Design and Development of English to Telugu Translation System based on Transformer with Deep Learning

Team

Jayanth Reddy G (20103010)
Akhila Banothu (20103052)

Supervised By

Dr Khelchandra Thongam
Associate Professor
Computer Science and Engineering Dept.

CONTENTS

INTRODUCTION

AIM

This project aims to create a proficient English to Telugu translation system using transformer architectures in deep learning. By bridging language gaps, it enhances communication and accessibility, offering an advanced tool for seamless translation between English and Telugu.

OBJECTIVES

1. Implement a transformer-based deep learning model for text translation.
2. Fine-tune the model to handle language-specific nuances and improve performance.
3. Implement appropriate evaluation metrics (e.g.,BLEU-score, accuracy)to quantitatively assess the translation quality.
4. Create an intuitive and user-friendly interface

MOTIVATION

Our project, aiming to create an English to Telugu transformer-based translation system, is fueled by the commitment to empower the Telugu-speaking community. By breaking language barriers, this initiative provides seamless access to global information, education, and diverse perspectives, allowing the Telugu community to express themselves on a broader stage.

This transformative tool not only translates words but preserves the cultural nuances of Telugu, fostering cross-cultural communication and understanding.

SCOPE

1. **Language-specific Challenges:** Address Telugu's linguistic nuances to ensure contextually accurate translations.
2. **User Interface Design:** Develop an intuitive interface for seamless English to Telugu text translation.
3. **Real-time Translation:** Strive for real-time translation capabilities to enhance user experience.
4. **Cross-Platform Compatibility:** Ensure the system is compatible across various platforms and devices.
5. **Scalability:** Design the system to be scalable for potential expansion to additional languages or model enhancements.

LITERATURE SURVEY

SURVEY - 1

Rule-Based Machine Translation (RBMT):

- RBMT relies on explicit linguistic rules and grammatical structures to translate text from one language to another. These rules are typically created by linguists and translation experts and may involve syntax, semantics, and morphology.
- RBMT systems heavily depend on linguistic knowledge and require extensive manual rule creation. Linguists need to encode language specific rules and translation patterns, making the process labor-intensive.

Transformer-Based Approach:

- Transformer models learn translation patterns directly from large parallel corpora without explicit rule definition. They are trained on vast amounts of data, allowing them to generalize well across various language pairs and contexts.
- Unlike RBMT, transformer-based systems follow an end-to-end learning approach. They automatically learn hierarchical representations of input sentences and generate translations without relying on predefined linguistic rules.

Reference : English to Telugu Rule based Machine Translation System: A Hybrid Approach by
Keerthi Lingam, Srujana Inturi, E. Ramalakshmi Assistant Professors Department of IT CBIT, India
International Journal of Computer Applications (0975 – 8887) Volume 101– No.2, September 2014

SURVEY - 2

LSTM (Long Short-Term Memory):

- LSTMs are based on recurrent neural networks (RNNs) and process input sequences sequentially. They maintain hidden states that capture information from previous time steps, allowing them to model temporal dependencies.
- LSTMs have hidden states and memory cells that enable them to capture and store information over long sequences. The memory cells help in mitigating the vanishing gradient problem associated with standard RNNs.

Transformer-Based Approach:

- The Transformer architecture relies on attention mechanisms, allowing it to capture relationships between words in a sequence in a parallelized manner. This attention mechanism allows each position in the input sequence to focus on different parts of the input sequence during processing.
- Transformers are highly parallelizable, making them computationally efficient. The attention mechanism allows for simultaneous processing of all positions in a sequence, enabling effective use of hardware accelerators like GPUs.

Reference : Neural Machine Translation System for English to Indian Language Translation Using MTIL Parallel Corpus by B. Premjith, M. Anand Kumar and K.P. Soman
Received February 8, 2018; previously published online March 20, 2019.

SURVEY - 3

Statistical Machine Translation (SMT):

- SMT relies on a combination of rule-based and statistical methods. It involves the creation of linguistic rules and the estimation of statistical models based on bilingual corpora.
- SMT systems require extensive feature engineering, where linguistic experts manually design features and weights to capture translation patterns, word alignments, and other linguistic phenomena.

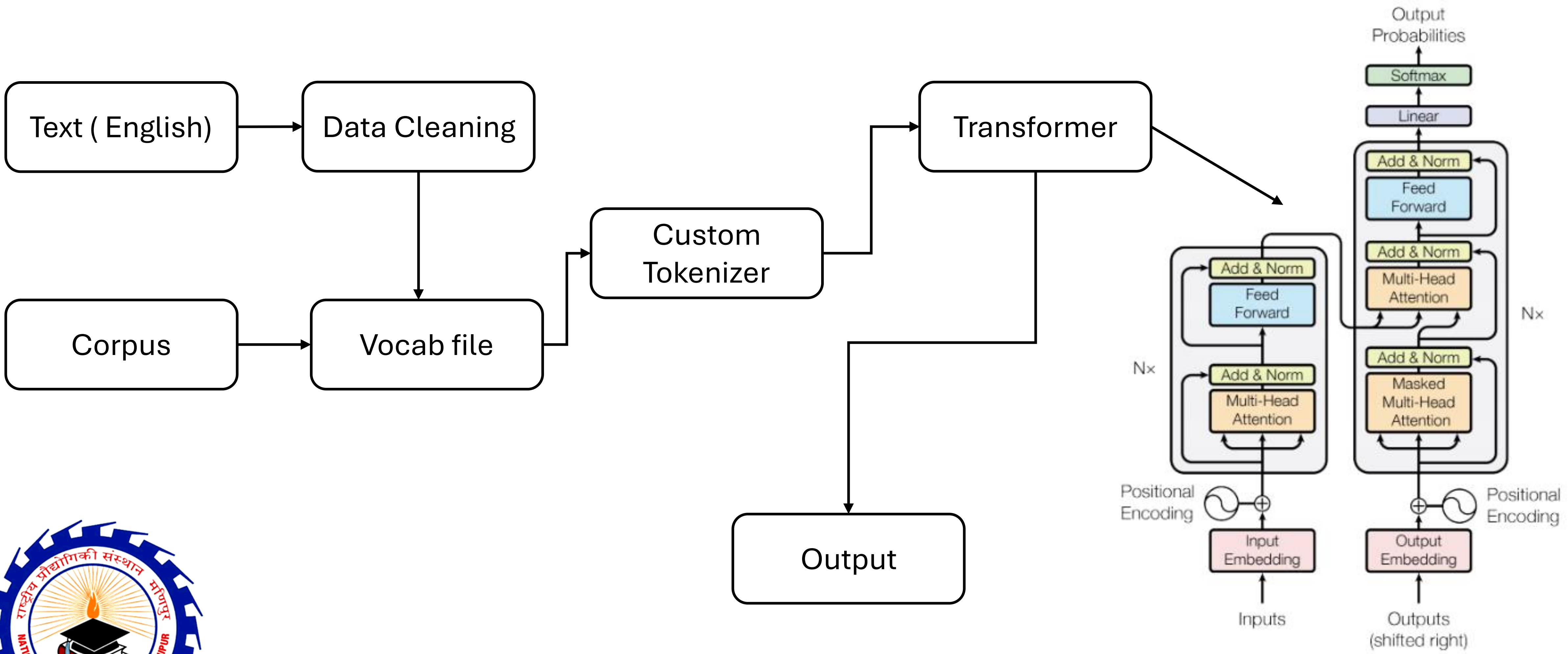
Transformer-Based Approach:

- Transformers leverage self-attention mechanisms, allowing the model to focus on different parts of the input sequence when generating each part of the output sequence. This attention mechanism captures dependencies effectively and enables the model to handle long-range dependencies.
- Transformers excel at capturing contextual understanding, as they consider the entire context of a sentence during both training and inference. This enables them to generate translations that are more contextually relevant and coherent.

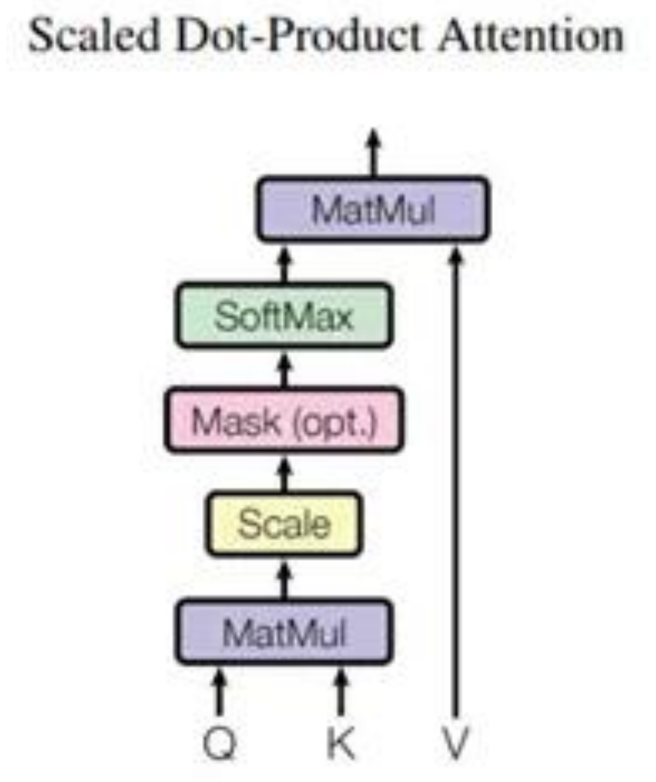
Reference : A SURVEY ON FORMAL LANGUAGE TRANSLATION (TELUGU - ENGLISH)
by Mrs. P. Swaroopa, Kalakonda Vishnu, Pulipati Akshay, S. Siva Sai Sandip, Vennam Vivek.
2022 JETIR November 2022, Volume 9, Issue 11

METHODOLOGY

ARCHITECTURE AND FLOW DIAGRAM



SCALED DOT PRODUCT ATTENTION

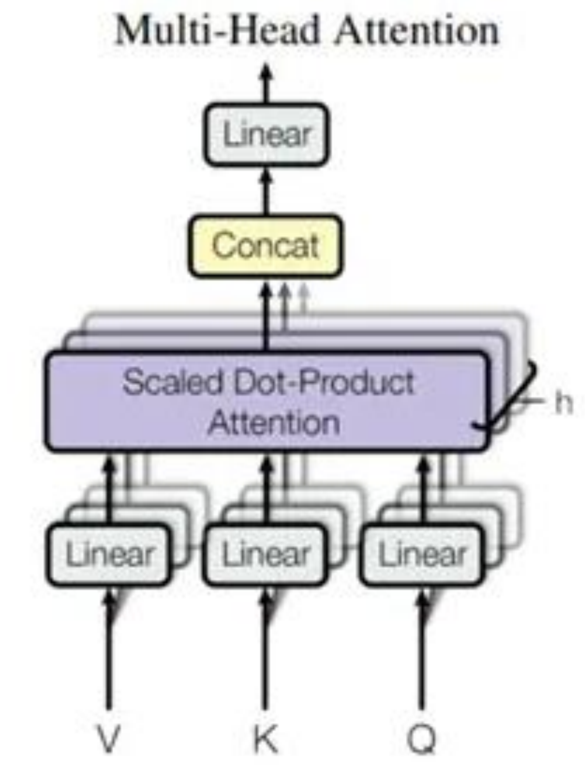


The attention function used by the transformer takes three inputs: Q (query), K (key), V (value). The equation used to calculate the attention weights is:

$$Attention(Q, K, V) = softmax_k\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Query vector: Represented by a word vector in the sequence and defines the hidden state of the decoder.
Key vector: Represented by all the words in the sequence and defines the hidden state of the encoder.
Value vector: Represented by all the words in the sequence and defines the attention weights of the encoder hidden states.
The dot-product attention is scaled by a factor of square root of the depth. This is done because for large values of depth, the dot product grows large in magnitude pushing the softmax function where it has small gradients resulting in a very hard softmax.

MULTIHEAD ATTENTION



Multi-head attention consists of four parts: **Linear layers and split into heads, Scaled dot-product attention, Concatenation of heads, Final linear layer** . Each multi-head attention block gets three inputs**; Q (query), K (key), V (value).** These are put through linear (Dense) layers and split up into multiple heads.

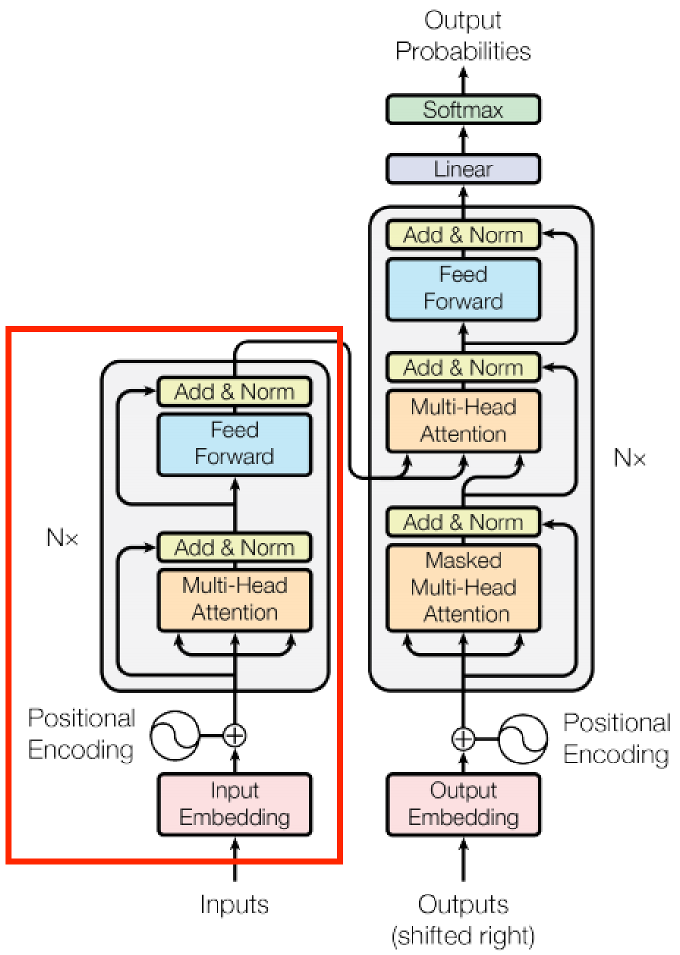
The scaled_dot_product_attention defined above is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated and put through a final Dense layer.

Instead of one single attention head, Q, K, and V are split into multiple heads because it allows the model to jointly attend to information from different representation subspaces at different positions. After the split each head has a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.

This ensures that the words we want to focus on are kept

irrelevant words are flushed out.

ENCODER

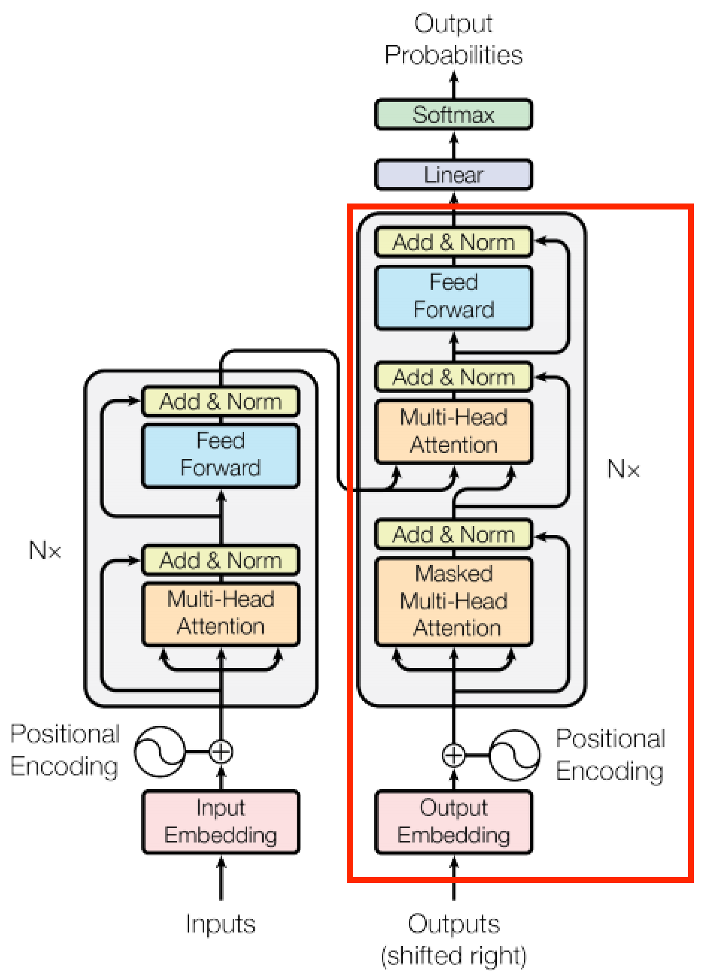


The encoder consists of:

- A PositionalEmbedding layer at the input.
- A stack of EncoderLayer layers.

The encoder receives the embedding vectors as a list of vectors, each of 512 (can be tuned as a hyper-parameter) size dimension. Both the encoder and the decoder add a positional encoding (that will be explained later) to their input. Both also use a bypass that is called a residual connection followed by an addition of the original input of the sub-layer and another normalization layer (which is also known as a batch normalization)

DECODER



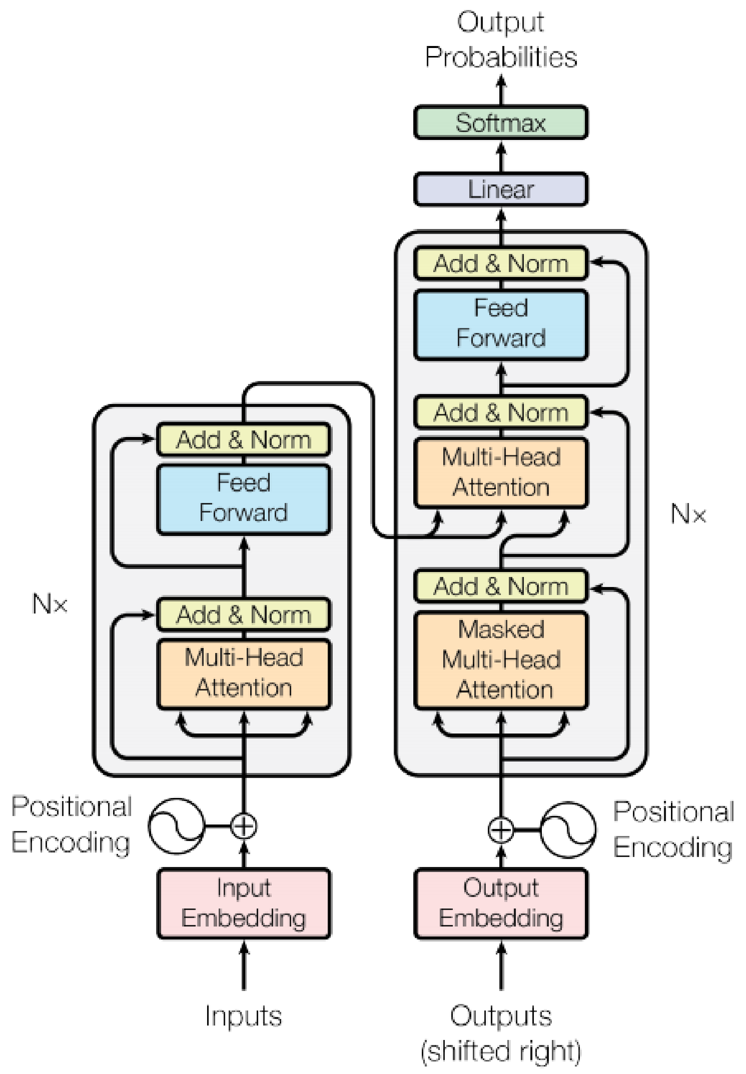
The decoder's stack is slightly more complex, with each DecoderLayer containing a CausalSelfAttention, a CrossAttention, and a FeedForward layer

Similar to the Encoder, the Decoder consists of a PositionalEmbedding, and a stack of DecoderLayers

Unlike the encoder, the decoder uses an addition to the Multi-head attention that is called masking. This operation is intended to prevent exposing posterior information from the decoder. It means that in the training level the decoder doesn't get access to tokens in the target sentence that will reveal the correct answer and will disrupt the learning procedure. It's really important part in the decoder because if we will not use the masking the model will not learn anything and will just repeat the target sentence.

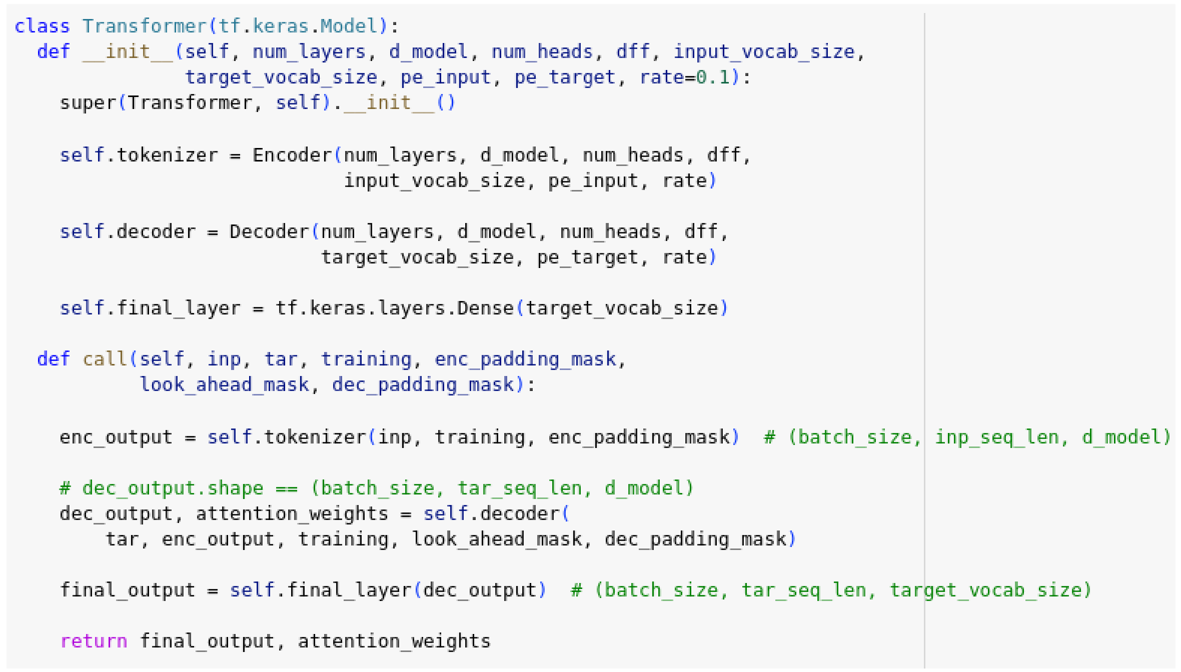
TRANSFORMER

The transformer



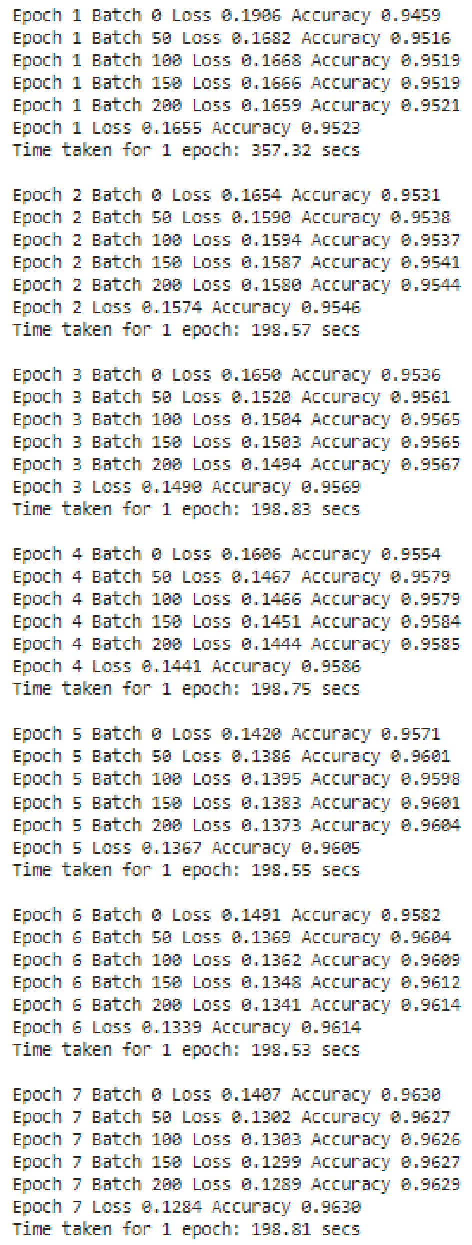
We now have Encoder and Decoder. To complete the Transformer model, we need to put them together and add a final linear (Dense) layer which converts the resulting vector at each location into output token probabilities.

The output of the decoder is the input to this final linear layer.

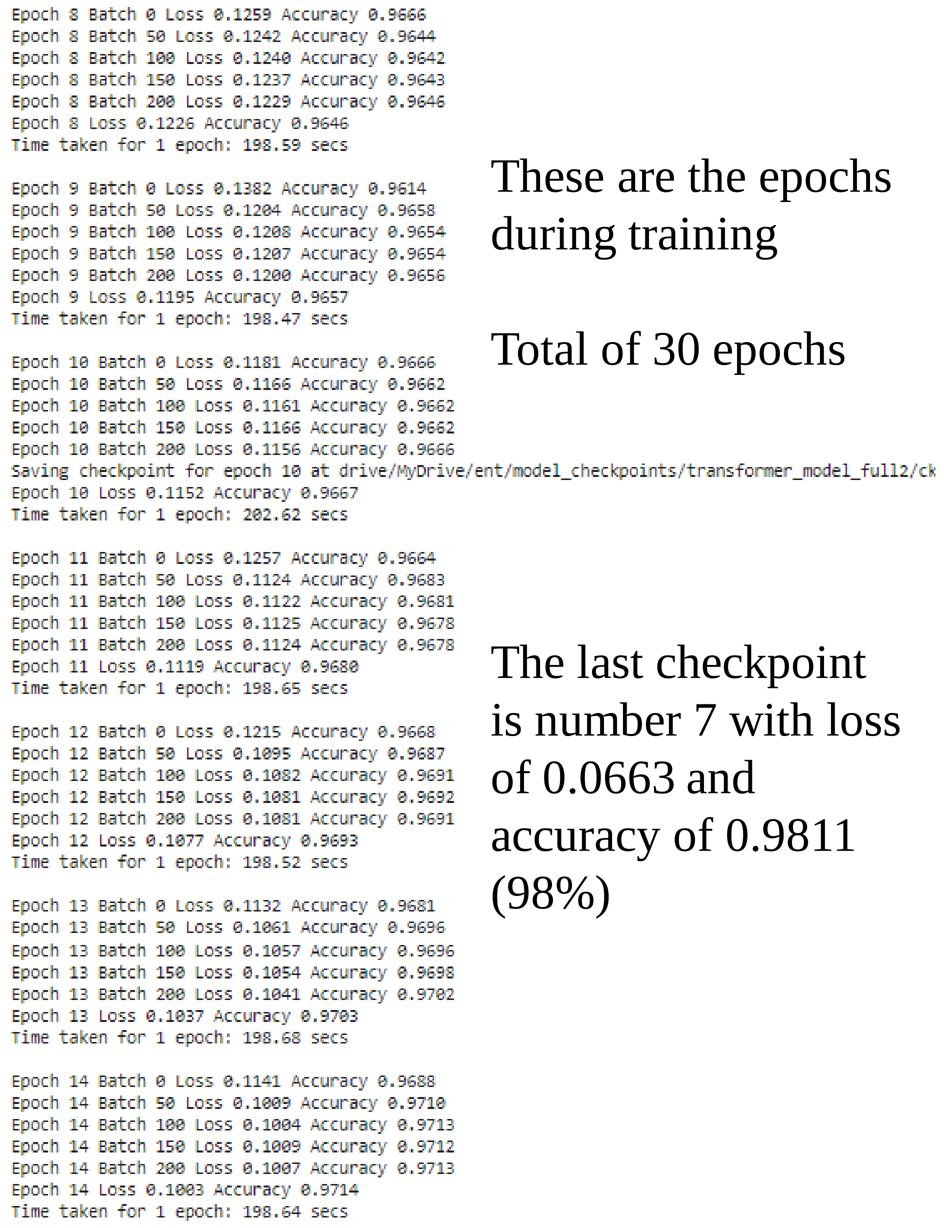


```
class Transformer(tf.keras.Model):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               target_vocab_size, pe_input, pe_target, rate=0.1):
    super(Transformer, self).__init__()

    self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                             input_vocab_size, pe_input, rate)

    self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                           target_vocab_size, pe_target, rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inp, tar, training, enc_padding_mask,
           look_ahead_mask, dec_padding_mask):

    enc_output = self.tokenizer(inp, training, enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

    # dec_output.shape == (batch_size, tar_seq_len, d_model)
    dec_output, attention_weights = self.decoder(
        tar, enc_output, training, look_ahead_mask, dec_padding_mask)

    final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, target_vocab_size)

    return final_output, attention_weights
```

TRAINING



```
Epoch 1 Batch 0 Loss 0.1906 Accuracy 0.9459
Epoch 1 Batch 50 Loss 0.1682 Accuracy 0.9516
Epoch 1 Batch 100 Loss 0.1668 Accuracy 0.9519
Epoch 1 Batch 150 Loss 0.1666 Accuracy 0.9519
Epoch 1 Batch 200 Loss 0.1659 Accuracy 0.9521
Epoch 1 Loss 0.1655 Accuracy 0.9523
Time taken for 1 epoch: 357.32 secs

Epoch 2 Batch 0 Loss 0.1654 Accuracy 0.9531
Epoch 2 Batch 50 Loss 0.1590 Accuracy 0.9538
Epoch 2 Batch 100 Loss 0.1594 Accuracy 0.9537
Epoch 2 Batch 150 Loss 0.1587 Accuracy 0.9541
Epoch 2 Batch 200 Loss 0.1580 Accuracy 0.9544
Epoch 2 Loss 0.1574 Accuracy 0.9546
Time taken for 1 epoch: 198.57 secs

Epoch 3 Batch 0 Loss 0.1650 Accuracy 0.9536
Epoch 3 Batch 50 Loss 0.1520 Accuracy 0.9561
Epoch 3 Batch 100 Loss 0.1504 Accuracy 0.9565
Epoch 3 Batch 150 Loss 0.1503 Accuracy 0.9565
Epoch 3 Batch 200 Loss 0.1494 Accuracy 0.9567
Epoch 3 Loss 0.1490 Accuracy 0.9569
Time taken for 1 epoch: 198.83 secs

Epoch 4 Batch 0 Loss 0.1606 Accuracy 0.9554
Epoch 4 Batch 50 Loss 0.1467 Accuracy 0.9579
Epoch 4 Batch 100 Loss 0.1466 Accuracy 0.9579
Epoch 4 Batch 150 Loss 0.1451 Accuracy 0.9584
Epoch 4 Batch 200 Loss 0.1444 Accuracy 0.9585
Epoch 4 Loss 0.1441 Accuracy 0.9586
Time taken for 1 epoch: 198.75 secs

Epoch 5 Batch 0 Loss 0.1420 Accuracy 0.9571
Epoch 5 Batch 50 Loss 0.1386 Accuracy 0.9601
Epoch 5 Batch 100 Loss 0.1395 Accuracy 0.9598
Epoch 5 Batch 150 Loss 0.1383 Accuracy 0.9601
Epoch 5 Batch 200 Loss 0.1373 Accuracy 0.9604
Epoch 5 Loss 0.1367 Accuracy 0.9605
Time taken for 1 epoch: 198.55 secs

Epoch 6 Batch 0 Loss 0.1491 Accuracy 0.9582
Epoch 6 Batch 50 Loss 0.1369 Accuracy 0.9604
Epoch 6 Batch 100 Loss 0.1362 Accuracy 0.9609
Epoch 6 Batch 150 Loss 0.1348 Accuracy 0.9612
Epoch 6 Batch 200 Loss 0.1341 Accuracy 0.9614
Epoch 6 Loss 0.1339 Accuracy 0.9614
Time taken for 1 epoch: 198.53 secs

Epoch 7 Batch 0 Loss 0.1407 Accuracy 0.9630
Epoch 7 Batch 50 Loss 0.1302 Accuracy 0.9627
Epoch 7 Batch 100 Loss 0.1303 Accuracy 0.9626
Epoch 7 Batch 150 Loss 0.1299 Accuracy 0.9627
Epoch 7 Batch 200 Loss 0.1289 Accuracy 0.9629
Epoch 7 Loss 0.1284 Accuracy 0.9630
Time taken for 1 epoch: 198.81 secs
```

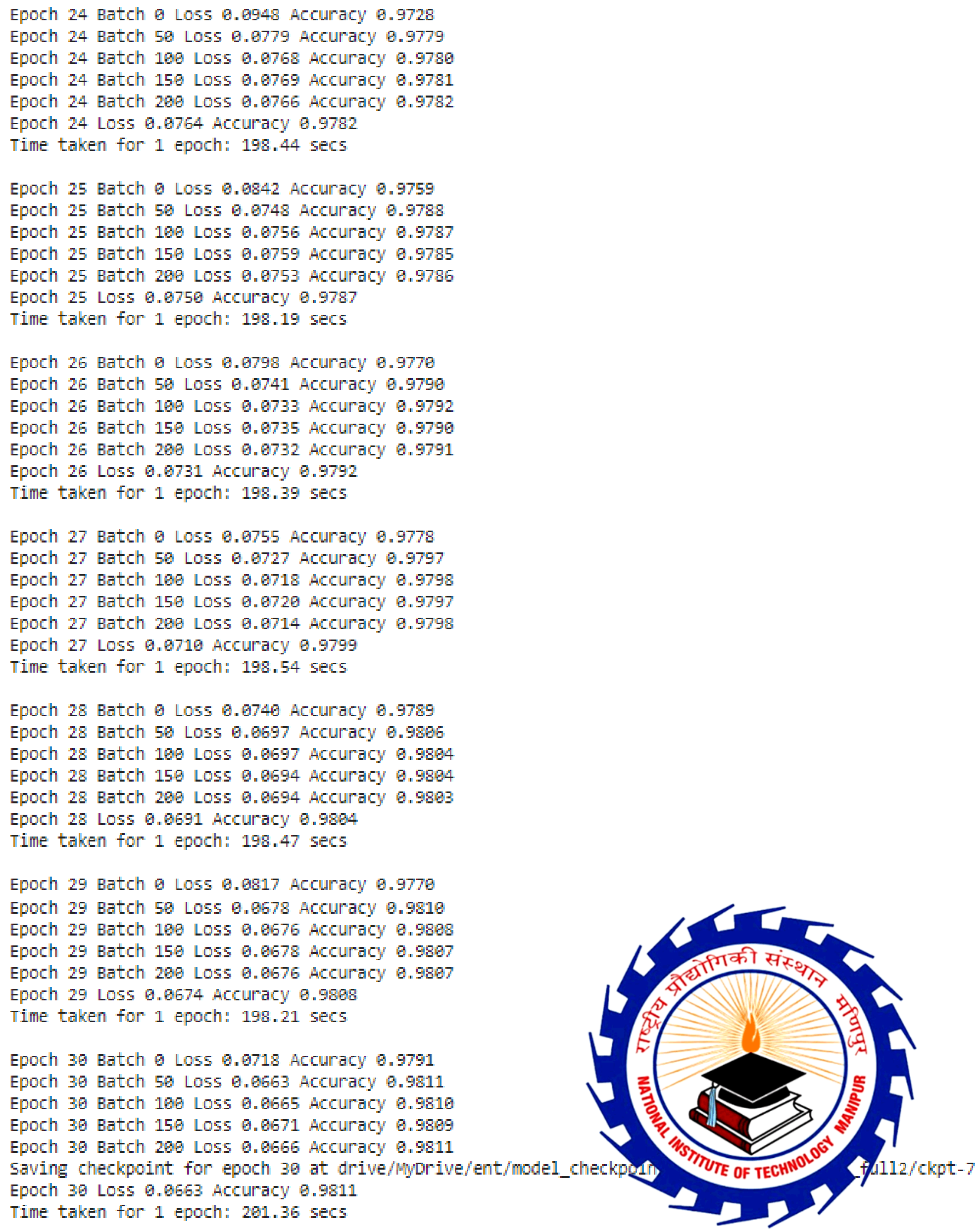


```
Epoch 8 Batch 0 Loss 0.1259 Accuracy 0.9666
Epoch 8 Batch 50 Loss 0.1242 Accuracy 0.9644
Epoch 8 Batch 100 Loss 0.1240 Accuracy 0.9642
Epoch 8 Batch 150 Loss 0.1237 Accuracy 0.9643
Epoch 8 Batch 200 Loss 0.1229 Accuracy 0.9646
Epoch 8 Loss 0.1226 Accuracy 0.9646
Time taken for 1 epoch: 198.59 secs

Epoch 9 Batch 0 Loss 0.1382 Accuracy 0.9614
Epoch 9 Batch 50 Loss 0.1204 Accuracy 0.9658
Epoch 9 Batch 100 Loss 0.1208 Accuracy 0.9654
Epoch 9 Batch 150 Loss 0.1207 Accuracy 0.9654
Epoch 9 Batch 200 Loss 0.1200 Accuracy 0.9656
Epoch 9 Loss 0.1195 Accuracy 0.9657
Time taken for 1 epoch: 198.47 secs

Epoch 10 Batch 0 Loss 0.1181 Accuracy 0.9666
Epoch 10 Batch 50 Loss 0.1166 Accuracy 0.9662
Epoch 10 Batch 100 Loss 0.1161 Accuracy 0.9662
Epoch 10 Batch 150 Loss 0.1166 Accuracy 0.9662
Epoch 10 Batch 200 Loss 0.1156 Accuracy 0.9666
Saving checkpoint for epoch 10 at drive/MyDrive/ent/model_checkpoints/transformer_model_full2/ck
Epoch 10 Loss 0.1152 Accuracy 0.9667
Time taken for 1 epoch: 202.62 secs

Epoch 11 Batch 0 Loss 0.1257 Accuracy 0.9664
Epoch 11 Batch 50 Loss 0.1124 Accuracy 0.9683
Epoch 11 Batch 100 Loss 0.1122 Accuracy 0.9681
Epoch 11 Batch 150 Loss 0.1125 Accuracy 0.9678
Epoch 11 Batch 200 Loss 0.1124 Accuracy 0.9678
Epoch 11 Loss 0.1119 Accuracy 0.9680
Time taken for 1 epoch: 198.65 secs

Epoch 12 Batch 0 Loss 0.1215 Accuracy 0.9668
Epoch 12 Batch 50 Loss 0.1095 Accuracy 0.9687
Epoch 12 Batch 100 Loss 0.1082 Accuracy 0.9691
Epoch 12 Batch 150 Loss 0.1081 Accuracy 0.9692
Epoch 12 Batch 200 Loss 0.1081 Accuracy 0.9691
Epoch 12 Loss 0.1077 Accuracy 0.9693
Time taken for 1 epoch: 198.52 secs

Epoch 13 Batch 0 Loss 0.1132 Accuracy 0.9681
Epoch 13 Batch 50 Loss 0.1061 Accuracy 0.9696
Epoch 13 Batch 100 Loss 0.1057 Accuracy 0.9696
Epoch 13 Batch 150 Loss 0.1054 Accuracy 0.9698
Epoch 13 Batch 200 Loss 0.1041 Accuracy 0.9702
Epoch 13 Loss 0.1037 Accuracy 0.9703
Time taken for 1 epoch: 198.68 secs

Epoch 14 Batch 0 Loss 0.1141 Accuracy 0.9688
Epoch 14 Batch 50 Loss 0.1009 Accuracy 0.9710
Epoch 14 Batch 100 Loss 0.1004 Accuracy 0.9713
Epoch 14 Batch 150 Loss 0.1009 Accuracy 0.9712
Epoch 14 Batch 200 Loss 0.1007 Accuracy 0.9713
Epoch 14 Loss 0.1003 Accuracy 0.9714
Time taken for 1 epoch: 198.64 secs
```

These are the epochs during training

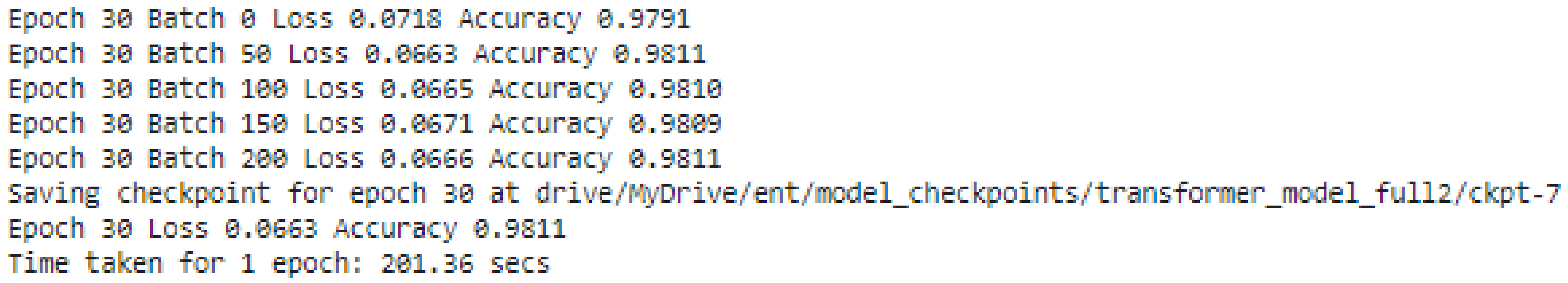
Total of 30 epochs

The last checkpoint is number 7 with loss of 0.0663 and accuracy of 0.9811 (98%)



```
Epoch 24 Batch 0 Loss 0.0948 Accuracy 0.9728
Epoch 24 Batch 50 Loss 0.0779 Accuracy 0.9779
Epoch 24 Batch 100 Loss 0.0768 Accuracy 0.9780
Epoch 24 Batch 150 Loss 0.0769 Accuracy 0.9781
Epoch 24 Batch 200 Loss 0.0766 Accuracy 0.9782
Epoch 24 Loss 0.0764 Accuracy 0.9782
Time taken for 1 epoch: 198.44 secs

Epoch 25 Batch 0 Loss 0.0842 Accuracy 0.9759
Epoch 25 Batch 50 Loss 0.0748 Accuracy 0.9788
Epoch 25 Batch 100 Loss 0.0756 Accuracy 0.9787
Epoch 25 Batch 150 Loss 0.0759 Accuracy 0.9785
Epoch 25 Batch 200 Loss 0.0753 Accuracy 0.9786
Epoch 25 Loss 0.0750 Accuracy 0.9787
Time taken for 1 epoch: 198.19 secs

Epoch 26 Batch 0 Loss 0.0798 Accuracy 0.9770
Epoch 26 Batch 50 Loss 0.0741 Accuracy 0.9790
Epoch 26 Batch 100 Loss 0.0733 Accuracy 0.9792
Epoch 26 Batch 150 Loss 0.0735 Accuracy 0.9790
Epoch 26 Batch 200 Loss 0.0732 Accuracy 0.9791
Epoch 26 Loss 0.0731 Accuracy 0.9792
Time taken for 1 epoch: 198.39 secs

Epoch 27 Batch 0 Loss 0.0755 Accuracy 0.9778
Epoch 27 Batch 50 Loss 0.0727 Accuracy 0.9797
Epoch 27 Batch 100 Loss 0.0718 Accuracy 0.9798
Epoch 27 Batch 150 Loss 0.0720 Accuracy 0.9797
Epoch 27 Batch 200 Loss 0.0714 Accuracy 0.9798
Epoch 27 Loss 0.0710 Accuracy 0.9799
Time taken for 1 epoch: 198.54 secs

Epoch 28 Batch 0 Loss 0.0740 Accuracy 0.9789
Epoch 28 Batch 50 Loss 0.0697 Accuracy 0.9806
Epoch 28 Batch 100 Loss 0.0697 Accuracy 0.9804
Epoch 28 Batch 150 Loss 0.0694 Accuracy 0.9804
Epoch 28 Batch 200 Loss 0.0694 Accuracy 0.9803
Epoch 28 Loss 0.0691 Accuracy 0.9804
Time taken for 1 epoch: 198.47 secs

Epoch 29 Batch 0 Loss 0.0817 Accuracy 0.9770
Epoch 29 Batch 50 Loss 0.0678 Accuracy 0.9810
Epoch 29 Batch 100 Loss 0.0676 Accuracy 0.9808
Epoch 29 Batch 150 Loss 0.0678 Accuracy 0.9807
Epoch 29 Batch 200 Loss 0.0676 Accuracy 0.9807
Epoch 29 Loss 0.0674 Accuracy 0.9808
Time taken for 1 epoch: 198.21 secs

Epoch 30 Batch 0 Loss 0.0718 Accuracy 0.9791
Epoch 30 Batch 50 Loss 0.0663 Accuracy 0.9811
Epoch 30 Batch 100 Loss 0.0665 Accuracy 0.9810
Epoch 30 Batch 150 Loss 0.0671 Accuracy 0.9809
Epoch 30 Batch 200 Loss 0.0666 Accuracy 0.9811
Saving checkpoint for epoch 30 at drive/MyDrive/ent/model_checkpoin        full2/ckpt-7
Epoch 30 Loss 0.0663 Accuracy 0.9811
Time taken for 1 epoch: 201.36 secs
```

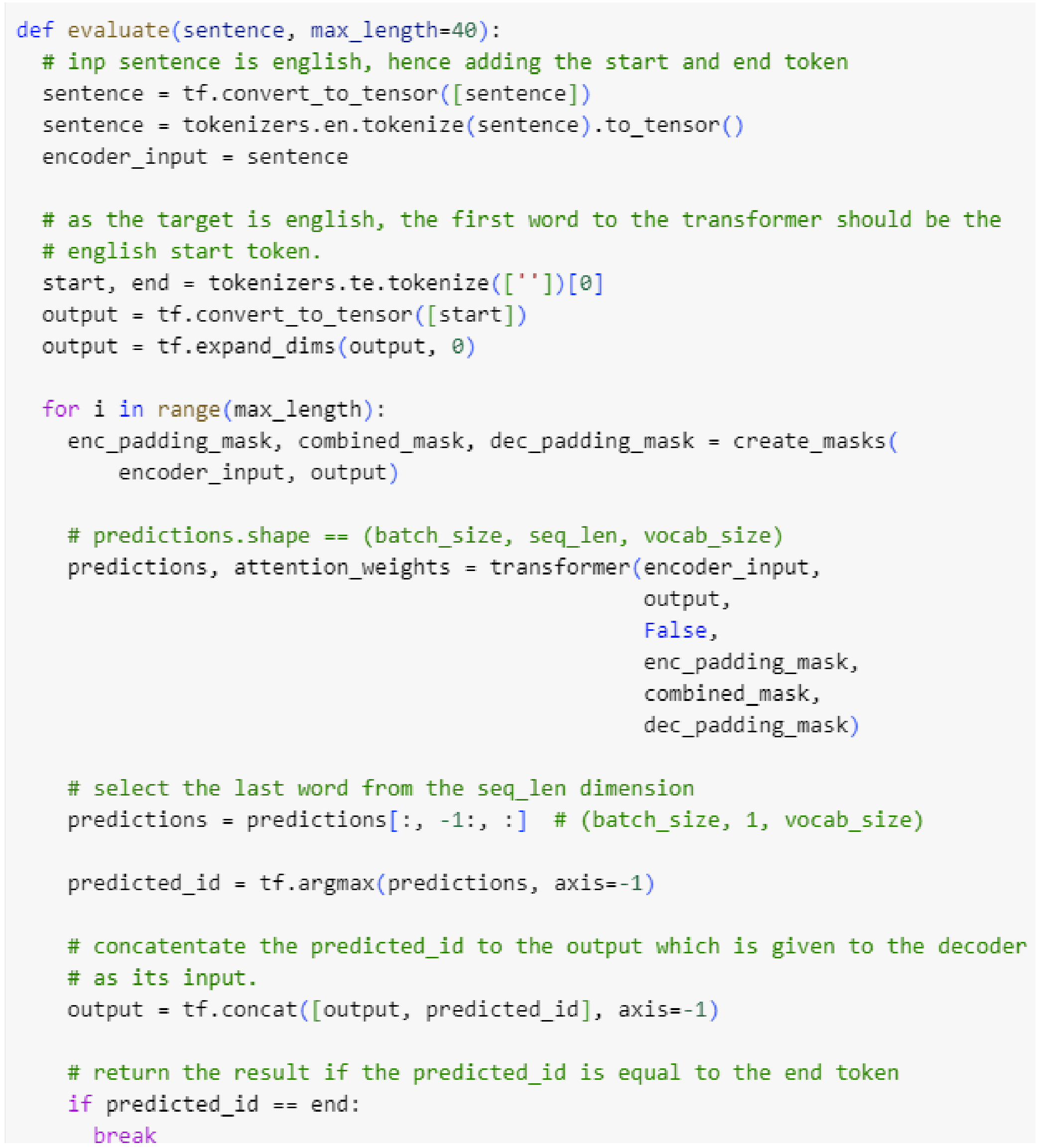
TRAINING



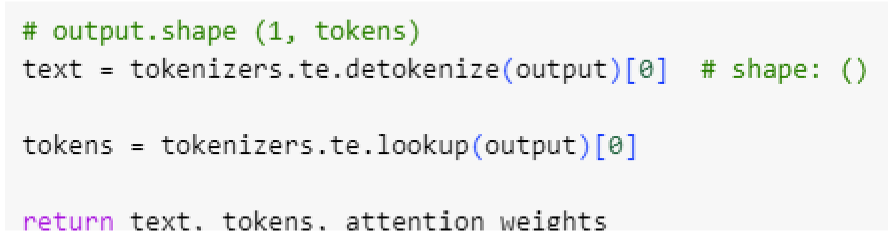
```
Epoch 30 Batch 0 Loss 0.0718 Accuracy 0.9791
Epoch 30 Batch 50 Loss 0.0663 Accuracy 0.9811
Epoch 30 Batch 100 Loss 0.0665 Accuracy 0.9810
Epoch 30 Batch 150 Loss 0.0671 Accuracy 0.9809
Epoch 30 Batch 200 Loss 0.0666 Accuracy 0.9811
Saving checkpoint for epoch 30 at drive/MyDrive/ent/model_checkpoints/transformer_model_full2/ckpt-7
Epoch 30 Loss 0.0663 Accuracy 0.9811
Time taken for 1 epoch: 201.36 secs
```

Achieved accuracy of 98% on training evaluation for 30 epochs

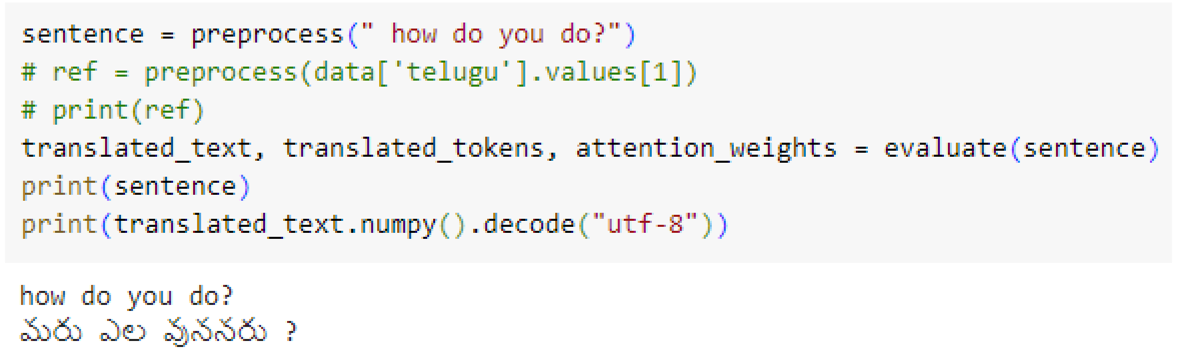
TESTING



```
def evaluate(sentence, max_length=40):
  # inp sentence is english, hence adding the start and end token
  sentence = tf.convert_to_tensor([sentence])
  sentence = tokenizers.en.tokenize(sentence).to_tensor()
  encoder_input = sentence

  # as the target is english, the first word to the transformer should be the
  # english start token.
  start, end = tokenizers.te.tokenize([''])[0]
  output = tf.convert_to_tensor([start])
  output = tf.expand_dims(output, 0)

  for i in range(max_length):
    enc_padding_mask, combined_mask, dec_padding_mask = create_masks(
        encoder_input, output)

    # predictions.shape == (batch_size, seq_len, vocab_size)
    predictions, attention_weights = transformer(encoder_input,
                                                 output,
                                                 False,
                                                 enc_padding_mask,
                                                 combined_mask,
                                                 dec_padding_mask)

    # select the last word from the seq_len dimension
    predictions = predictions[:, -1:, :]  # (batch_size, 1, vocab_size)

    predicted_id = tf.argmax(predictions, axis=-1)

    # concatentate the predicted_id to the output which is given to the decoder
    # as its input.
    output = tf.concat([output, predicted_id], axis=-1)

    # return the result if the predicted_id is equal to the end token
    if predicted_id == end:
      break
```



```
# output.shape (1, tokens)
text = tokenizers.te.detokenize(output)[0]  # shape: ()

tokens = tokenizers.te.lookup(output)[0]

return text, tokens, attention_weights
```

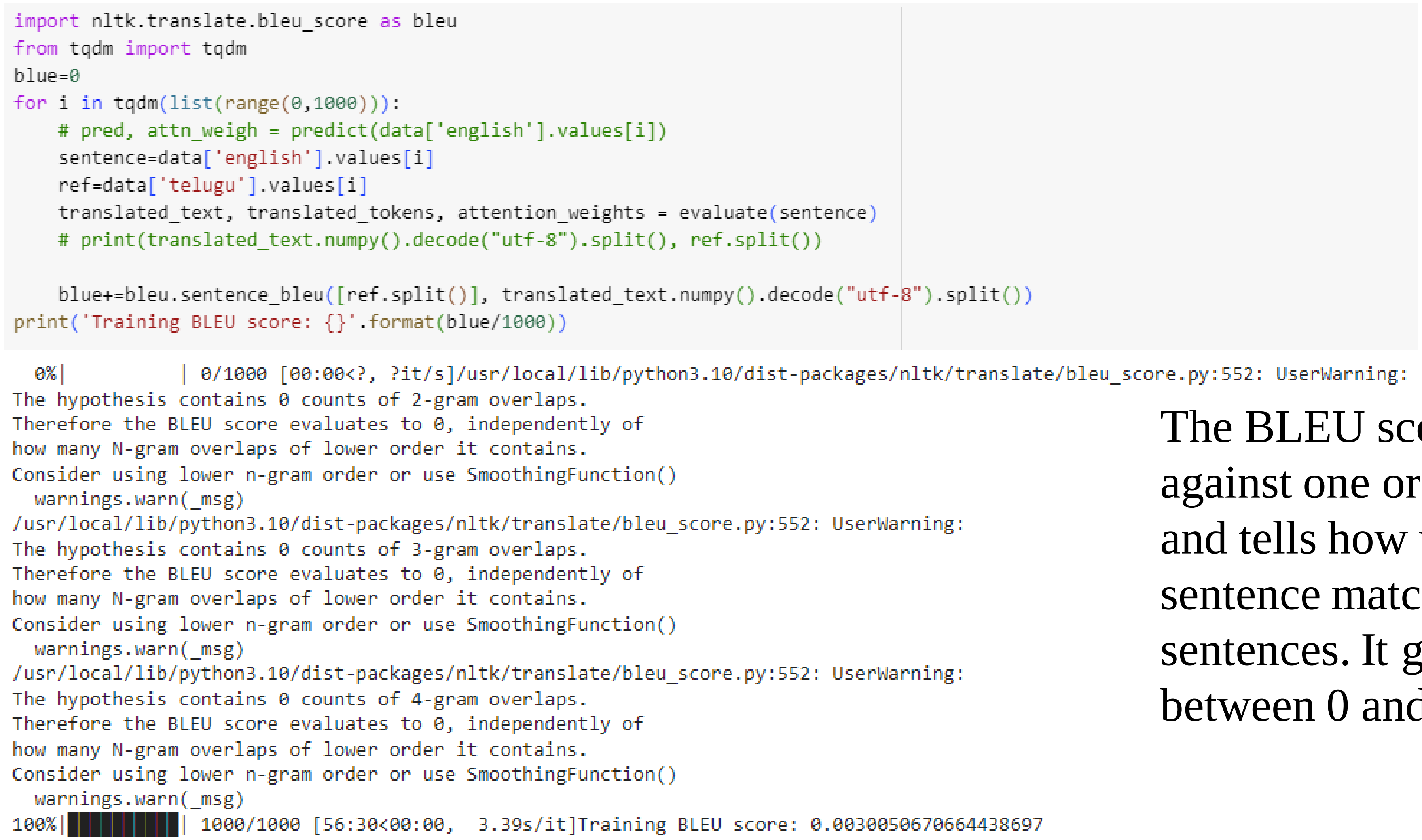


```
sentence = preprocess(" how do you do?")
# ref = preprocess(data['telugu'].values[1])
# print(ref)
translated_text, translated_tokens, attention_weights = evaluate(sentence)
print(sentence)
print(translated_text.numpy().decode("utf-8"))
```

```
how do you do?
మరు ఎల వునసరు ?
```

Evaluation is done by first preprocessing the input , getting output as text, tokens, weights after predicting using transformer model which is already trained

EVALUATION METRICS

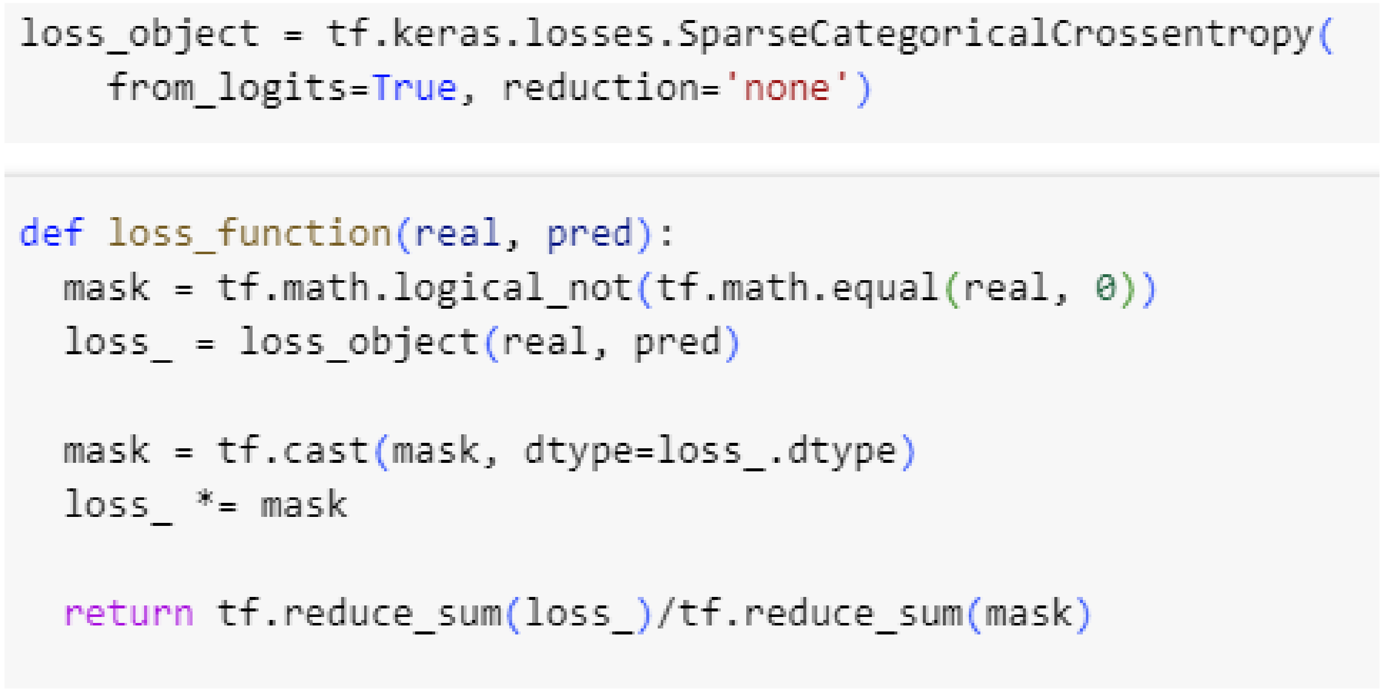


```python
import nltk.translate.bleu_score as bleu
from tqdm import tqdm
blue=0
for i in tqdm(list(range(0,1000))):
    # pred, attn_weigh = predict(data['english'].values[i])
    sentence=data['english'].values[i]
    ref=data['telugu'].values[i]
    translated_text, translated_tokens, attention_weights = evaluate(sentence)
    # print(translated_text.numpy().decode("utf-8").split(), ref.split())

    blue+=bleu.sentence_bleu([ref.split()], translated_text.numpy().decode("utf-8").split())
print('Training BLEU score: {}'.format(blue/1000))
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 2-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 3-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 4-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
100%|██████████| 1000/1000 [56:30<00:00,  3.39s/it]Training BLEU score: 0.0030050670664438697
```

The BLEU score compares a sentence against one or more reference sentences and tells how well does the candidate sentence matched the list of reference sentences. It gives an output score between 0 and 1

EVALUATION METRICS



```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
```

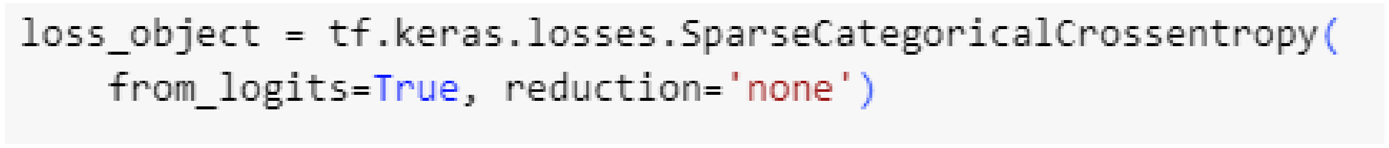
```
def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)

  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_sum(loss_)/tf.reduce_sum(mask)
```

tf.keras.losses.SparseCategoricalCrossentropy: This is an implementation of the Sparse Categorical Crossentropy loss function provided by TensorFlow's Keras API. It's commonly used for multi-class classification tasks where the target labels are integers.

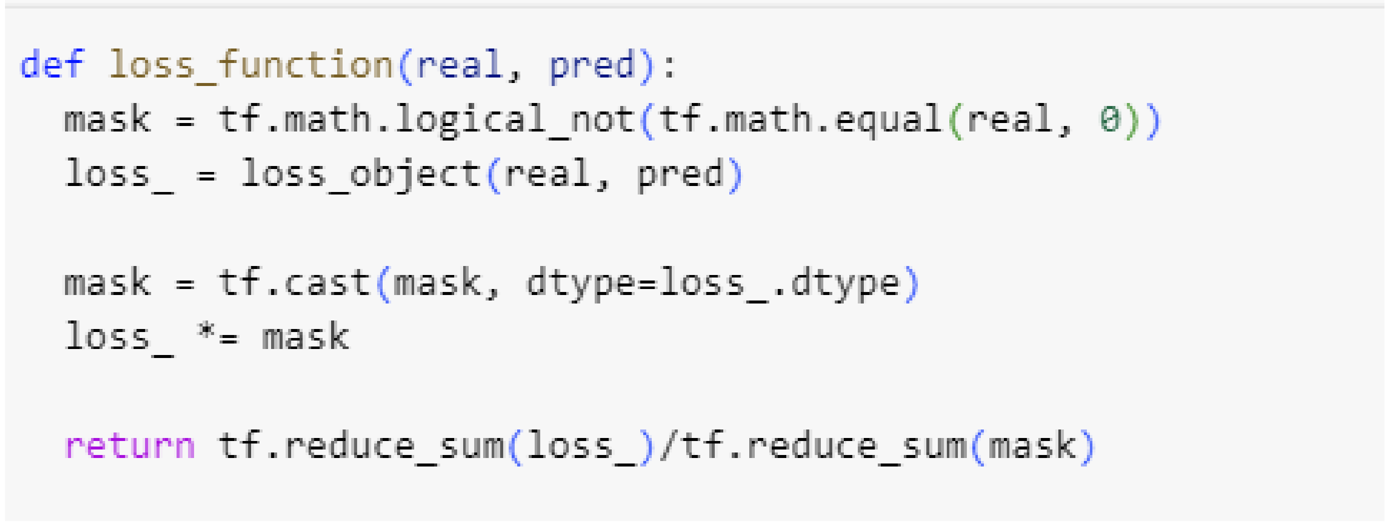
loss_object=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none'): This line creates an instance of the SparseCategoricalCrossentropy loss function.

from_logits=True: Indicates that the input predictions are not normalized (i.e., they are raw logits) and need to be normalized (e.g., via a softmax activation) before being used in the loss calculation.
reduction='none': Specifies that no reduction should be applied to the individual losses. This means that the loss will be calculated separately for each sample in the batch, rather than being aggregated across the batch.

EVALUATION METRICS



```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
```



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

  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_sum(loss_)/tf.reduce_sum(mask)
```

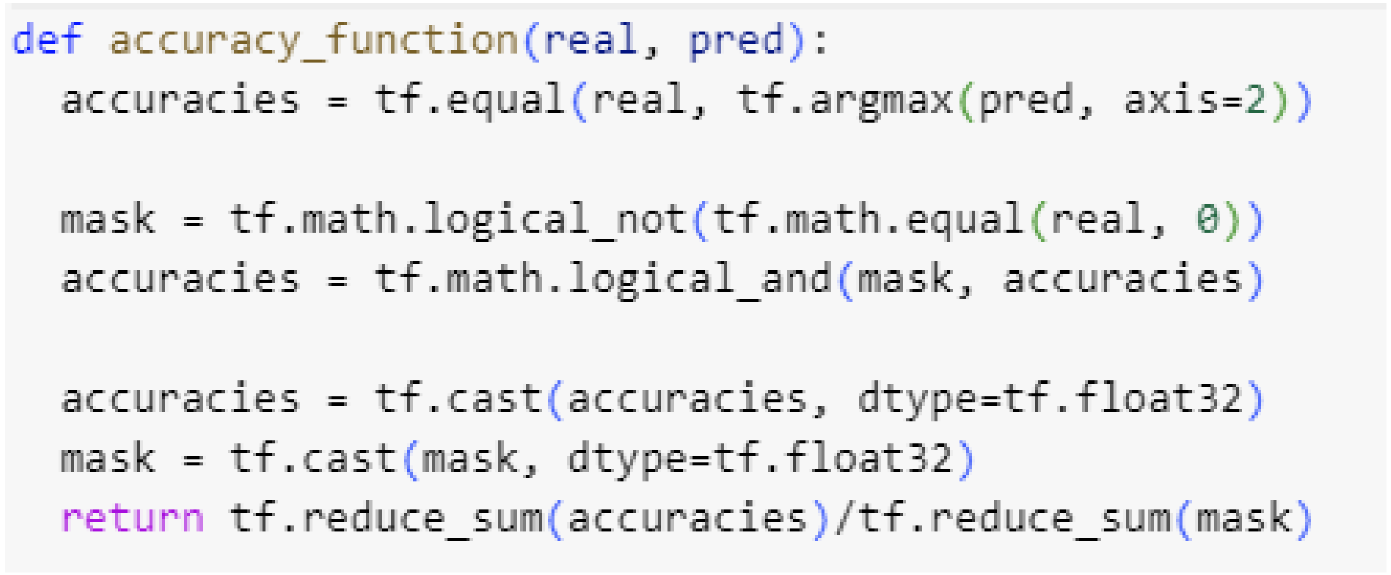
mask = tf.math.logical_not(tf.math.equal(real, 0)): This line creates a mask tensor where elements of real that are equal to 0 are marked as False and all other elements are marked as True. This is typically used to mask out padding elements in sequence data.

loss_ = loss_object(real, pred): This line calculates the loss using the SparseCategoricalCrossentropy loss function created earlier. Since reduction='none', this will return a tensor containing the individual losses for each sample in the batch.

mask = tf.cast(mask, dtype=loss_.dtype): This line casts the mask tensor to the same data type as the computed loss tensor.

loss_ *= mask: This line applies the mask to the computed loss tensor. This effectively zeros out the loss for padding elements, as they are marked as False in the mask. And at last we calculate **mean losses**

EVALUATION METRICS



```
def accuracy_function(real, pred):
  accuracies = tf.equal(real, tf.argmax(pred, axis=2))

  mask = tf.math.logical_not(tf.math.equal(real, 0))
  accuracies = tf.math.logical_and(mask, accuracies)

  accuracies = tf.cast(accuracies, dtype=tf.float32)
  mask = tf.cast(mask, dtype=tf.float32)
  return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)
```

tf.equal(real, tf.argmax(pred, axis=2)) returns a tensor where each element is True if the corresponding element in real matches the index of the maximum value in the corresponding position of pred, and False otherwise

mask =tf.math.logical_not(tf.math.equal(real, 0)): This creates a boolean mask where True indicates positions where the real values are not equal to 0 (non-padding positions).

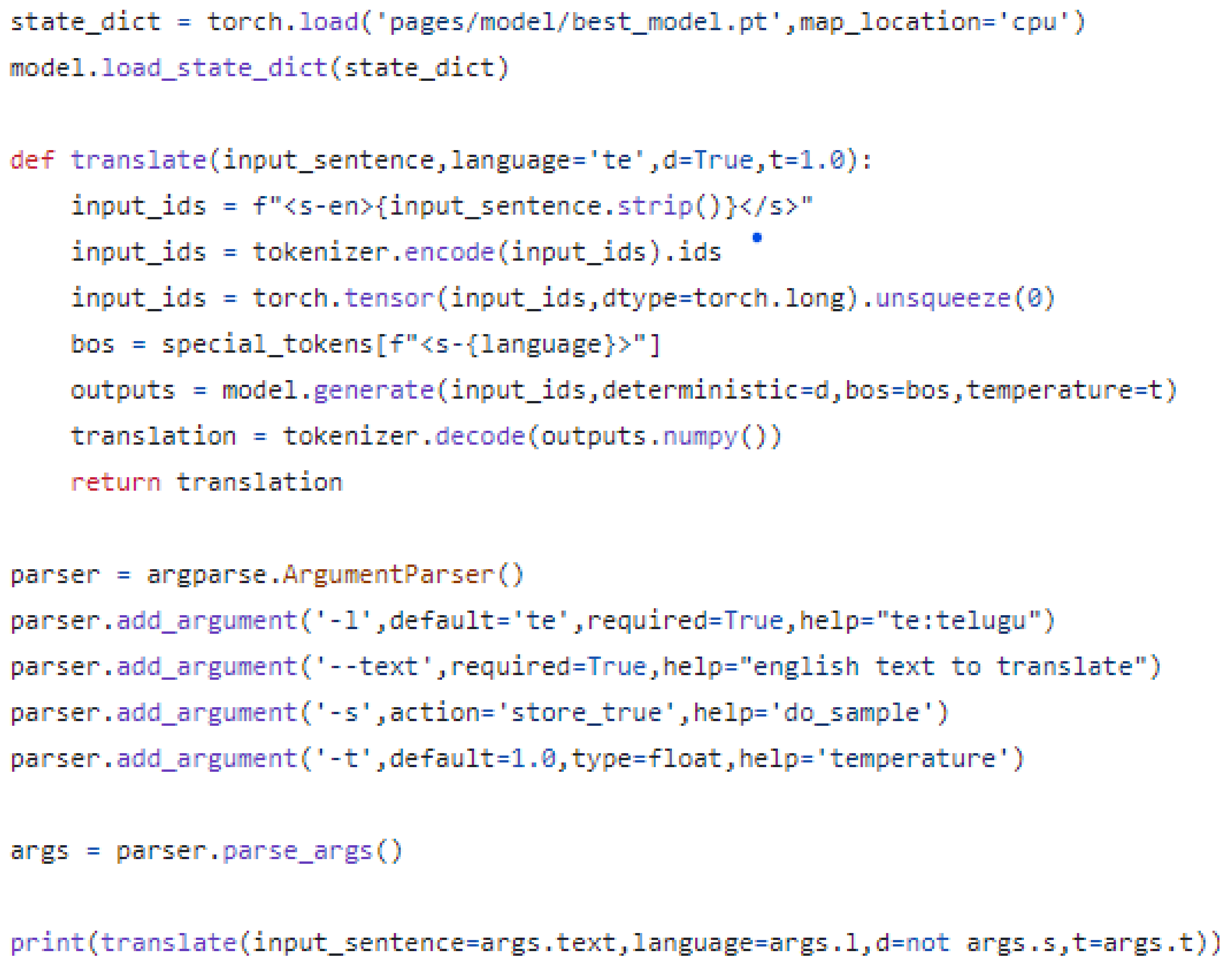
tf.math.logical_and(mask, accuracies): Here, the mask is further refined by performing a logical AND operation with the previously computed accuracies. This ensures that only positions where the real values are not padding and the predictions are correct contribute to the accuracy calculation.

accuracies = tf.cast(accuracies, dtype=tf.float32): Converts the boolean tensor of accuracies to float32 data type. This conversion is necessary for subsequent calculations.

mask = tf.cast(mask, dtype=tf.float32): Similarly, converts the boolean mask to float32 data type. This will be used for normalization in the accuracy calculation.

Finally **mean accuracy** is calculated

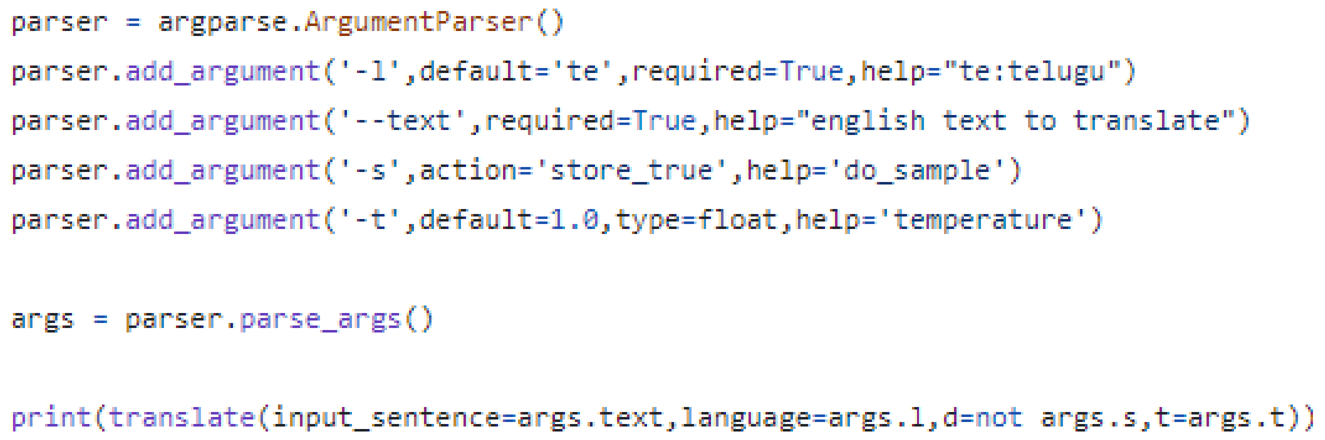
DEPLOYMENT



```
state_dict = torch.load('pages/model/best_model.pt',map_location='cpu')
model.load_state_dict(state_dict)

def translate(input_sentence,language='te',d=True,t=1.0):
    input_ids = f"<s-en>{input_sentence.strip()}</s>"
    input_ids = tokenizer.encode(input_ids).ids
    input_ids = torch.tensor(input_ids,dtype=torch.long).unsqueeze(0)
    bos = special_tokens[f"<s-{language}>"]
    outputs = model.generate(input_ids,deterministic=d,bos=bos,temperature=t)
    translation = tokenizer.decode(outputs.numpy())
    return translation

parser = argparse.ArgumentParser()
parser.add_argument('-l',default='te',required=True,help="te:telugu")
parser.add_argument('--text',required=True,help="english text to translate")
parser.add_argument('-s',action='store_true',help='do_sample')
parser.add_argument('-t',default=1.0,type=float,help='temperature')

args = parser.parse_args()

print(translate(input_sentence=args.text,language=args.l,d=not args.s,t=args.t))
```

The implementation involves utilizing a pre-trained transformer model, saved as a pickle file, which is loaded using PyTorch for predicting Telugu text. The process begins by converting the provided input, initially in English, into tokens. These tokens are then utilized as input for the transformer model for prediction.

DEPLOYMENT

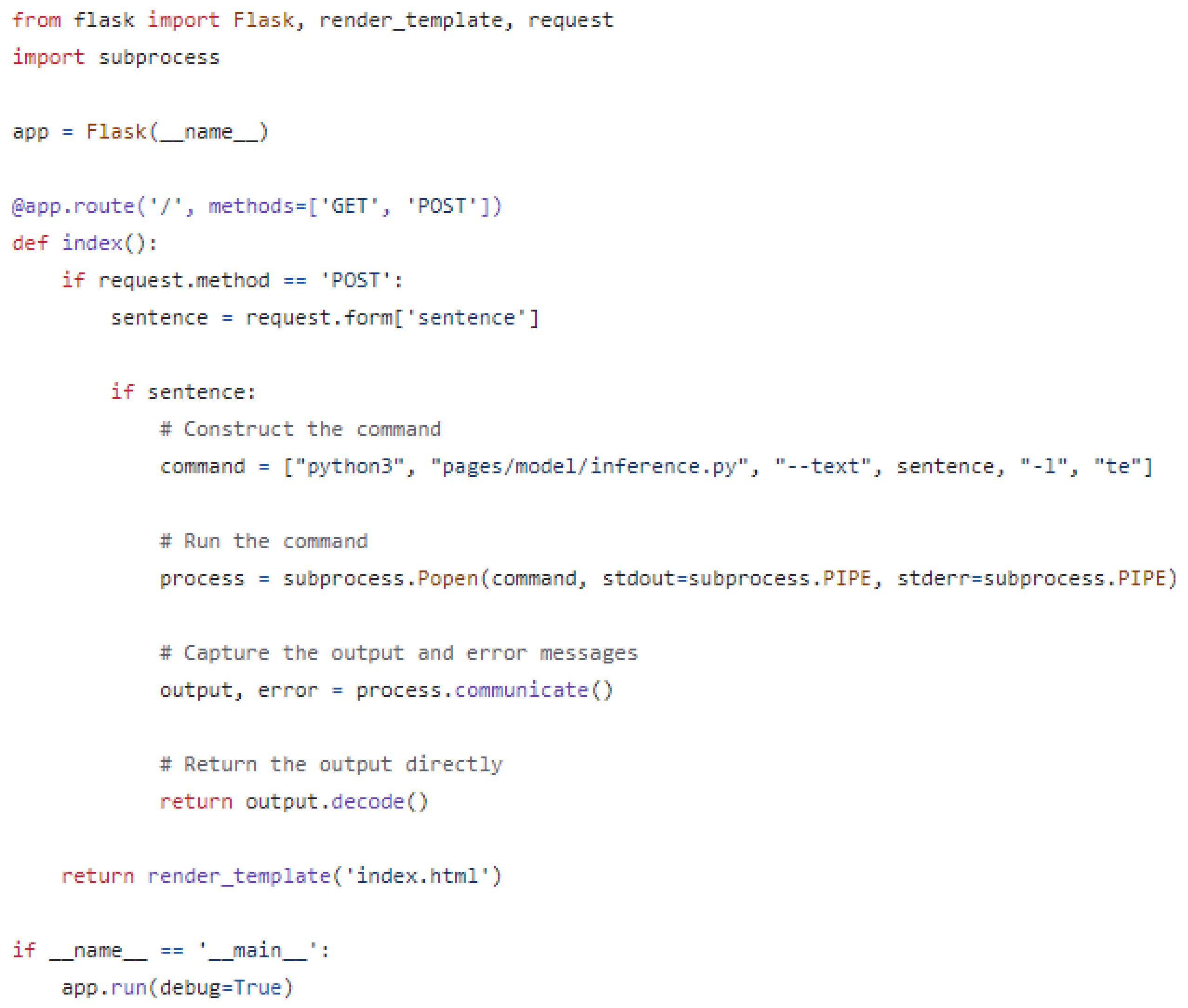


```
parser = argparse.ArgumentParser()
parser.add_argument('-l',default='te',required=True,help="te:telugu")
parser.add_argument('--text',required=True,help="english text to translate")
parser.add_argument('-s',action='store_true',help='do_sample')
parser.add_argument('-t',default=1.0,type=float,help='temperature')

args = parser.parse_args()

print(translate(input_sentence=args.text,language=args.l,d=not args.s,t=args.t))
```

The implementation supports command-line arguments for customization. Specifically, the following arguments are utilized:
This implementation allows for seamless integration of language translation functionalities, providing users with a versatile tool for processing multilingual text inputs.

-l: Denotes the language. Presently, the system is configured for Telugu but can be generalized and extended to accommodate other languages.
--text: Represents the input text. Currently, the system accepts English input but can be expanded to support other languages.
-s: Enables sampling.
-t: Represents the temperature parameter, facilitating the generation of diverse contextual outputs.

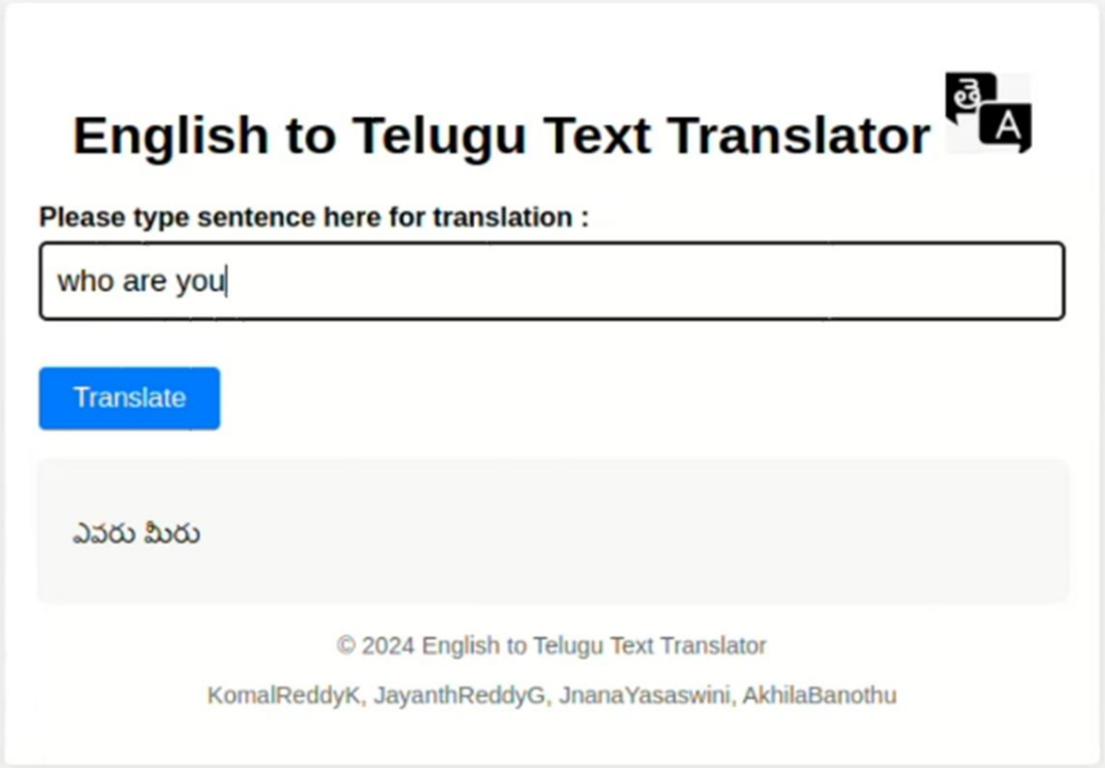
DEPLOYMENT



```python
from flask import Flask, render_template, request
import subprocess

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        sentence = request.form['sentence']

        if sentence:
            # Construct the command
            command = ["python3", "pages/model/inference.py", "--text", sentence, "-l", "te"]

            # Run the command
            process = subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

            # Capture the output and error messages
            output, error = process.communicate()

            # Return the output directly
            return output.decode()

    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

The deployment strategy involves utilizing Flask to host the application locally. When a POST request is made to the server via a form submission, Flask triggers the execution of a specific command. Subsequently, Flask renders an HTML file to provide a user interface for interaction.

RESULT



English to Telugu Text Translator

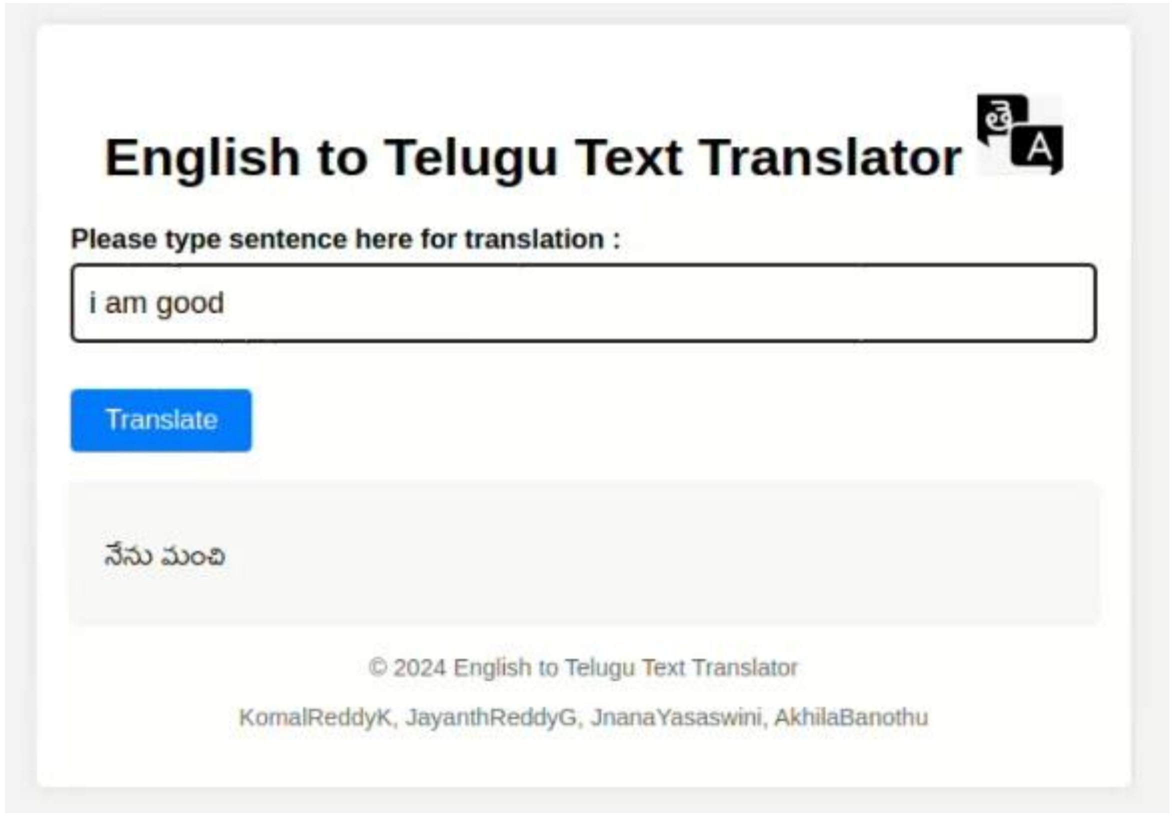
Please type sentence here for translation :

who are you

ఎవరు మీరు

© 2024 English to Telugu Text Translator

KomalReddyK, JayanthReddyG, JnanaYasaswini, AkhilaBanothu



English to Telugu Text Translator

Please type sentence here for translation :

i am good

నేను మంచి

© 2024 English to Telugu Text Translator

KomalReddyK, JayanthReddyG, JnanaYasaswini, AkhilaBanothu

CHALLENGES FACED

BLEU Score Evaluation: One of the challenges faced was ensuring high BLEU (Bilingual Evaluation Understudy) scores, which are critical for assessing the quality of machine-translated text. Achieving a high BLEU score requires the model to produce translations that closely match reference translations, capturing both accuracy and fluency.

Dataset Size and Quality: A significant challenge was the need for large, high-quality parallel corpora to train the Transformer model effectively. Large datasets help the model learn better representations and translations, but collecting and curating such datasets, especially for low-resource languages, was a daunting task.

Training Time: Training the Transformer model took an extensive amount of time, even with powerful hardware. The complexity of the model and the need for processing large datasets meant that training could span days or even weeks. By using google colab we managed to train for 1 lakh 30 thousand's rows data in sets which helped us in this management of CPU and RAM.

Translation Accuracy: Ensuring high translation accuracy was a continuous challenge. The model needed to not only understand the source text accurately but also generate coherent and contextually appropriate translations in the target language. This required extensive tuning and evaluation, with careful attention to handling linguistic nuances, idiomatic expressions

REFERENCES

REFERENCES

[1] Lingam, K., Ramalakshmi, E., & Inturi, S. (2014). English to Telugu Rule-based Machine Translation System: A Hybrid Approach. International Journal of Computer Applications, Computer Science, Linguistics. Published September 18, 2014. ***accessed in October 2023***

[2] Premjith, B., Kumar, M. A., & Soman, K. P. (2019). Neural Machine Translation System for English to Indian Language Translation Using MTIL Parallel Corpus. Journal of Intelligent Systems, 28(3), Special Issue on Natural Language Processing. De Gruyter. DOI:10.1515/jisys-2019-2510. License: CC BY-NC-ND 3.0 ***accessed in October 2023***

[3] Swaroopa, P., Vishnu, K., Akshay, P., Sandip, S. S. S., & Vivek, V. (2022). A Survey on Formal Language Translation (Telugu - English). ACE Engineering College, Hyderabad, Telangana, India, ***accessed in October 2023***

[4] https://github.com/himanshudce/Indian-Language-Dataset ***accessed in November 2023***

[5] https://www.tensorflow.org/text/tutorials/transformer ***accessed in November 2023***

[6] https://pytorch.org/docs/stable/generated/torch.save.html ***accessed in April 2024***

[7] https://flask.palletsprojects.com/en/3.0.x/ ***accessed in May 2024***

TIMELINE

Oct. 2023

Dataset Preparation

Data Preprocessing

Nov. 2023

Creating Vocab File

Custom tokenizer

Dec. 2023

Building Attention Mechanism

Jan. 2024

Encoder Layer

Feb. 2024

Decoder Layer

Mar. 2024

Transformer and training

Apr. 2024

Testing

May. 2024

Deployment

CONCLUSION

CONCLUSION

In conclusion, the English to Telugu text translation system employs a Transformers model, facilitating accurate and contextually rich translations. This model is deployed using Flask, a lightweight web framework, which orchestrates the execution of commands upon receiving POST requests from user forms. Leveraging Flask's capabilities, the system seamlessly integrates the prediction functionality, with the model loaded via a pickle file, and utilizes HTML and CSS for rendering templates, ensuring an intuitive and visually appealing user interface.

FUTURE SCOPE

FUTURE SCOPE

In the future, our aim is to enhance the English to Telugu text translation system by deploying it on a global scale rather than locally, thereby increasing its accessibility and reach. Additionally, acquiring a larger dataset and conducting thorough training would be pivotal in improving the system's translation quality, thereby enhancing its BLEU score. This expansion and refinement of the system would enable it to better serve users across diverse linguistic contexts and contribute to more accurate and nuanced translations between English and Telugu.

WE

Jayanth Reddy(20103010)

Akhila Banothu (20103052)

AS A TEAM THANK YOU