# Neural Machine Translation using Transformer for English to Hindi Translation

## 1. Introduction

Neural Machine Translation (NMT) has revolutionized how we translate text across different languages, utilizing deep learning models. In this report, we focus on building an English-to-Hindi translation system using the Transformer architecture. The model is trained using a parallel English-Hindi corpus, leveraging TensorFlow and Keras to implement a sequence-to-sequence learning framework. The goal is to encode an input English sentence and generate its corresponding Hindi translation.

The Transformer model, introduced by Vaswani et al. (2017), has become the state-of-the-art architecture in machine translation tasks due to its ability to handle long-range dependencies, faster training, and scalability compared to traditional models such as RNNs and LSTMs. This report details the methodology and implementation used for training a translation model on the Hindi-English dataset.

## 2. Data Preprocessing

The dataset used is a **parallel Hindi-English corpus**, which contains aligned sentences in both languages. The data preprocessing steps are critical to transforming raw sentences into a format that can be fed into the Transformer model.

### 2.1 Data Loading

The dataset was loaded using Pandas, with the Hindi and English sentences represented in respective columns:

```python
train_df = pd.read_csv("/kaggle/input/hindi-english-parallel-corpus/hindi_english_parallel.csv")

train_df.rename(columns={'hindi': 'hindi_sentence', 'english': 'english_sentence'}, inplace=True)
```

## 2.2 Data Cleaning

To optimize the learning process, sentences were filtered based on length. Sentences that were too short (<20 characters) or too long (>200 characters) were removed. This helps the model learn more effectively without being overwhelmed by extreme outliers.

## 2.3 Adding Special Tokens

Special tokens such as `<SOS>` (Start of Sentence) and `<EOS>` (End of Sentence) were added to both the Hindi and English sentences:

```python
eng = eng.apply(lambda x: "<SOS> " + str(x) + " <EOS>")
hind = hind.apply(lambda x: "<SOS> "+ str(x) + " <EOS>")
```

## 2.4 Tokenization and Padding

Both the English and Hindi sentences were tokenized using `Tokenizer` from TensorFlow, converting each word to a unique integer index. The tokenized sequences were then padded to a fixed length (`ENCODER_LEN` and `DECODER_LEN`) for consistency:

```python
inputs = tf.keras.preprocessing.sequence.pad_sequences(inputs, maxlen=ENCODER_LEN, padding='post', truncating='post')
targets = tf.keras.preprocessing.sequence.pad_sequences(targets, maxlen=DECODER_LEN, padding='post', truncating='post')
```

# 3. Model Architecture

The Transformer architecture, comprising an **Encoder** and a **Decoder**, was used for the translation task. The key components of the Transformer include Multi-Head Self-Attention, Feed Forward Networks, and Positional Encoding.

## 3.1 Encoder

The encoder consists of multiple layers of self-attention and feed-forward networks. Each layer processes the input sentence, capturing dependencies between words. Positional encodings are added to the embeddings to capture the sequence of the input.

```python
class EncoderLayer(tf.keras.layers.Layer):
    ...
    def call(self, x, training, mask):
        attn_output, _ = self.mha(x, x, x, mask)
        ...
        return out2
```

## 3.2 Decoder

The decoder, similar to the encoder, consists of multiple layers. It takes the encoder's output and translates it into the target language. The decoder uses self-attention to focus on relevant parts of the sentence being generated and cross-attention to focus on relevant parts of the encoder's output.

```python
class DecoderLayer(tf.keras.layers.Layer):
    ...
    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)
        ...
        return out3, attn_weights_block1, attn_weights_block2
```

## 3.3 Multi-Head Attention

Multi-Head Attention is a core feature of the Transformer. It allows the model to focus on different parts of the sentence simultaneously by creating attention heads that operate in parallel.

```python
class MultiHeadAttention(tf.keras.layers.Layer):
    def call(self, v, k, q, mask):
        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)
        ...
        return output, attention_weights
```

## 4. Training the Model

The training involved iterating through batches of data, optimizing the model's parameters using the Adam optimizer with a custom learning rate scheduler. The custom learning rate increases linearly during the initial steps (warmup) and then decreases proportionally to the inverse square root of the step number.

```python
learning_rate = CustomSchedule(d_model)
optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
```

### 4.1 Loss and Accuracy

The model's loss function is **Sparse Categorical Crossentropy**, ignoring padded positions (zeros). The accuracy is measured by comparing the predicted token with the actual token at each position in the sentence.

```python
def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    return tf.reduce_sum(loss_)/tf.reduce_sum(mask)
```

### 4.2 Training Process

The model was trained for 10 epochs, with checkpoints saved every 5 epochs. During each epoch, the loss and accuracy were printed to monitor performance.

~~~
```python
for epoch in range(EPOCHS):
    start = time.time()
    train_loss.reset_states()
    for (batch, (inp, tar)) in enumerate(dataset):
        train_step(inp, tar)
    print(f'Epoch {epoch + 1} Loss {train_loss.result():.4f} Accuracy {train_accuracy.result():.4f}')’’’
~~~

## 5. Evaluation and Inference

Once the model was trained, it was evaluated on test sentences by feeding an English sentence and allowing the decoder to predict the Hindi translation step by step. The model uses **beam search decoding** to predict the best sequence.

~~~
```python
def evaluate(text):
    text = eng_tokenizer.texts_to_sequences([text])
    ...
    for i in range(DECODER_LEN):
        predictions, attention_weights = transformer(
            encoder_input, output, False, enc_padding_mask, combined_mask, dec_padding_mask)
        ```
~~~

## 6. Results and Discussion

The Transformer achieved satisfactory performance on the task, with stable convergence in training loss and accuracy. Some key observations include:

- The model effectively learned translation patterns between English and Hindi, especially for common phrases and syntactic structures.

- Attention weights allowed the model to focus on the relevant parts of the English sentence while generating the Hindi translation.

- The model struggled with more complex sentences, which could be improved by increasing the dataset size or tuning hyper parameters.

## 7. Conclusion

In this report, we implemented a Neural Machine Translation model using the Transformer architecture to translate English sentences to Hindi. The model showed promising results, highlighting the power of the Transformer in sequence-to-sequence tasks. With further improvements in training data and fine-tuning, the model's performance could be enhanced for more accurate translations.

## 8. Future Work

- Hyper parameter Tuning: Experimenting with more layers, larger embedding’s, and increasing the dataset size could improve performance.

- Advanced Decoding Techniques: Implementing beam search or other decoding strategies could generate more fluent translations.

- Generalization: Applying the model to other language pairs to evaluate its versatility.

This project demonstrates the potential of the Transformer model in language translation, paving the way for more sophisticated and scalable machine translation systems.