Faculty of Computers & Artificial Intelligence

Benha University

Selected Topics in AI – Project 1

Transformers Project

Project Name:

English-Hindi Machine Translation

Team No.: *13*

Team Members:

Ahmed Mahmoud Ismael

Ahmed Elsayed Ahmed

Mariam Yasser Hassan

Mennatu-allah Nabil Ali

Nosaiba Rafet AbdelGhany

Under Supervision of :

Eng. Maryam Mahmoud

Table of Contents

Table of Figures

Chapter 1

Problem Definition

CHAPTER 1

1. Problem Definition

1.1 Task Selection

The chosen task is Machine Translation, specifically translating text from English to Hindi. This is a vital task in Natural Language Processing (NLP) with broad applications such as cross-lingual communication, education, government services, and content localization in India, where Hindi is one of the most widely spoken languages.

1.2 Problem Definition

The problem is to develop and evaluate a machine translation system that takes English text as input and generates its accurate Hindi translation. The goal is to bridge the language barrier by producing translations that preserve the meaning, context, and fluency of the original text.

1.3 Dataset Description

For the English-to-Hindi machine translation task, we use a custom dataset hosted on the Hugging Face Hub under the namespace **damerajee/english-to-hindi-l**. This dataset is stored in **Parquet format**, which is efficient for large-scale tabular data and allows fast reading and query using tools like **Pandas**.

1.3.1 Dataset Source and Structure

- **Source:** Hugging Face Hub (https://huggingface.co/datasets/damerajee/english-to-hindi-l)
- **File:** train-00000-of-00001.parquet
- **Access Method:** Loaded via pandas.read_parquet, which reads the dataset into a Data Frame for preprocessing and analysis.
- **Format:** Apache Parquet – columnar storage optimized for performance.

1.3.2 Contents of Dataset

The dataset is assumed to be structured with at least the following key columns:

- **english_sentence** – Contains the original English sentences (source text).
- **hindi_sentence** – Contains the corresponding Hindi translations (target text).

These columns form the parallel corpus needed to train or evaluate a supervised machine translation model. A sample schema may look like:

	english_sentence	hindi_sentence
0	When it is said to him: 'Fear Allah' egotism t...	और जब उससे कहा जाता है, "अल्लाह से डर", तो अहं...
1	This profile exists already.	यह प्रोफ़ाइल पहले से ही है.
2	Halo with Ornamental Borde	विवरण: एक पारंपरिक कमल के फूल के साथ पत्थर की ...
3	and the jinn We had created before from flamin...	और हम ही ने जिन्नात को आदमी से (भी) पहले वे धु...
4	Ladies and Gentlemen, the Government of India ...	शहरीकरण की तेज गति के साथ अवसंरचना और सेवाओं क...

1-1 Dataset Sample

1.4 Evaluation Metrics

To assess the translation quality of the model, we use **BLEU**, **ROUGE-1**, **ROUGE-2**, and **ROUGE-L**. These metrics are selected to provide a balanced evaluation of both lexical overlap and structural similarity between generated and reference translations.

- **BLEU** is used for its standardization in machine translation tasks and its focus on **precision** across n-grams.
- **ROUGE-1** and **ROUGE-2** capture **recall-based** unigram and bigram overlap, useful for identifying missed content.
- **ROUGE-L** measures the **longest common subsequence**, helping evaluate fluency and sentence structure.

Using multiple metrics ensures robust evaluation from different linguistic perspectives and avoids overfitting to a single scoring criterion.

Chapter 2

Model Selection and Implementation

CHAPTER 2

2. Model Selection and Implementation

2.1 Model Selection

The model selected for this machine translation task is **Helsinki-NLP/opus-mt-en-hi**, a pretrained neural machine translation model based on the **MarianMT** architecture. This model is part of the **OPUS-MT project**, developed by the Language Technology group at the University of Helsinki. It is hosted and maintained on the Hugging Face Model Hub, making it easily accessible and integrable into modern NLP workflows.

2.2 Model Architecture

The opus-mt-en-hi model is built on **MarianMT**, a Transformer-based encoder-decoder architecture specifically optimized for neural machine translation. MarianMT is implemented in C++ with efficient Python bindings and integrates seamlessly with the Hugging Face Transformers library, enabling streamlined training and inference workflows. The architecture closely follows the original Transformer design introduced by Vaswani et al. (2017), and it supports translation across a wide range of language pairs.

This specific model configuration includes:

- **Encoder Layers:** 6
- **Decoder Layers:** 6
- **Total Trainable Parameters:** 76,381,184

Key Components:

- **Encoder:** Encodes the input English sentence into contextualized embeddings, capturing semantic and syntactic features.
- **Decoder:** Autoregressively generates the Hindi translation by attending to the encoder outputs and previously generated tokens.
- **Shared Embeddings:** Source and target language embeddings may be shared to reduce parameter count and improve translation consistency.

This balanced architecture allows the model to maintain a strong trade-off between performance and computational efficiency, making it suitable for both academic research and production deployment

2.3 Data Preprocessing

2.3.1.1.1 Dataset Quality Assessment

The original dataset comprises 1,786,788 entries. An initial data quality analysis revealed that there were no missing values in either column, indicating a complete dataset. However, 238,016 duplicate sentence pairs were identified. These duplicates were removed to ensure diversity and to reduce redundancy in training, resulting in a cleaned dataset containing 1,548,772 unique English-Hindi sentence pairs.

2.3.1.1.2 Data Cleaning

Before training and evaluating the machine translation model, a text cleaning step was applied to both the English and Hindi sentences to ensure data consistency and reduce noise. The cleaning function performs the following operations:

1. **Unicode Normalization:** All text is normalized using the Unicode **NFKC** (Normalization Form Compatibility Composition) standard, which ensures that visually identical characters have the same internal representation.
2. **Removal of Control Characters:** Control characters (e.g., formatting and non-printable characters) are removed by filtering out characters with Unicode categories starting with 'C'.
3. **Digit Removal:** All numerical digits are removed from the text using a regular expression (\d+), as numbers are often inconsistently translated or irrelevant for model learning in many general-domain translation tasks.
4. **Character Filtering:**
   The text is restricted to contain only:
   - **English alphanumeric characters**
   - **Hindi characters from the Devanagari Unicode block** (\u0900–\u097F)
   - **Basic punctuation** (commas, periods, question marks, exclamation points, semicolons, and colons)
   - **Whitespace characters**: Any other characters or symbols are removed to eliminate noise, especially from mixed-language or informal content.

Whitespace Normalization: Multiple consecutive whitespace characters are reduced to a single space, and leading/trailing spaces are trimmed.

2.3.1.1.3 Dataset Splitting

To facilitate efficient training and evaluation of the machine translation model, a subset of the full dataset was selected based on sentence length. Specifically, a new column total_length was computed after cleaning by summing the character lengths of the English and Hindi sentences in each row. This metric serves as a proxy for sentence complexity and content richness. The dataset was then sorted in descending order by total_length, and the top 30,000 entries were retained. This ensures that the model is trained and evaluated on longer, more informative sentence pairs, which can improve learning and generalization, especially during early experimentation.
From this filtered subset of 30,000 sentence pairs:

- The **training set** consists of the first 10,000 rows.
- The **validation set** is taken from rows 25,000 to 27,000.
- The **test set** includes rows 28,000 to 30,000.

2.3.1.1.4 Dataset Tokenization

To prepare the English-Hindi parallel corpus for training and inference with the Helsinki-NLP/opus-mt-en-hi model, tokenization is performed using the Hugging Face AutoTokenizer. This tokenizer is pretrained and specifically aligned with the vocabulary and subword segmentation rules of the MarianMT model architecture.

2.4 Fine Tuning Pipeline

To adapt the pretrained Helsinki-NLP/opus-mt-en-hi MarianMT model for our specific English-to-Hindi dataset, we constructed a fine-tuning pipeline comprising dataset preparation, tokenization, and formatting for the Hugging Face Trainer API.
To adapt the pretrained Helsinki-NLP/opus-mt-en-hi MarianMT model for our specific English-to-Hindi dataset, we constructed a fine-tuning pipeline comprising dataset preparation, tokenization, and formatting for the Hugging Face Trainer API.

2.4.1.1 .1 Dataset Preparation

The dataset was first sorted by a total_length metric (sum of character lengths of English and Hindi sentences) to prioritize longer and more content-rich pairs. From this sorted dataset of 50,000 sentence pairs, the data was split as follows:

- **Training Set**: 14,000 sentence pairs (rows 1,000 to 15,000)
- **Validation Set**: 4,000 sentence pairs (rows 30,000 to 34,000)
- **Test Set**: 2,000 sentence pairs (rows 38,000 to 40,000)

This split ensures a balance between training performance and validation coverage during fine-tuning.

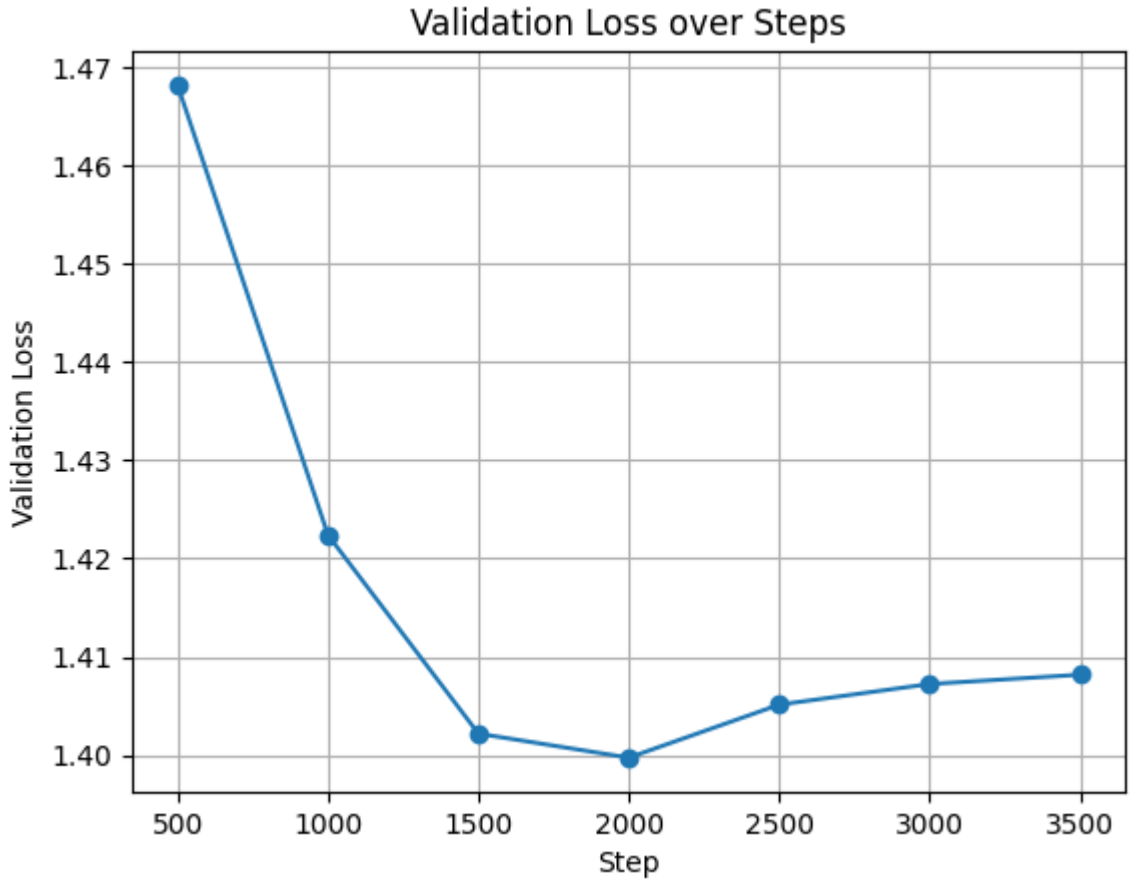
```
df['total_length'] = df['english_sentence'].str.len() + df['hindi_sentence'].str.len()
df = df.sort_values(by='total_length', ascending=False).head(50000).reset_index(drop=True)
df.drop(columns='total_length', inplace=True)

train_df = df.iloc[1000:15000]
val_df = df.iloc[30000:34000]
test_df = df.iloc[38000:40000]
```

2.4.1.1.2 Tokenization

Tokenization was handled using the AutoTokenizer class from the Hugging Face Transformers library, which loads the MarianMT-compatible tokenizer for the chosen model. This ensures alignment between the tokenization scheme and the pretrained model vocabulary.

```
model_name = "Helsinki-NLP/opus-mt-en-hi"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
```

2.5 Validation Loss Over Training Epochs



CHAPTER 3

3. Optimization

3.1 Tuning Pipeline

The fine-tuning pipeline in this project is built using the Hugging Face Transformers library with a PyTorch backend. The goal is to fine-tune a pretrained sequence-to-sequence translation model, Helsinki-NLP/opus-mt-en-hi, for English-to-Hindi translation. The model is trained and evaluated using BLEU score as the primary performance metric, with hyperparameter tuning applied to identify the best combination of learning rate, batch size, and weight decay.
The tuning was done only on 10,000 rows from the training data and 2,000 rows from the validation data for faster tuning.
Before the data preprocessing is done, the dataframes are converted to Hugging Face Dataset object.

```
train_dataset = Dataset.from_pandas(train_df)
test_dataset = Dataset.from_pandas(test_df)
val_dataset = Dataset.from_pandas(val_df)
```

3-1 Convert Dataframes to Hugging Face Dataset Objects

We used the DataCollatorForSeq2Seq in the Trainer to simplify preparing input and target sequences for training. Since sequence-to-sequence tasks involve variable-length sequences, this collator automatically pads them to the length of the longest sequence in a batch, saving memory. It also creates attention masks to ensure the model ignores padding tokens and ensures the target labels are properly padded. By using this collator, we avoid manual preprocessing and ensure the data is correctly formatted for the model to focus on learning the important parts of the sequences.

3.2 Hyperparameters Tuning

Hyperparameter tuning is conducted through a manual grid search. The tuning phase explores combinations of three hyperparameters: learning rate (5e-5, 2e-5), batch size (16,4, 8), and weight decay (0.01, 0.1). For each combination, a Seq2SeqTrainingArguments object is created, and the model is trained for five epochs. During training, BLEU scores are evaluated periodically using a custom callback or internal evaluation mechanism. The best-performing configuration is selected based on the highest BLEU score.

The Seq2SeqTrainer class from Hugging Face simplifies training and evaluation. It automatically handles the forward pass, loss calculation, optimization, evaluation, and checkpoint saving. It also supports metric tracking and best model selection based on the evaluation metric.

```python
bleu_metric = evaluate.load("bleu")
rouge_metric = evaluate.load("rouge")
def compute_metrics(eval_pred):
    preds, labels = eval_pred

    decoded_preds = tokenizer.batch_decode(preds, skip_special_tokens=True)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    decoded_preds = [pred.strip() for pred in decoded_preds]
    decoded_labels = [label.strip() for label in decoded_labels]

    bleu_result = bleu_metric.compute(
        predictions=decoded_preds,
        references=[[label] for label in decoded_labels]
    )

    rouge_result = rouge_metric.compute(
        predictions=decoded_preds,
        references=decoded_labels,
        use_stemmer=True
    )

    result = {
        "bleu": bleu_result["bleu"],
        "rouge1": rouge_result["rouge1"],
        "rouge2": rouge_result["rouge2"],
        "rougeL": rouge_result["rougeL"],
    }

    return result
```

3-2 Evaluation Metrics Computation

```python
training_args = Seq2SeqTrainingArguments(
    output_dir=None,
    learning_rate=lr,
    per_device_train_batch_size=bs,
    per_device_eval_batch_size=bs,
    weight_decay=wd,
    num_train_epochs=5,
    adam_beta1=0.95,
    max_grad_norm=0.5,
    predict_with_generate=True,
    save_steps=100,
    save_total_limit=1,
    generation_num_beams=5,
    metric_for_best_model="bleu",
    greater_is_better=True,
)
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train()
```

3-3 Hugging Face Trainer

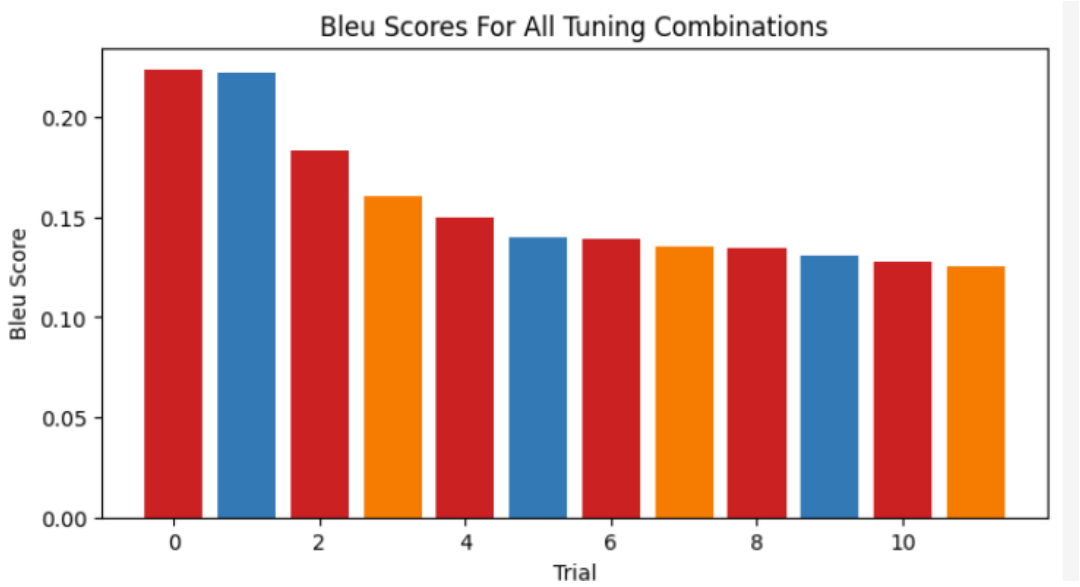
For more debugging, custom callbacks were coded to inspect memory usage and training time for each epoch, and to evaluate on regular periods.

```python
class PrintEpochCallback(TrainerCallback):
    def __init__(self):
        self.epoch_start_time = None

    def on_epoch_begin(self, args, state, control, **kwargs):
        self.epoch_start_time = time.time()
        print(f"Starting Epoch {state.epoch:.2f}")

    def on_epoch_end(self, args, state, control, **kwargs):
        if self.epoch_start_time:
            duration = time.time() - self.epoch_start_time
            print(f"Epoch {state.epoch-1:.2f} time: {duration:.2f} seconds")

class MemoryCallback(TrainerCallback):
    def on_epoch_end(self, args, state, control, **kwargs):
        allocated = torch.cuda.memory_allocated() / 1024**3
        reserved = torch.cuda.memory_reserved() / 1024**3
        print(f"[Step {state.global_step}] Allocated: {allocated:.2f} GB | Reserved: {reserved:.2f} GB")

class EvalEveryNStepsCallback(TrainerCallback):
    def __init__(self, eval_steps=500):
        self.eval_steps = eval_steps

    def on_step_end(self, args, state, control, **kwargs):
        if state.global_step % self.eval_steps == 0 and state.global_step != 0:
            control.should_evaluate = True
        if state.global_step == state.max_steps:
            control.should_evaluate = True

        return control
```

3-4 Training Custom Callbacks

Tuning was done by performing the 12 trials based on the hyperparameters combinations. As shown by the BLEU scores in the figure 3-4, the best results are for the first trial with the parameters:

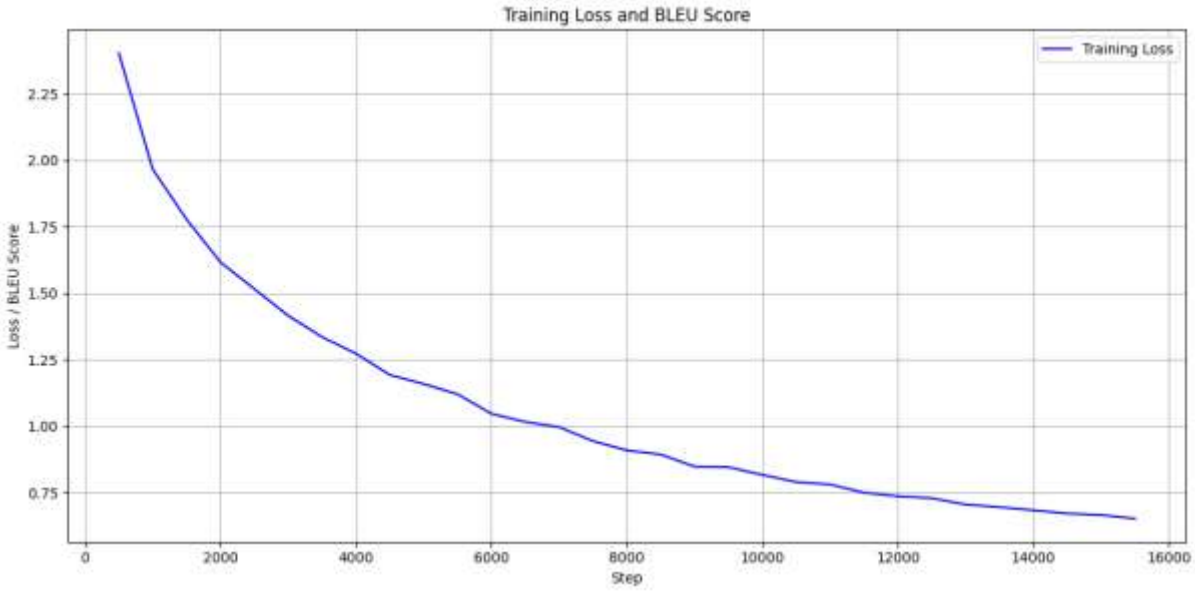
- Learning rate: 5e-05
- Batch size: 16
- Weight decay: 0.1

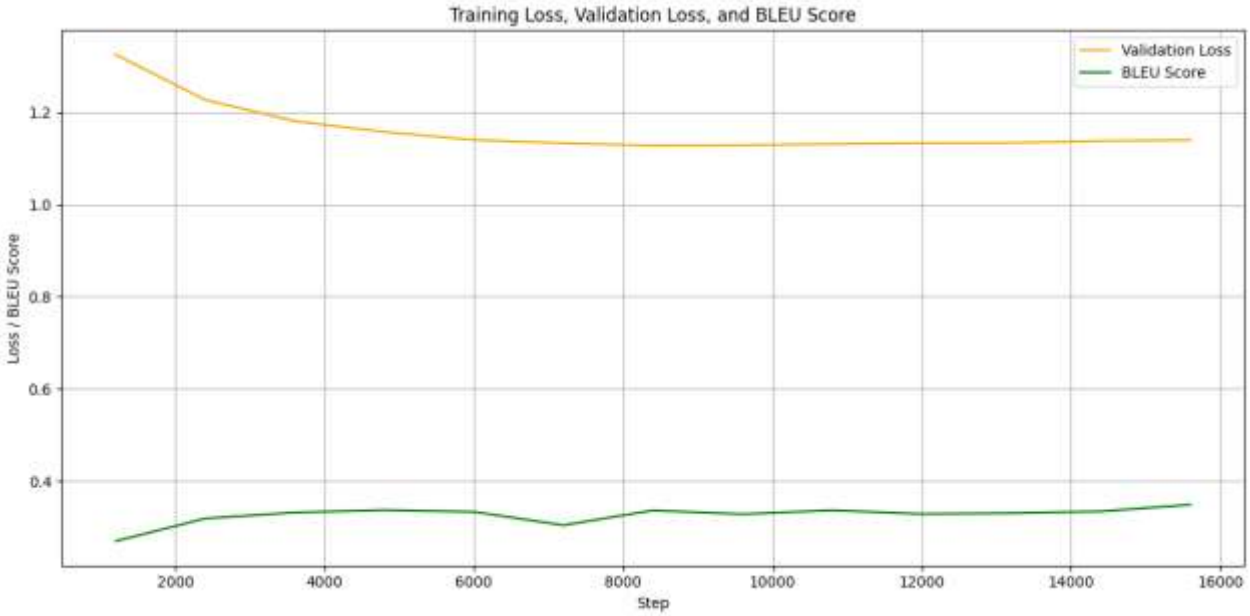


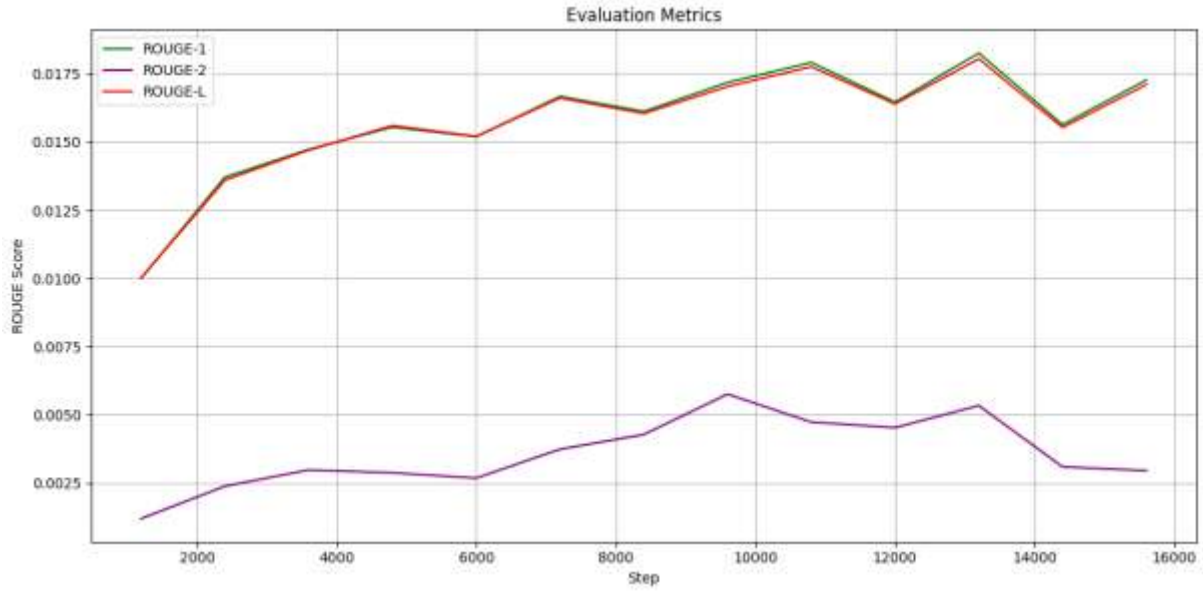
3-5 Hyperparameters Tuning BLEU Scores

3.3 Training With Best Tuning Parameters

After tuning, training was performed using the best parameters with 10 epochs on full training dataset (25,000 rows).



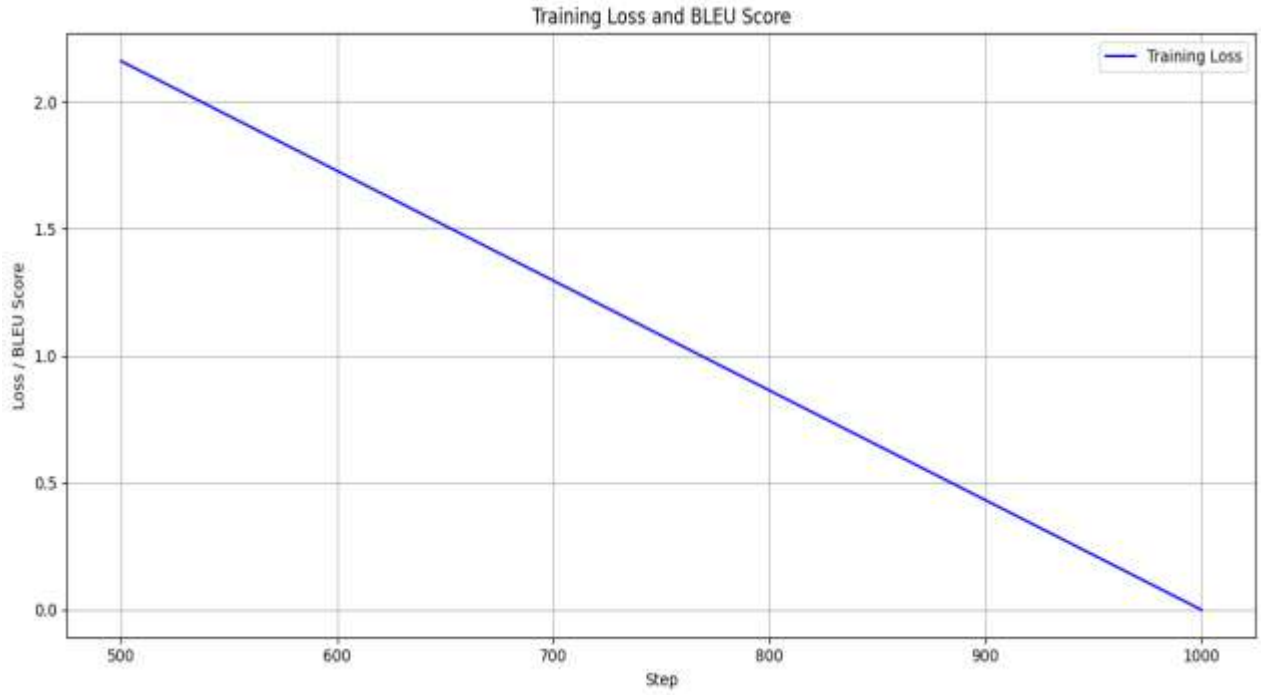




3.4 Training with best tuning parameters + Mixed precision

To improve training efficiency and reduce memory usage, Mixed Precision Training was enabled using FP16 precision by setting the fp16=True and fp16_full_eval=True flags in the Seq2SeqTrainingArguments.
This configuration allows the model to leverage 16-bit floating point (FP16) arithmetic, significantly reducing memory consumption and speeding up training time.

```
training_args = Seq2SeqTrainingArguments(
    output_dir="/kaggle/working/results/best",
    do_train=True,
    do_eval=True,
    per_device_train_batch_size=best_params['batch_size'],
    per_device_eval_batch_size=best_params['batch_size'],
    predict_with_generate=True,
    learning_rate=best_params['learning_rate'],
    max_grad_norm=0.5,
    weight_decay=best_params['weight_decay'],
    adam_beta1=0.95,
    num_train_epochs=25,
    save_steps=100,
    save_total_limit=1,
    fp16=True,
    fp16_full_eval=True,
    # deepspeed=None,
    # fsdp=None,
    # local_rank=-1,
    dataloader_num_workers=0,
    generation_num_beams=5,
    metric_for_best_model="bleu",
    greater_is_better=True,
)
```



Impact on Evaluation Metrics :

While mixed precision training provided clear advantages in terms of performance speed and GPU memory efficiency, it had a negative impact on evaluation scores in this experiment. Specifically, all automatic evaluation metrics dropped to zero:

- **ROUGE-1:** 0.0000
- **ROUGE-2:** 0.0000
- **ROUGE-L:** 0.0000
- **BLEU:** 0.0000