very long sentences, in both English and German datasets, resulting in a long right skew.

- Long right skew indicates that there are more shorter sentences than longer ones.
- Both have maximum sentence lengths of > 100

Checking maximum length of cleaned English and German sentences in sample data

```
Maximum English Sentence length: 172
Maximum German Sentence length: 193
```

Tokenization and Padding

Tokenization is a process to assign a unique id to each word in the dataset vocabulary, which is then used to convert each sentence to a sequence of word IDs. This is done as we need to convert words to numerical values before using as input for deep learning model.

Also, when we feed these sequences of word IDs into the model, each sequence needs to be the same length. Since sentences are dynamic in length, we add padding to the end of the sequences to make them the same length.

Then we create a function to perform tokenization and padding, in which

- All the words of the sentence are tokenized. As we have sentences in 2 languages, two tokenizers are defined - an English Tokenizer and a German tokenizer.
- Then based on the calculated maximum length of sentences in sample data, we perform padding for English and German sequences.

```
# Define function to perform tokenization
def tokenization(sentences, max_length):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(sentences)
    sequences = tokenizer.texts_to_sequences(sentences)
    padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post')

    return tokenizer, padded_sequences
```

The function returns the tokenizer and padded sequences.

Using tokenizer, we can find the sampled English and German vocabulary size

```
English Vocabulary size: 34856
German Vocabulary size: 59642
```

Even with just 10000 sentences, we get a huge vocabulary for both English and German.

We also check the Padded sequence size

```
English Padded Sequence Shape: (30000, 172)
German Padded Sequence Shape: (30000, 193)
```

Both English and German sentence lengths are standardized.

Sample English and German sentences and their respective padded sequences

```
Actual English Sentence:
iron cement is a ready for use paste which is laid as a fillet by putty knife or finger in the mould edges corners of the steel ingot mould

Tokenized and Padded English Sentence:
[ 3723  8346     7     6  1482     9   129 13675    22     7  1946    16
     6 18499    21 18500 18501    31  9565     5     1 11138  9566  4880
     2     1  1430 11139 11138     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0]
```

```
Actual German Sentence:
iron cement ist eine gebrauchsfertige paste die mit einem spachtel oder den fingern als hohlkehle in die formecken winkel der stahlguss kokille aufgetragen wird

Tokenized and Padded German Sentence:
[17746 13715     9    12 26073 17747     1    14    48 26074    40     6
 26075    26 26076     4     1 26077 17748     2 17749 17750 26078    29
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0]
```

5. Model Selection and Architecture

Model Selection

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) models are widely used in language translation tasks due to their ability to handle sequential data. Here are the key advantages of using these models:

- **RNN Model**: RNNs are designed to process sequences of data, making them well-suited for tasks like language translation where the order of words matters. They can capture temporal dependencies by maintaining a hidden state that evolves as the input sequence is processed.

  RNNs can retain information about previous words in a sequence, which helps in understanding the context necessary for translating sentences correctly.
- **LSTM Model**: LSTMs, a specialized form of RNNs, are particularly effective at handling long sequences. They address the vanishing gradient problem, which RNNs often face when dealing with long-term dependencies. Selectively remembering and forgetting information through their gating mechanisms (input, forget, and output gates), enables them to maintain context over longer sequences, improving the translation of complex sentences.

We will build, train and test below Models

- Basic RNN and LSTM Models without Embeddings (Milestone -1)
- RNN and LSTM Model with Embeddings (Milestone -2)
- Bi-Directional RNN and LSTM Models (Milestone -2)
- Encoder-Decoder RNN & LSTM Models (Milestone -2)

Model Architecture

We will build a Sequential Model with each Model Architecture having one of more layers from below

- Input Layer: which takes the English Sequences as input
- Reshape Layer: Reshapes the input sequence to be compatible with given model
- Embedding Layer: allows for representation of words in a dense vector space, capturing their semantic meanings and relationships, leading to better performance.
- Model Layer: RNN or LSTM model. We will keep the no of units small to ensure that the model can work on the limited computing power.
- Dropout Layer: Dropout of X% to make model more robust
- Time Distributed Layer: Ensures that the model can predict the entire sequence of words in the target language, with each time step being independently processed but contributing to the overall sequence prediction.

- Dense Layer: Neural Network layer with the number of neurons corresponding to the size of German vocabulary. This allows the model to predict probabilities for each German word.
- Activation Layer: Softmax Activation is to ensure the output of the dense layer is a probability distribution, where each value represents the likelihood of a specific German word being the correct translation.

Model Compilation

- Optimizer: Adam optimizer with a learning rate and clipvalue.

  Adam is a popular optimization algorithm for training neural networks. It adapts the learning rate for each parameter during training.

  A smaller learning rate leads to slower but potentially more stable convergence.

  Clip Value limits the magnitude of the gradients to prevent exploding gradients, which can cause instability during training
- Loss Function: sparse_categorical_crossentropy is used as we are dealing with a multi-class classification problem where each word in the target sentence is a class, and the labels are integers.
- Metrics: accuracy is a common metric to evaluate the performance of classification models. It measures the percentage of correctly predicted words.

6. Simple Model Build, Training and Testing

Train-Test Split

We split the padded sequences into Train and Test data. 10% of sample is taken for testing

```
Shape of X_train: (27000, 172)
Shape of y_train: (27000, 193)
Shape of X_test: (3000, 172)
Shape of y_test: (3000, 193)
```

English sequences are the input, so X_train has length of max English sentence length from sample

German sequences are the output, so y_train has length of max German sentence length from sample

Simple RNN Model (without Embeddings)

First, we build a Base RNN Model without Embeddings, with

- Input Layer
- SimpleRNN layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

We also reshape the input and output to add a 3rd dimension to make it compatible with RNN

Model Summary

Model: "sequential"

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 193, 64)	4,224
dropout (Dropout)	(None, 193, 64)	0
dense (Dense)	(None, 193, 59642)	3,876,730

Total params: 3,880,954 (14.80 MB)
Trainable params: 3,880,954 (14.80 MB)
Non-trainable params: 0 (0.00 B)

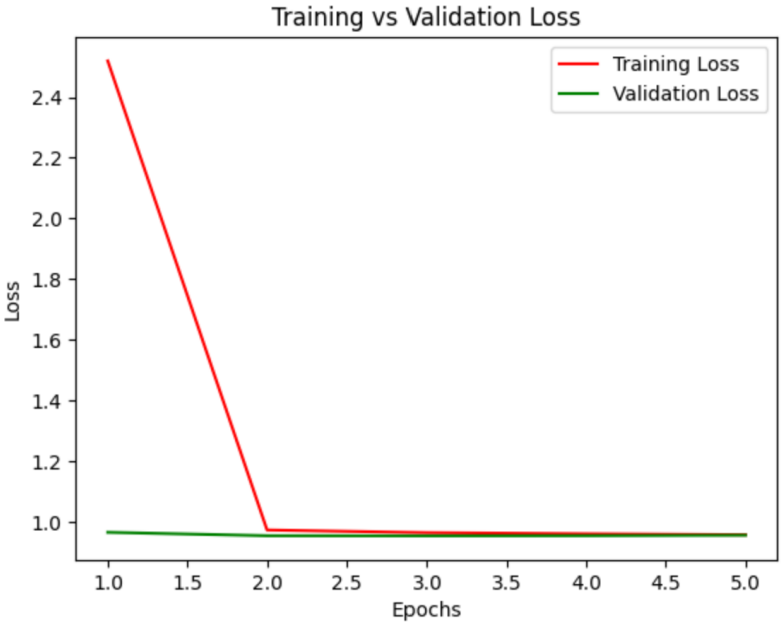
Model Training

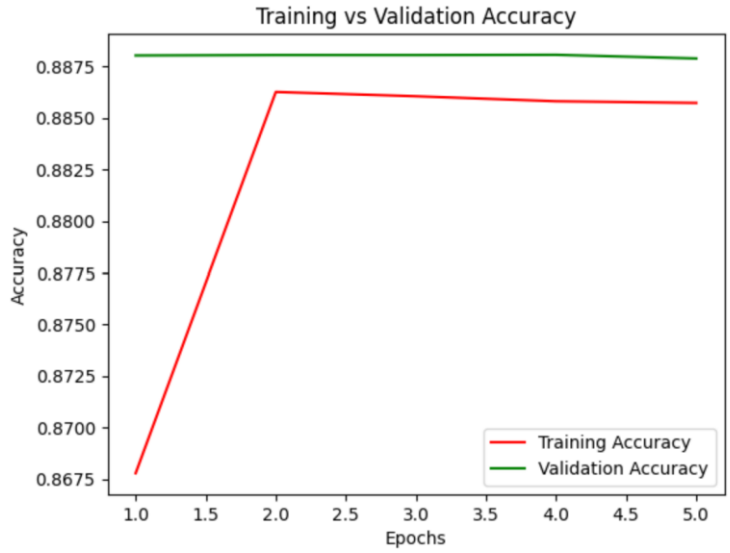
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_simple_rnn_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

simple_rnn_history = simple_rnn.fit(X_train_reshaped, y_train_reshaped, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 115s 303ms/step - accuracy: 0.8123 - loss: 5.1235 - val_accuracy: 0.8880 - val_loss: 0.9663
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 123s 279ms/step - accuracy: 0.8867 - loss: 0.9715 - val_accuracy: 0.8881 - val_loss: 0.9551
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 141s 276ms/step - accuracy: 0.8862 - loss: 0.9641 - val_accuracy: 0.8881 - val_loss: 0.9547
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 143s 279ms/step - accuracy: 0.8853 - loss: 0.9655 - val_accuracy: 0.8881 - val_loss: 0.9553
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 141s 278ms/step - accuracy: 0.8863 - loss: 0.9539 - val_accuracy: 0.8879 - val_loss: 0.9568
```





Observations:

- In first Epoch, when weights are randomized, the loss is high and accuracy is lower.
- From 2nd Epoch onwards, loss comes down to ~0.97 and then remains pretty much same
- Same is seen for accuracy as well, from 2nd epoch onwards, accuracy remains at ~88%
- This RNN model shows stable performance on validation data throughout the epochs, as evident from straight line
- Maximum Training Accuracy is 88.67%, and Maximum Validation Accuracy is 88.81%
- Since the training and validation loss and accuracy are pretty close, which means model is not overfitting.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = simple_rnn_model.evaluate(X_test_reshaped[:100], y_test_reshaped[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 0.9517783522605896
Test Accuracy: 0.8900517821311951
```

Even with a Simple RNN model without Embedding, we have a very good accuracy of 89%

But the loss remains high at 0.95.

Simple LSTM Model (without Embeddings)

Now, we build a Base LSTM Model without Embeddings, with

- Input Layer
- LSTM layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

Here also, we use the same reshaped input and output training data

Model Summary

Model: "sequential_1"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 193, 64)	16,896
dropout_1 (Dropout)	(None, 193, 64)	0
dense_1 (Dense)	(None, 193, 59642)	3,876,730

Total params: 3,893,626 (14.85 MB)
Trainable params: 3,893,626 (14.85 MB)
Non-trainable params: 0 (0.00 B)

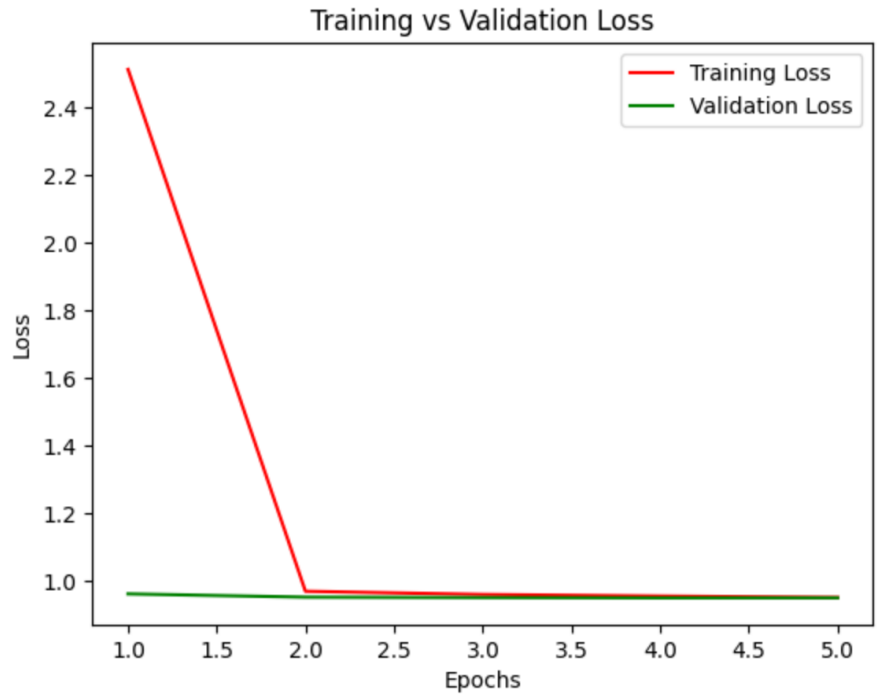
Model Training

Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_simple_lstm_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

simple_lstm_history = simple_lstm.fit(X_train_reshaped, y_train_reshaped, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 203s 586ms/step - accuracy: 0.8665 - loss: 5.1829 - val_accuracy: 0.8877 - val_loss: 0.9630
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 198s 584ms/step - accuracy: 0.8873 - loss: 0.9686 - val_accuracy: 0.8877 - val_loss: 0.9534
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 203s 586ms/step - accuracy: 0.8864 - loss: 0.9620 - val_accuracy: 0.8881 - val_loss: 0.9520
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 214s 622ms/step - accuracy: 0.8866 - loss: 0.9571 - val_accuracy: 0.8884 - val_loss: 0.9511
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 249s 585ms/step - accuracy: 0.8874 - loss: 0.9491 - val_accuracy: 0.8885 - val_loss: 0.9514
```


Training vs Validation Loss
Training Loss
Validation Loss
Loss
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Epochs


Training vs Validation Accuracy
Accuracy
0.888
0.887
0.886
0.885
0.884
0.883
0.882
Training Accuracy
Validation Accuracy
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Epochs

Observations:

- Both Training and Validation, Loss and Accuracy curves are very similar to what we saw earlier for RNN model
- Training and Validation Loss curves are almost same as that of RNN Model, but some difference is seen for Training vs Validation Accuracy.
- For Training Accuracy, in RNN we it reaching its high in 2nd epoch and then slightly decreasing towards the end, while in LSTM it is slightly increases from 86.65% to 88.74%
- For Validation Accuracy also, in RNN we see a straight line and then a slight decrease towards the end, while in LSTM increases slightly from 88.77% to 88.85%
- Maximum Training Accuracy is 88.74%, and Maximum Validation Accuracy is 88.85%.
- Since the training and validation loss and accuracy are pretty close, which means model is not overfitting.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = simple_lstm_model.evaluate(X_test_reshaped[:100], y_test_reshaped[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 0.8728080987930298
Test Accuracy: 0.9013472199440002
```

Simple LSTM model without Embeddings, results in a slightly higher accuracy of ~90%

Loss here is better at 87%, but still high.

7. Improving Model Performance

Here are some approaches to enhance the performance of the RNN and LSTM models:

Hyperparameter Tuning:

- Model Complexity: Tune the number of layers and units in each layer to balance model capacity and avoid overfitting.
- Adjust Learning Rate: Experiment with different learning rates and optimizers. A learning rate scheduler can help in finding the optimal rate.
- Batch Size and Epochs: Modify the batch size and number of epochs to find the best training setup. Sometimes, increasing batch size or adding more epochs can improve performance.

Regularization Techniques:

- Dropout: Experiment with dropout percentage to prevent overfitting by randomly dropping neurons during training.
- L2 Regularization: Apply L2 regularization to the weights to reduce overfitting by penalizing large weights.

Advanced Model Architectures:

- **Add Embeddings:** Use pre-trained word embeddings (e.g., GloVe, Word2Vec) to capture semantic relationships between words. Embeddings can significantly improve model performance by providing richer word representations.
- **Use Bidirectional Layers:** Bidirectional RNNs and LSTMs process the sequence in both forward and backward directions, capturing context from both ends of the sequence.
- **Use Encoder-Decoder Model:** For machine translation tasks, encoder-decoder architectures with attention mechanisms can capture complex dependencies between input and output sequences by enabling the model to focus on relevant parts of the input sequence when generating each word in the output. This can improve translation accuracy.
- **Explore Pre-trained Models:** Utilize pre-trained models and fine-tune them for the specific task. Transfer learning can leverage the knowledge from large-scale datasets.

In Summary, improving model performance involves a combination of better data cleaning and pre-processing, more sophisticated and complex model architectures, and careful tuning of hyperparameters. By implementing these strategies, we can enhance the accuracy and robustness of any sequence-to-sequence models.

8. Conclusion (Milestone 1)

For Milestone 1, we have focussed on setting up the foundational aspects of the project, including data acquisition, cleaning, preprocessing, and the initial base model development using RNN and LSTM architectures.

A Basic RNN model was implemented as a baseline to understand the sequential nature of the data. The model was trained and tested on the preprocessed data, with performance metrics recorded to establish a benchmark.

Following the RNN model, an LSTM model was also implemented to address the vanishing gradient problem often encountered with RNNs. The LSTM model's performance was evaluated and compared with the RNN model.

Conclusion: Both models show reasonable performance, with the LSTM model slightly outperforming the RNN model, indicating better performance in capturing complex patterns and long-range dependencies in the sequences.

Moving forward, **Milestone 2** will focus on experimenting with advanced variants of both RNN and LSTM architectures. The goal will be to refine the translation accuracy and address any remaining challenges in the translation task

Milestone 2

Experimenting with Advanced Model Architectures

As part of Milestone-2, we will further experiment with below advanced model architectures

- RNN and LSTM Model with Embeddings
- Bi-Directional RNN and LSTM Models
- Encoder-Decoder RNN & LSTM Models
- Pre-trained Transformer Models

9. RNN and LSTM Model with Embeddings

Using Word Embeddings have several advantages –

Feature	Without Embeddings	With Embeddings
Word Representation	One-hot, sparse	Dense, semantic
Dimensionality	High (equal to vocab size)	Low (configurable, e.g., 50-300)
Handling of Rare Words	Poor	Better generalization
Capturing Word Relationships	Limited	Semantic and syntactic understanding
Efficiency and Memory Usage	High dimensional, less efficient	Reduced dimensionality, more efficient
Generalization	Limited to training data	Transfer learning possible
Contextual Understanding	Basic	Captures context and meaning
Training Stability	May suffer from gradient issues	Improved gradients, better convergence

RNN Model (with Embeddings)

We will build an RNN Model with

- Input Layer
- Embedding Layer (with dense vector size of 128 dim)
- SimpleRNN layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

We also reshape the input to be of same length as output

Model Summary

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 193, 128)	4,050,688
simple_rnn_1 (SimpleRNN)	(None, 193, 64)	12,352
dropout_1 (Dropout)	(None, 193, 64)	0
dense_1 (Dense)	(None, 193, 57414)	3,731,910

Total params: 7,794,950 (29.74 MB)
Trainable params: 7,794,950 (29.74 MB)
Non-trainable params: 0 (0.00 B)

Model Training

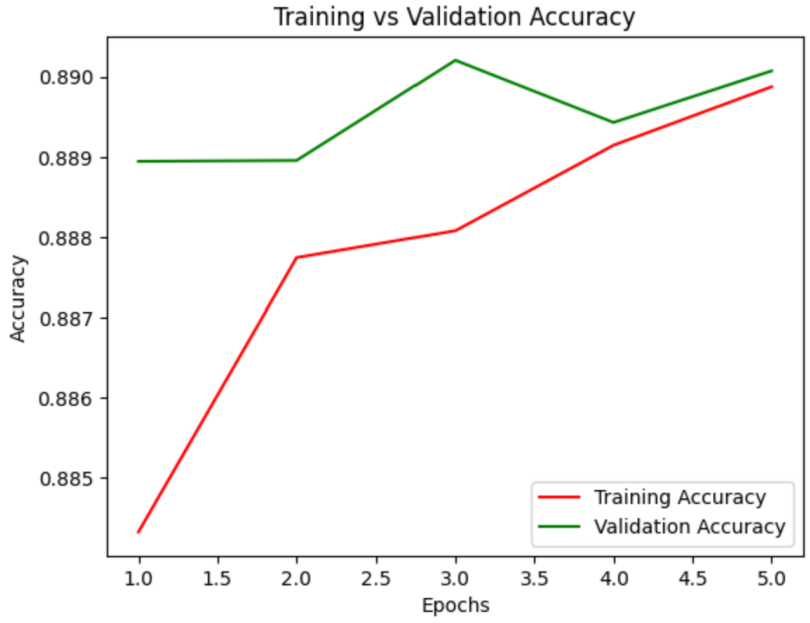
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_embed_rnn_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

embed_rnn_history = embed_rnn.fit(X_train_reshaped, y_train, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 95s 256ms/step - accuracy: 0.8665 - loss: 4.8125 - val_accuracy: 0.8889 - val_loss: 0.9588
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 82s 244ms/step - accuracy: 0.8866 - loss: 0.9783 - val_accuracy: 0.8890 - val_loss: 0.9463
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 86s 255ms/step - accuracy: 0.8876 - loss: 0.9521 - val_accuracy: 0.8902 - val_loss: 0.9326
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 86s 255ms/step - accuracy: 0.8884 - loss: 0.9370 - val_accuracy: 0.8894 - val_loss: 0.9280
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 87s 256ms/step - accuracy: 0.8905 - loss: 0.9142 - val_accuracy: 0.8901 - val_loss: 0.9243
```


Training vs Validation Loss
Training Loss
Validation Loss
Loss
Epochs
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Training vs Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs
0.890
0.889
0.888
0.887
0.886
0.885
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Observations:

- **Training Accuracy:** The accuracy started at **86.65%** in Epoch 1 and increased to **89.05%** by Epoch 5. This indicates that the model is learning and improving its performance on the training data.
- **Training Loss:** The loss decreased from **4.81** in the first epoch to **0.91** in the last epoch. A reduction in loss suggests that the model's predictions are becoming more aligned with the actual target values as training progresses.
- **Validation Accuracy:** The validation accuracy is fairly stable, starting at **88.89%** in Epoch 1 and finishing at **89.01%** in Epoch 5. While it doesn't fluctuate much, the stability suggests that the model generalizes well and isn't overfitting.
- **Validation Loss:** The validation loss shows a consistent decrease, from **0.96** to **0.92**, though the changes are small. This trend suggests that the model is slowly improving its performance on unseen data, which is a positive indicator.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = embed_rnn_model.evaluate(X_test_reshaped[:100], y_test[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 0.9261179566383362
Test Accuracy: 0.8926424384117126
```

RNN model with Embeddings, results in a very good accuracy of 89.26%. A test loss of **0.926** suggests the model is making decent predictions, but there is still room for improvement in reducing errors

LSTM Model (with Embeddings)

Now, we build a Base LSTM Model without Embeddings, with

- Input Layer
- Embedding Layer (with dense vector size of 128 dim)
- LSTM layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

Here also, we use the same reshaped input data

Model Summary

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 193, 128)	4,050,688
lstm (LSTM)	(None, 193, 64)	49,408
dropout_2 (Dropout)	(None, 193, 64)	0
dense_2 (Dense)	(None, 193, 57414)	3,731,910

Total params: 7,832,006 (29.88 MB)
Trainable params: 7,832,006 (29.88 MB)
Non-trainable params: 0 (0.00 B)

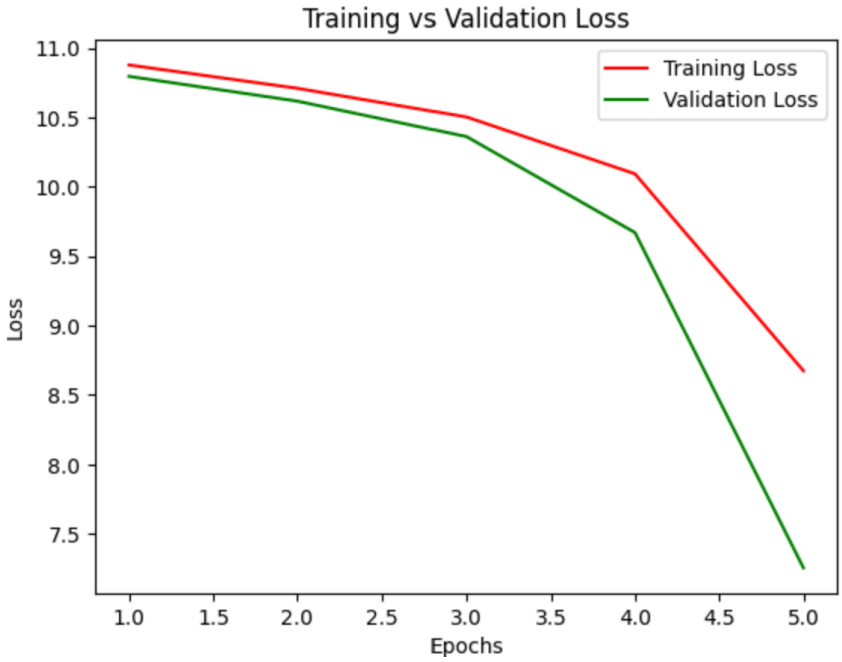
Model Training

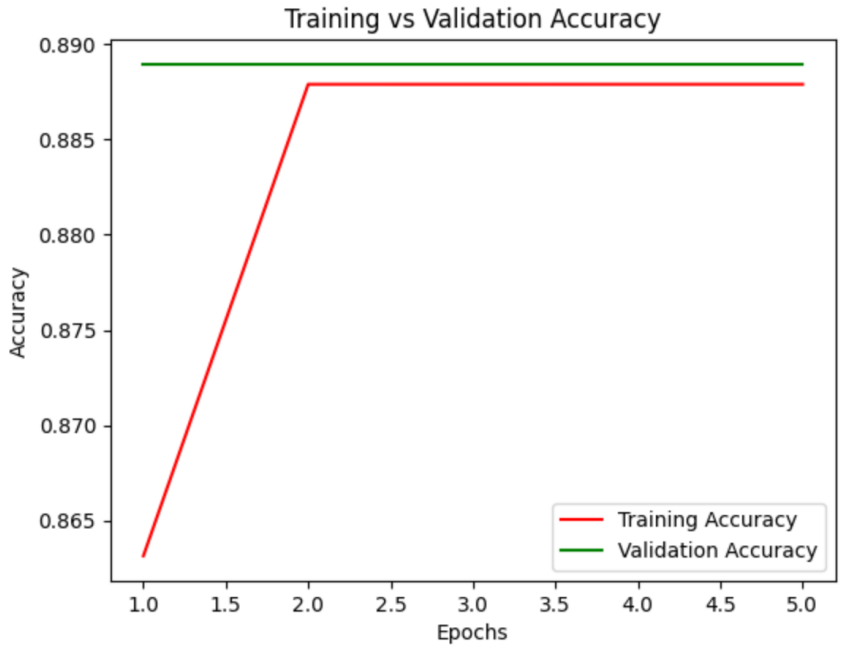
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_embed_lstm_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

embed_lstm_history = embed_lstm.fit(X_train_reshaped, y_train, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 538ms/step - accuracy: 0.7767 - loss: 10.9179 - val_accuracy: 0.8889 - val_loss: 10.7959
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 184s 545ms/step - accuracy: 0.8875 - loss: 10.7536 - val_accuracy: 0.8889 - val_loss: 10.6172
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 184s 545ms/step - accuracy: 0.8876 - loss: 10.5635 - val_accuracy: 0.8889 - val_loss: 10.3619
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 184s 546ms/step - accuracy: 0.8878 - loss: 10.2471 - val_accuracy: 0.8889 - val_loss: 9.6702
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 547ms/step - accuracy: 0.8880 - loss: 9.2345 - val_accuracy: 0.8889 - val_loss: 7.2548
```





Observations:

- **Training Accuracy**: The training accuracy starts at **77.67%** in Epoch 1 and quickly improves to **88.80%** by Epoch 5. This shows that the model is learning over time and performing better on the training set.
- **Validation Accuracy**: The validation accuracy is fixed at **89.89%** throughout all epochs. This indicates that while the model is performing well on the validation set, it isn't improving or deteriorating during the training process. This could be a sign that the validation set is relatively easy or that the model isn't generalizing better despite improving on the training set.

- **Training Loss**: The training loss starts at **10.92** and decreases steadily to **9.23** by the last epoch. While this trend indicates that the model is improving, the final loss is still quite high, which suggests that the model's predictions are still far from the actual targets.
- **Validation Loss**: The validation loss starts at **10.80** and decreases gradually to **7.25** by Epoch 5. While it's good to see validation loss decreasing, the values are still quite high. Typically, loss values should be much lower, especially if accuracy is nearing 90%.

- **Validation Accuracy Stagnation**: The fact that validation accuracy stays fixed at 88.89% suggests either a data-related issue (e.g., the validation set may not be representative) or a saturation point has been reached early on. The model might not be learning new patterns beyond the first epoch.
- **High Loss Values**: The unusually high loss values despite reasonably high accuracy indicate that the model might be learning the overall structure but still making errors on specific finer details (e.g., small differences in sequences in a translation task). The model might also be struggling with overconfidence, producing predictions that are far off from the target, contributing to the high loss.
- **Training vs. Validation Gap**: The training accuracy and validation accuracy are quite close (around 88%), which is a good sign that the model isn't overfitting. However, the relatively high loss on both training and validation sets implies the model could benefit from further tuning,

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = embed_lstm_model.evaluate(X_test_reshaped[:100], y_test[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 10.795449256896973
Test Accuracy: 0.8914507031440735
```

LSTM model with Embeddings, results in a similar accuracy of **~89.14%,** meaning the model is correctly translating about 89% of the input sequences. However, accuracy in translation can be somewhat misleading because many translations can be "almost right" but not count as correct under strict accuracy metrics.

But unlike the RNN model, the loss here is quite high at **10.795,** which suggests that the predictions are still not close to the target translations. This could mean that while the model may get many of the words right, it is struggling with certain nuances such as word order, grammar, or specific vocabulary.

10. Bi-Directional RNN and LSTM Models

Using Bi-directional RNN models provides several advantages over unidirectional models.

Standard RNNs and LSTMs process sequences in only one direction, typically from left to right. This means they only have access to previous words in a sentence, which can limit their ability to understand the full context.

While Bi-Directional models process input sequences in both directions — forward (left to right) and backward (right to left). This allows them to consider not only the past words but also future words when predicting a word, allowing them to capture a more comprehensive understanding of the context, which is crucial for language translation where the target language word order or structure may differ significantly from the source language.

Bi-Directional RNN Model

Now, we will build an RNN Model with

- Input Layer
- Embedding Layer (with dense vector size of 128 dim)
- Bi-directional SimpleRNN layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

SGD optimizer is used instead of Adam.

We also reshape the input to be of same length as output

Model Summary

Layer (type)	Output Shape	Param #
embedding_4 (Embedding)	(None, 193, 128)	4,050,688
bidirectional_1 (Bidirectional)	(None, 193, 128)	24,704
dropout_4 (Dropout)	(None, 193, 128)	0
dense_4 (Dense)	(None, 193, 57414)	7,406,406

Total params: 11,481,798 (43.80 MB)
Trainable params: 11,481,798 (43.80 MB)
Non-trainable params: 0 (0.00 B)

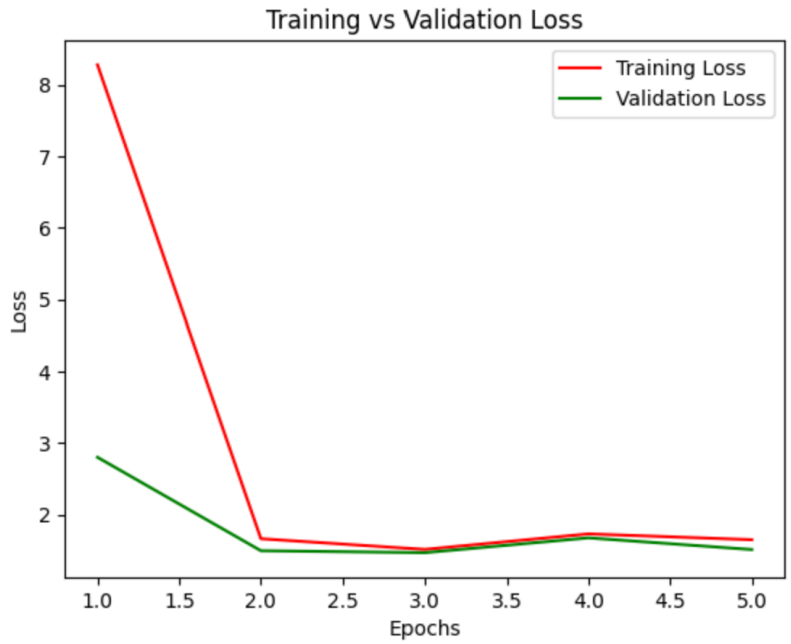
Model Training

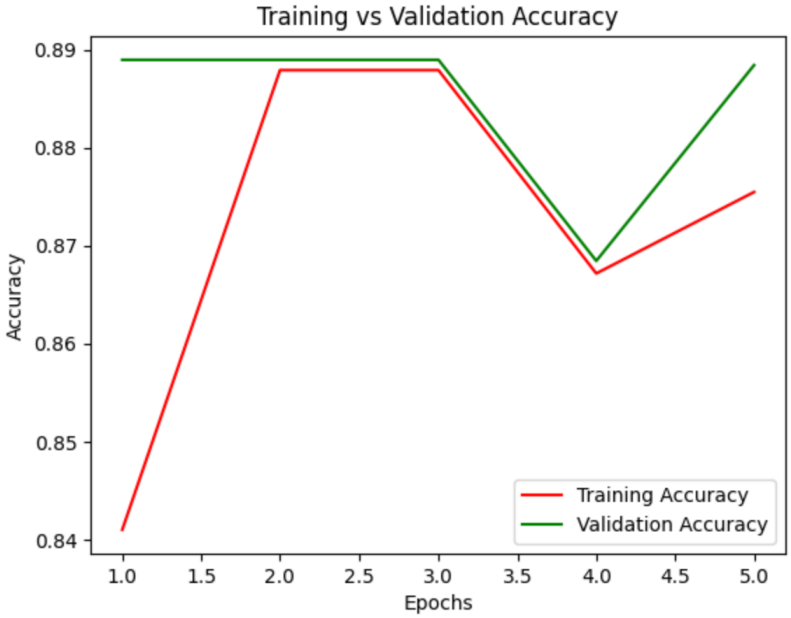
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_bidirec_rnn_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

bidirec_rnn_history = bidirec_rnn.fit(X_train_reshaped, y_train, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 134s 386ms/step - accuracy: 0.7056 - loss: 10.0793 - val_accuracy: 0.8889 - val_loss: 2.7987
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 117s 347ms/step - accuracy: 0.8885 - loss: 1.8911 - val_accuracy: 0.8889 - val_loss: 1.4918
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 114s 338ms/step - accuracy: 0.8868 - loss: 1.5318 - val_accuracy: 0.8889 - val_loss: 1.4648
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 115s 340ms/step - accuracy: 0.8759 - loss: 1.6296 - val_accuracy: 0.8684 - val_loss: 1.6717
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 114s 338ms/step - accuracy: 0.8716 - loss: 1.6715 - val_accuracy: 0.8884 - val_loss: 1.5084
```





Observations:

- **Training Accuracy:** The training accuracy starts at **70.56%** in Epoch 1 and improves to **87.16%** by Epoch 5. This shows that the model is learning and improving its ability to predict.
- **Validation Accuracy:** The validation accuracy is quite high, starting at **88.89%** in Epoch 1, and stays consistent till Epoch 3, after which there's a slight dip to **86.84%** in Epoch 4. Overall consistency suggests that the model is not overfitting, but the lack of significant improvement in validation accuracy also indicates that the model might have reached a performance plateau early on.
- **Training Loss:** The training loss starts at **10.08** in Epoch 1, which is quite high, but drops quickly to **1.89** in Epoch 2. From there, it fluctuates slightly, reaching **1.67** by Epoch 5. While the loss generally decreases, it doesn't drop as consistently after Epoch 2, and there's even a slight increase in later epochs.
- **Validation Loss:** The validation loss starts at **2.8** in Epoch 1, which is considerably lower than the training loss. It improves to **1.49** by Epoch 2 but remains relatively flat, with minor fluctuations after that. Interestingly, the validation loss also increases in Epoch 4 to **1.67** before slightly improving to **1.50** in Epoch 5.
- **Validation Accuracy and Loss Stability:** Model does not seem to be generalizing well after the first few epochs. The slight increase in validation loss in later epochs suggests that the model might be starting to overfit or that it is struggling with generalizing to new data.
- **Training vs. Validation Loss Discrepancy:** Initially, the training loss is much higher than the validation loss (in Epoch 1, 10.0793 vs 2.7987), which could be due to differences in data distribution between the training and validation sets or due to model updating its weights more efficiently on the validation set, perhaps because of factors like batch size differences or other hyperparameters.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = bidirec_rnn_model.evaluate(X_test_reshaped[:100], y_test[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 2.775775671005249
Test Accuracy: 0.8914507031440735
```

Bi-directional RNN model also results in a very good accuracy of **89.14%**, but test loss of **2.77** is on higher side, leaving room for improvement.

Bi-Directional LSTM Model

Now, we will build an LSTM Model with

- Input Layer
- Embedding Layer (with dense vector size of 128 dim)
- Bi-directional LSTM layer with 64 units to process the input sequence.
- Dropout Layer with 20% dropout
- Dense Layer

SGD optimizer is used instead of Adam.

We also reshape the input to be of same length as output

Model Summary

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 193, 128)	4,050,688
bidirectional_2 (Bidirectional)	(None, 193, 128)	98,816
dropout_5 (Dropout)	(None, 193, 128)	0
dense_5 (Dense)	(None, 193, 57414)	7,406,406

Total params: 11,555,910 (44.08 MB)
Trainable params: 11,555,910 (44.08 MB)
Non-trainable params: 0 (0.00 B)

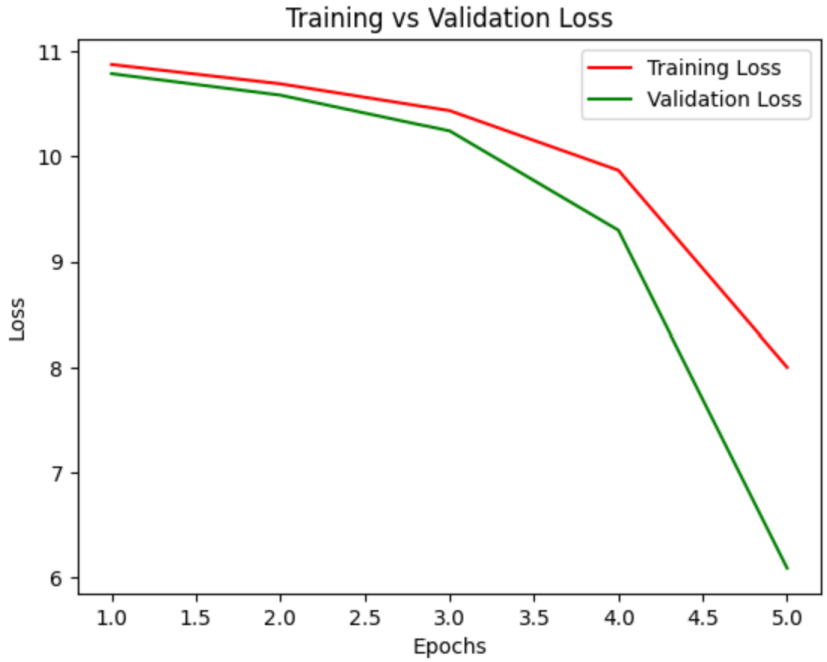
Model Training

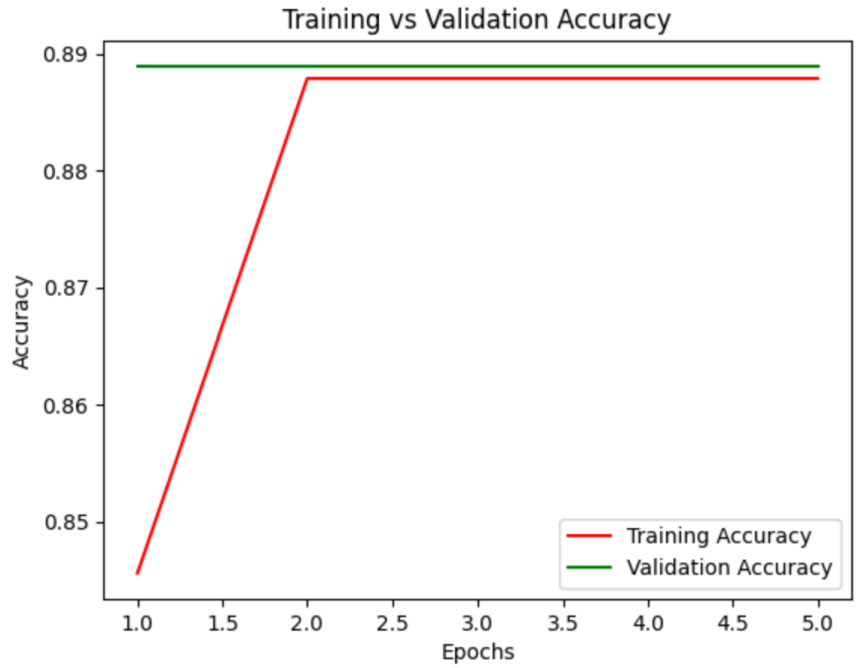
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_bidirec_lstm_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

bidirec_lstm_history = bidirec_lstm.fit(X_train_reshaped, y_train, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 224s 660ms/step - accuracy: 0.7192 - loss: 10.9166 - val_accuracy: 0.8889 - val_loss: 10.7864
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 225s 666ms/step - accuracy: 0.8878 - loss: 10.7398 - val_accuracy: 0.8889 - val_loss: 10.5827
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 225s 666ms/step - accuracy: 0.8878 - loss: 10.5144 - val_accuracy: 0.8889 - val_loss: 10.2423
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 225s 667ms/step - accuracy: 0.8885 - loss: 10.0795 - val_accuracy: 0.8889 - val_loss: 9.2978
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 226s 669ms/step - accuracy: 0.8877 - loss: 8.7361 - val_accuracy: 0.8889 - val_loss: 6.0918
```





Observations:

- **Training Accuracy:** The training accuracy starts at **71.92%** in Epoch 1 and improves to **88.77%** by Epoch 5. This shows that the model is learning and improving its ability to predict.
- **Validation Accuracy:** The validation accuracy stays constant at **88.89%** across all epochs. This suggests that while the model is performing well on the validation set, it's not improving or learning anything new after the first epoch. It could indicate that the model is saturating early and struggling to generalize better.

- **Training Loss:** The training loss starts at **10.91** in Epoch 1 and steadily decreases to **8.73** by Epoch 5. A consistent decrease in loss shows that the model is learning and getting better at minimizing errors on the training data. However, the values are quite high, which indicates that the model is still making significant errors.
- **Validation Loss:** The validation loss starts at **10.78** in Epoch 1 and drops to **6.09** by Epoch 5. This is a significant decrease, suggesting that the model is making progress in reducing errors on unseen data. However, this does not correspond to an increase in validation accuracy, which remains fixed at **88.89%**, which might indicate that the model is learning to make more "partial" improvements (i.e., getting closer to the correct outputs without being exactly right).
- **High Loss Values:** Both the training and validation losses are quite high, particularly in the early epochs. Even by Epoch 5, the loss values remain in the range of **6-9**, which indicates that the model is still making large errors. This could be due to overconfidence in incorrect predictions or challenges with certain difficult samples.
- **Training vs. Validation Loss Discrepancy:** The training loss decreases faster than the validation loss, and by the final epoch, there is a significant gap between training loss (**8.7361**) and validation loss (**6.0918**). This suggests that the model is overfitting slightly to the training data—getting better at the on the training set but not significantly improving on the validation set.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = bidirec_lstm_model.evaluate(X_test_reshaped[:100], y_test[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 10.78596305847168
Test Accuracy: 0.8914507031440735
```

Bi-directional LSTM model also results in a very good accuracy of **89.14%**, but similar to previously seen loss in LSTM Model with Embedding, the loss here is also quite high at **10.78**, which again suggests that the predictions are still not close to the target translations

11. Encoder-Decoder Model

The encoder-decoder model architecture is highly effective for language translation tasks due to its ability to handle variable-length sequences, capture long-term dependencies, and produce translations based on full-context representations.

When combined with attention mechanisms, it can focus on relevant parts of the input sentence, leading to higher translation quality. Additionally, the encoder-decoder design is scalable, flexible, and memory-efficient, making it an ideal solution for real-world translation challenges.

The encoder is responsible for encoding the input sequence into a context representation, which is later used by the decoder to generate the output sequence.

We will build an Encoder-Decoder Model using LSTM

An **Encoder** consisting of

- Input Layer (where the sequence length will be dynamically determined based on the actual input).
- Embedding Layer (with dense vector size of 128 dim)
- LSTM layer with 64 units (with return_state=True flag which indicates that the LSTM will return its internal states - state_h for the hidden state and state_c for the cell state.)
- Encoder Output Layer: executes the LSTM model and returns the output at each time step as well as the hidden and cell states. These states will be used to initialize the decoder LSTM, providing the context from the input sequence.

A **Decoder** consisting of

- Input Layer (allowing for sequences of unknown length)
- Embedding Layer (with dense vector size of 128 dim)
- LSTM layer with 64 units, with return_sequences=True to ensure that the output is a sequence and return_state=True ensures that the LSTM returns its states.
- Decoder Output Layer: executes the LSTM model taking the hidden and cell states from the encoder as the initial states, allowing the decoder to start decoding with the context of the input sequence, and returns the sequence of output hidden states at each time step
- Dense Layer: defines a fully connected layer with the size of the target language's vocabulary
- Decoder Output Layer: The dense layer is applied to the decoder's output sequence to get a probability distribution over the vocabulary for each time step. This gives the predicted translation by providing the most likely words at each step of the output sequence.

Model Summary

Layer (type)	Output Shape	Param #	Connected to
input_layer_8 (InputLayer)	(None, None)	0	-
input_layer_9 (InputLayer)	(None, None)	0	-
embedding_8 (Embedding)	(None, None, 128)	4,050,688	input_layer_8[0][0]
embedding_9 (Embedding)	(None, None, 128)	7,348,992	input_layer_9[0][0]
lstm_4 (LSTM)	[(None, 64), (None, 64), (None, 64)]	49,408	embedding_8[0][0]
lstm_5 (LSTM)	[(None, None, 64), (None, 64), (None, 64)]	49,408	embedding_9[0][0], lstm_4[0][1], lstm_4[0][2]
dense_7 (Dense)	(None, None, 57414)	3,731,910	lstm_5[0][0]

Total params: 15,230,406 (58.10 MB)
Trainable params: 15,230,406 (58.10 MB)
Non-trainable params: 0 (0.00 B)

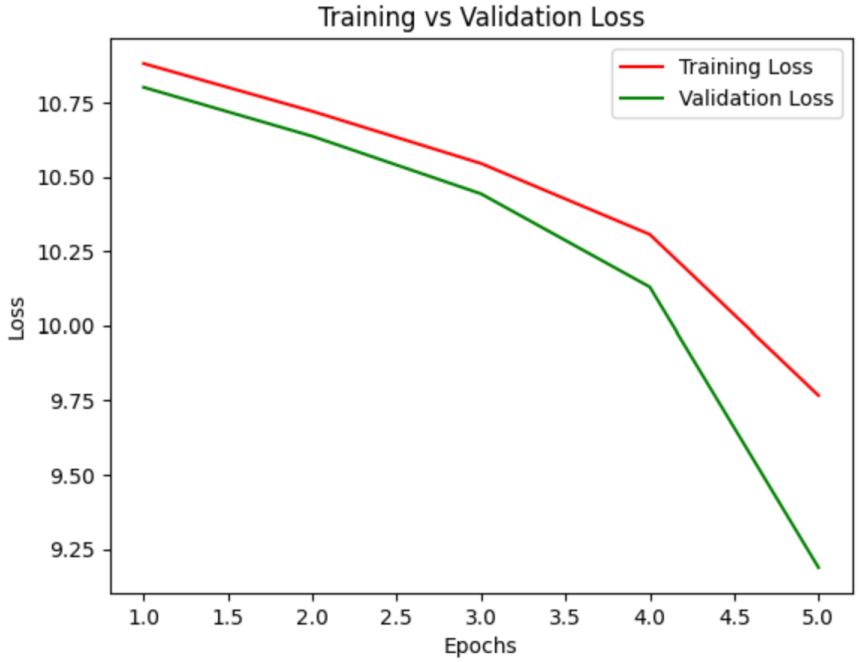
Model Training

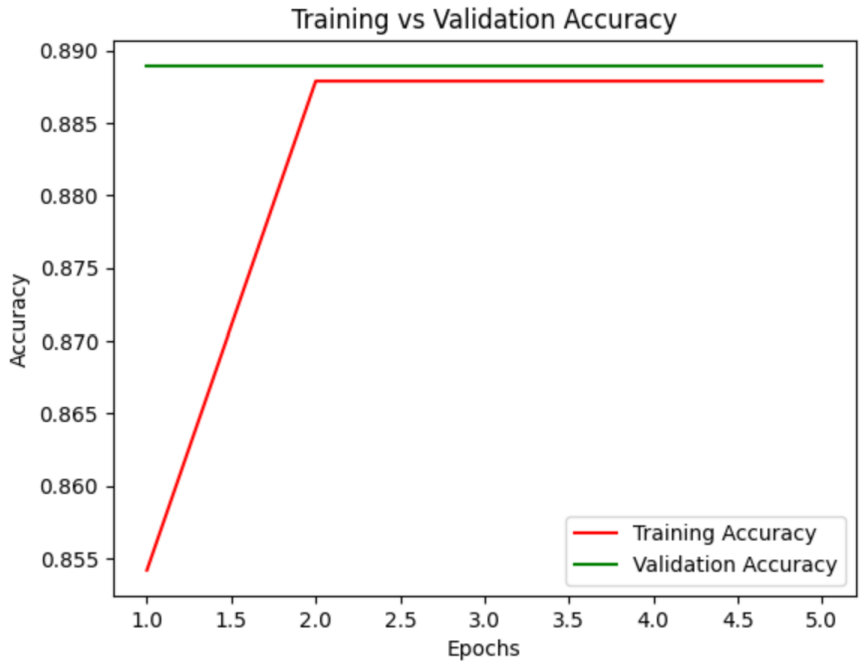
Model training is done with validation split of 20%, 5 epochs and a batch size of 64.

We also create a checkpoint to save the model with max validation accuracy

```
# Train the model
filename = 'mt_enc_dec_lstm_model.keras'
checkpoint = ModelCheckpoint(filename, monitor='val_accuracy', mode='max', save_best_only=True)   # Create checkpoint to save the model with best validation accuracy

enc_dec_lstm_history = enc_dec_lstm.fit([X_train, y_train], y_train, validation_split=0.2, epochs=5, batch_size=64, callbacks=[checkpoint], verbose=1)
```

```
Epoch 1/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 181s 531ms/step - accuracy: 0.7458 - loss: 10.9206 - val_accuracy: 0.8889 - val_loss: 10.8013
Epoch 2/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 547ms/step - accuracy: 0.8873 - loss: 10.7613 - val_accuracy: 0.8889 - val_loss: 10.6365
Epoch 3/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 547ms/step - accuracy: 0.8883 - loss: 10.5917 - val_accuracy: 0.8889 - val_loss: 10.4428
Epoch 4/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 546ms/step - accuracy: 0.8881 - loss: 10.3792 - val_accuracy: 0.8889 - val_loss: 10.1294
Epoch 5/5
338/338 ━━━━━━━━━━━━━━━━━━━━ 185s 547ms/step - accuracy: 0.8885 - loss: 9.9747 - val_accuracy: 0.8889 - val_loss: 9.1876
```





Observations:

- **Training Accuracy:** The training accuracy starts at **74.58%** in Epoch 1 and jumps to **88.85%** in Epoch 2 itself, after which remains flat, which means model has reached its learning limit early on and isn't learning new patterns after the initial epoch.
- **Validation Accuracy:** The validation accuracy is constant at **88.89%** across all epochs. While the model is achieving high accuracy on the validation set, the fact that it remains unchanged suggests that again the model isn't learning new patterns after the initial epoch.

- **Training Loss:** The training loss starts at **10.92** in Epoch 1 and decreases steadily to **9.97** by Epoch 5. Although the loss is decreasing, these values are relatively high, indicating that the model is still making substantial errors in its predictions and is still far from making good predictions.
- **Validation Loss:** The validation loss starts at **10.80** in Epoch 1 and drops to **9.18** by Epoch 5. This consistent decrease indicates that the model is gradually improving its performance on the validation data, however like the training loss the values are still relatively high, indicating that there are significant errors in the predictions.
- **Training vs. Validation Loss:** The gap between training and validation loss is not too large, which suggests that overfitting is not a major issue. However, both losses are still high, suggesting that the model might be struggling with the complexity of the task.

Model Evaluation

We can now evaluate the model on Test Data. To avoid crashing, we only use first 100 records from Test data for evaluation.

```
# Evaluate the model on 100 samples from Test data
loss, accuracy = enc_dec_lstm_model.evaluate([X_test[:100], y_test[:100]], y_test[:100], verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 10.800890922546387
Test Accuracy: 0.8914507031440735
```

Encoder-Decoder LSTM model also results in a very good accuracy of **89.14%**, but the loss here is quite high at **10.78**. The high loss despite relatively good accuracy suggests that the model is making small, incremental errors that add up.

In sequence-based models like encoder-decoder LSTMs, loss is often calculated over the entire sequence, so even if many individual tokens are correct, a few major mistakes can lead to high loss values. This pattern of high loss with high accuracy is common in sequence generation tasks where the model might miss smaller details, such as word order or specific grammar rules.

Model Prediction and Translation

We also used Encoder-Decoder Model for making predictions on sample test data and then used the predicted probabilities to translate to German sentence.

For this, we first defined inference models for both Encoder and Decoder. During inference the process differs from training, because at inference time the model generates the output step-by-step rather than having access to the entire target sequence.

- The **encoder inference model** takes a source sentence and outputs the encoded context (internal states), which will be used by the decoder to generate the translation.
- The **decoder inference model** starts with a "start" token and the encoder's states, predicts the next word and updated hidden and cell states, and then use the predicted word and updated states to predict the following word, and so on. This process continues until the model predicts an "end" token or reaches the maximum sequence length.

Then we defined a function which takes an input sentence (in the source language, like English) and generates its translation into the target language (like German).

In the function -

- First, the input sentence is tokenized and padded.
- Then the encoder inference model is called to encode the tokenized and padded sequence and get its internal states.
- Then the decoder inference model is called to start generating the translation one word at a time, using its previous predictions and the states.
- The process continues until an <END> token is produced or the maximum sequence length is reached.
- The final translation is returned.

Testing the model on a sentence from the dataset

```
# Test the model
test_sentence = 'iron cement is a ready for use paste which is laid as a fillet by putty knife or finger in the mould edges (corners) of the steel ingot mould'
translated_sentence = translate(test_sentence)
```

```
print('Translated Sentence:', translated_sentence)
```

```
Translated Sentence:  atommeilers atommeilers gezockt durchsetzen opfern oshakati finanzhilfen nachteil meßinstrument kreditwürdige finanzhilfen mackenziestrand eisenbahnstrecken zunge
```

Only part of the translated sentence is visible here, entire sentence has 193 tokens (being the max length of German Sentence used by Model)

With Encoder-Decoder model, we are able to perform English to German translation, though the translation is not accurate.

12. Model Performance Comparison

Below table compares different RNN and LSTM models based on training, validation, and test score

Model	Type	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss	Test Accuracy	Test Loss
RNN	**without Embeddings**	88.62	0.9641	88.81	0.9547	89	0.9517
LSTM	**without Embeddings**	88.74	0.9491	88.85	0.9514	90.13	0.8728
RNN	**with Embeddings**	89.05	0.9142	89.01	0.9243	89.26	0.9261
LSTM	**with Embeddings**	88.8	9.2345	88.89	7.2548	89.14	10.7954
RNN	**Bi-directional**	88.68	1.5318	88.89	1.4648	89.14	2.7757
LSTM	**Bi-directional**	88.77	8.7361	88.89	6.0918	89.14	10.7859
LSTM	**Encoder-Decoder**	88.85	9.9747	88.89	9.1876	89.14	10.8008

Summary -

- **RNN without embeddings:** Good accuracy, but the test loss is slightly higher than the LSTM without embeddings.
- **LSTM without embeddings:** LSTM performs better than RNN in both accuracy and loss, making it more effective for handling sequential data.
- **RNN with embeddings:** Marginally better test accuracy and lower loss compared to RNN without embeddings, indicating that embeddings enhance the model's performance.
- **LSTM with embeddings:** Surprisingly, despite having similar accuracy, the LSTM with embeddings has a very high loss (10.79), suggesting some inefficiency or overfitting in the model. Further tuning may be required.
- **Bi-directional RNN:** Performs better than basic models in handling more complex sequences, with lower test loss compared to basic RNN.
- **Bi-directional LSTM:** Similar accuracy to Bi-directional RNN but much higher loss. It may indicate inefficiency in handling sequence lengths or specific dataset characteristics.
- **Encoder-Decoder LSTM:** Slightly better test accuracy, but similar high loss like Bi-directional LSTM, showing that despite its complex structure, the encoder-decoder architecture is not improving loss significantly.

RNN with Embeddings is best in terms of overall Training and Validation Accuracy as well as Loss.

LSTM without embeddings is close second on Train data but is best in performance on Test data.

Considerations -

We see loss values can vary widely depending on the task complexity, dataset, and whether sequences are of variable lengths.

It’s important to understand and consider the nuances of accuracy and loss of the models.

- **Loss**: Loss is a measure of how far the model’s predicted translations are from the actual translations. Loss is a more granular metric than accuracy, as it captures how close the predicted sequence is to the target, even when the entire sequence isn't a perfect match. Therefore, loss can decrease even if accuracy remains unchanged.

  A high loss value like **10.8** (seen for encoder-decoder model), suggests that while the model is achieving decent accuracy, the predictions are still not close to the target translations. This could mean that while the model may get some of the words right, it is struggling with certain nuances such as word order, grammar, or specific vocabulary.

- **Accuracy:** Accuracy here typically refers to the proportion of the model's predictions that are correct However, it can be somewhat misleading and generally a difficult metric to optimize since multiple valid translations of a sentence may exist.

  That means based on its learned patterns, if model is able to predict a German word for a given English word, it will be considered as accurate, even though it may not be the exact word or even correct synonym.

If the model produces translations that are grammatically correct but phrased differently than the target (e.g., using synonyms), accuracy may remain high, but the loss will still penalize these differences.

Improving Model Performance –

To improve model performance, use techniques such as:

- **Teacher forcing** during training (feeding correct outputs back into the model) can help with learning sequence structure.
- **Regularization techniques** like dropout can prevent overconfidence and reduce large errors.
- **Fine-Tune the Model**: Adjust the hyperparameters such as the learning rate, batch size, no of epochs or optimizer used can help reduce the loss
- **Using BLEU or ROUGE scores** (rather than strict accuracy) for evaluating translation quality may provide a better understanding of how close translations are to the target, without requiring exact matches. These metrics better suited to account for correctness and language fluency in translation models, providing a better understanding of translation quality.
- Switching to more advanced architectures like **Transformer models**, which often outperform LSTMs on language translation tasks.

13. Limitations

The performance results in general as well as result of translation using Encoder-Decoder Model, highlight several limitations of using RNN and LSTM models, for complex tasks like language translation:

1. Difficulty with Long Sequences (Vanishing Gradients)

RNN Models: Standard RNNs struggle to capture long-term dependencies due to the vanishing gradient problem, which limits their ability to retain and propagate information over long sequences. While RNNs with embeddings show slight improvements, they still struggle with longer dependencies.

LSTM Models: While LSTMs were designed to address the vanishing gradient issue by using memory cells, the results show that they still face difficulties when handling very long sequences, leading to higher losses as errors compound over time. For example, LSTM models with embeddings or bi-directional LSTMs have high test loss values, suggesting that even they are not perfect for very long or complex sequences.

2. Overconfidence and Generalization Issues

Models like LSTM with embeddings show good accuracy but can sometimes become overconfident in their incorrect predictions, which can drive up loss disproportionately.

This is a common issue when dealing with complex datasets like language translation, where the diversity of sentence structures and context is vast.

3. Memory and Computational Complexity

The table shows that more complex architectures, like bi-directional LSTM and encoder-decoder models, have higher training loss and test loss, which suggests that these architectures may be inefficient or not well-suited to the current dataset.

Despite being designed specifically for sequence-to-sequence tasks like translation, LSTM encoder-decoder models still show high loss values, highlighting the difficulty of efficiently training these models, especially without significant tuning and resources.

4. Handling Context and Attention

These models rely solely on sequential processing of input data, which means they struggle to focus on the most relevant parts of the input sequence when translating longer or complex sentences.

This limitation becomes apparent in the high test loss values for LSTM-based models in the table, which suggests that the model is missing crucial context information for certain sequences. In language translation, context is crucial, and without mechanisms like attention, both RNNs and LSTMs struggle to handle this efficiently.

5. Bottleneck in Encoder-Decoder Models

These models condense the input sequence into a single fixed-size context vector before decoding.

This fixed-size representation becomes a bottleneck when dealing with long or complex sentences, resulting in model being unable to capture all the necessary information for translation.

6. Inconsistent Improvements from Embeddings

While embeddings generally improve model performance by providing dense, meaningful representations of words, the results show that LSTM with embeddings struggles more than LSTM without embeddings. This suggests that embeddings alone do not always guarantee better performance, and the way they are integrated into the architecture needs careful tuning, particularly for translation tasks where word order and context play critical roles.

Conclusion

RNNs and LSTMs have inherent limitations when dealing with complex tasks like language translation, particularly due to their difficulty in capturing long-term dependencies, handling large sequence variability, and generalizing well without overfitting. While these models have been effective for simpler tasks, the results here suggest that for more advanced applications like language translation, more sophisticated models, such as **Transformer architectures with attention mechanisms**, might be more effective in handling the challenges of long-term context, word relationships, and efficient computation.

14. Implications for Real World Applications

Based on the limitations we see, using RNN and LSTM models for language translation in real-world business applications has significant implications, especially when compared to more advanced models like Transformer-based models.

1. Accuracy and Translation Quality

While RNNs and LSTMs perform well in capturing sequential dependencies, they may struggle with long-range dependencies in sentences. This can lead to less accurate translations, especially in complex languages where word order or context from distant words is critical. LSTMs may suffice for simpler tasks, but accuracy concerns could affect user satisfaction and business outcomes.

2. Performance on Long Sentences

LSTMs mitigate some of the issues faced by vanilla RNNs when handling long sentences, but they can still face challenges with very long sequences due to vanishing gradients or difficulty in maintaining context over a long range.

3. Training Time and Computational Cost

Training RNNs and LSTMs can be computationally expensive and slow due to their sequential nature. Each step depends on the output of the previous step, making parallelization difficult. This can result in slower training times and higher operational costs.

4. Handling Multiple Languages and Dialects

LSTMs can be effective in handling different languages if trained properly. However, their performance may degrade when handling diverse language families, especially when there are significant syntactical or grammatical differences.

5. Real-time Applications and Latency

LSTMs are sequential, which can introduce latency when processing large-scale translation requests, especially in real-time applications like chatbots, customer service systems, or live captioning.

6. Scalability for Business Growth

RNNs and LSTMs may struggle with scalability as the size of the vocabulary, datasets, or translation requests increases. This could lead to increased infrastructure costs and slower response times as the business grows.

7. Maintenance and Model Updates

Maintaining and updating RNN or LSTM models can be resource-intensive, especially when they require retraining on new data to remain accurate as language evolves. LSTM models also may not easily incorporate new techniques like pre-trained language models.

Recommendations:

- For business-critical translations (e.g., legal, medical, or highly technical documents), using more advanced models like Transformers or combining LSTMs with attention mechanisms would improve accuracy. These are also more scalable and handle large vocabularies and datasets efficiently.
- Real world applications such customer interactions or live events usually require real-time translations with minimal delay and translation across a wide range of languages or dialects.

  Using more versatile models like Transformers, allow for more efficient parallelization and handle language diversity efficiently. LSTMs can still be useful in resource-constrained environments, non-time-sensitive translations or when the dataset is small.
- Adopting models that can be fine-tuned using transfer learning and are easier to update with new data would also reduce maintenance overhead and help to keep up with linguistic changes more easily.

15. Translation with Pre-trained Transformer Model

With the limitations of using RNN and LSTM models, we have used a pre-trained transformer model from HuggingFace to perform and test the translation

First, we import and initialize pre-trained transformer model from HuggingFace

```
from transformers import FSMTForConditionalGeneration, FSMTTokenizer
```

```
# Pretrained Transformer Model to use
mname = "facebook/wmt19-en-de"

# Initialize the tokenizer
tokenizer = FSMTTokenizer.from_pretrained(mname)

# Initialize the pretrained model
model = FSMTForConditionalGeneration.from_pretrained(mname)
```

Then, we define a function to take the input sentence, tokenize it, generate the output using model, and decode the output using tokenizer again.

```
def translate(inputs, model = model, tokenizer=tokenizer):
    # preprocesses a string and returns pytorch tensors
    tokens = tokenizer.encode(inputs, return_tensors="pt")

    # using tokens generate output with the model
    outputs = model.generate(tokens)

    # Decode the generated tensor and return a human-readable string
    translation = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return translation
```

Testing the model

```
# Test the translation
german = translate("iron cement is a ready for use paste which is laid as a fillet by putty knife or finger in the mould edges (corners) of the steel ingot mould")
german
```

```
'Eisenzement ist eine gebrauchsfertige Paste, die als Filet mit Spachtel oder Finger in die Kanten (Ecken) der Stahlbarren-Form gelegt wird.'
```

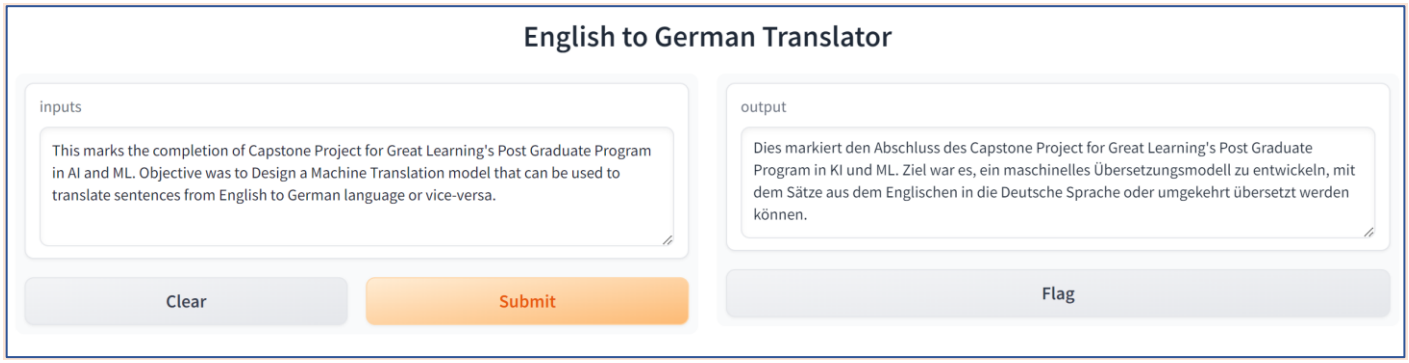
We see that the model has successfully translated the English sentence to German accurately.

16. Creating User Interface for Translation

We will now use the pre-trained transformer model created previously, to create a user interface for translation using Gradio.

```
import gradio as gr
```

```
# Define the Gradio interface using the updated syntax
iface = gr.Interface(
    fn=translate,
    inputs=[gr.Textbox(lines=4, placeholder="Enter text here...")],
    outputs="text",
    title="English to German Translator",
)

# Launch the Gradio app inside the notebook
iface.launch(share=True)
```



17. Closing Reflections

The capstone project on language translation using RNN and LSTM models has offered us valuable insights and skills, including:

1. **Understanding of Sequential Models**: project helped us to gain a good understanding of how RNN and LSTM models have ability to perform sequential tasks like language translation by capturing temporal dependencies, with LSTMs mitigating vanishing gradient issues inherent in traditional RNNs.
2. **Practical Use of Embeddings**: we learned how word embeddings transform text data into a numerical format that captures semantic meaning, improving translation quality by representing words in a more contextualized space.
3. **Importance of Model Architecture**: by experimenting with advanced models like bi-directional and encoder-decoder architectures, we realize how different architectures and enhancements can impact translation accuracy and model performance.
4. **Model Evaluation**: we gained understanding of the relationship between accuracy and loss metrics and nuances of how hyperparameter tuning and architectural adjustments impact them.
5. **Challenges in Machine Translation**: project also highlights the inherent difficulties in translating languages, such as handling long-range dependencies, context retention, and word-order differences across languages. We learnt that despite their effectiveness, RNNs and LSTMs face limitations in real-time applications due to their sequential nature, leading to longer processing times. These models also struggle with scalability in high-demand environments.
6. **Performance Trade-offs**: we gained insights into the trade-offs between model complexity, accuracy, and computational cost, especially when comparing RNN/LSTM models to more advanced architectures like Transformers.

The project equipped us with both conceptual and technical knowledge of machine translation, and providing us hands-on experience in applying this knowledge to solve real-world problems.

As language translation tasks continue to evolve, exploring advanced models like Transformer architectures offers a path towards improved efficiency, scalability, and accuracy, addressing many of the challenges observed in RNN and LSTM based systems.

The project also serves as a foundation for exploring these more advanced NLP techniques and architectures, which will drive future growth in the field of machine translation.