Task 2 (Translation) [30 marks]

Dataset:
WNT16 (german-english) dataset. https://huggingface.co/datasets/wmt16/viewer/de-en
Loading the model using HuggingFace: load_dataset('wmt16', 'de-en')
Task: Machine translation from English to german.
Model A: LSTM
Model B: LSTM + GLOBAL ATTENTION
Model C: LSTM + LOCAL ATTENTION
Refer to this paper for global and local attention concepts: https://aclanthology.org/D15-1166/

Note : For dataset loading, students can use HuggingFace dataset for maintaining the train-test consistency. Also for comparison use the same set of embeddings for each model setup. The embedding layer has to be trainable in all 3 models.

1. Visualise the dataset using five different techniques (sequence length, frequency of words, mean token length, word cloud, etc) to show how the 2 languages differ. [6 marks]

2. Implement Model A with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM based [5 marks]

3. Implement Model B with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM + global attention based [5 marks]

4. Implement Model B with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM + local attention based [5 marks]

5. Show plots of validation and training loss vs. number of epochs for Models A, B, C. [3 marks]

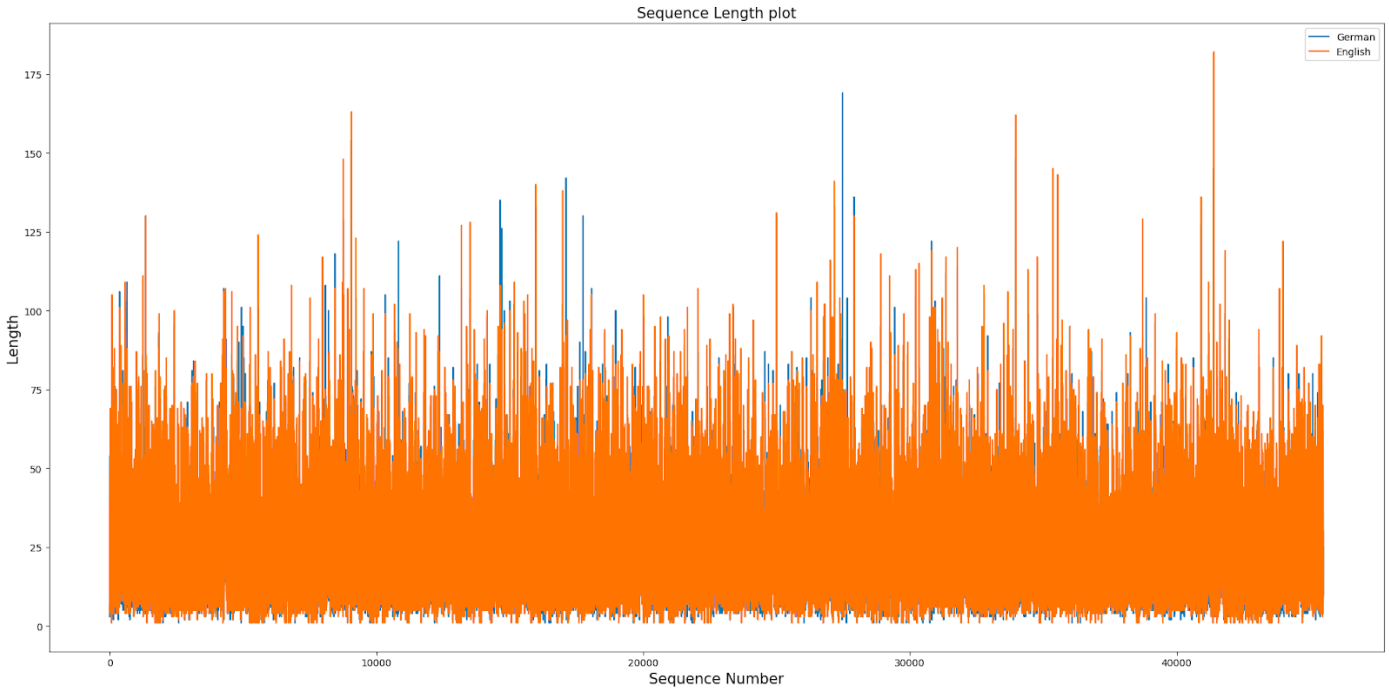
6. Report the overall Rouge-L score, Bleu-1,2 score for the test set. (metrics available in Huggingface metric) for Models A, B, C. [4 marks]

7. Out of the 3 models, which setup performed best and why? [2 marks]

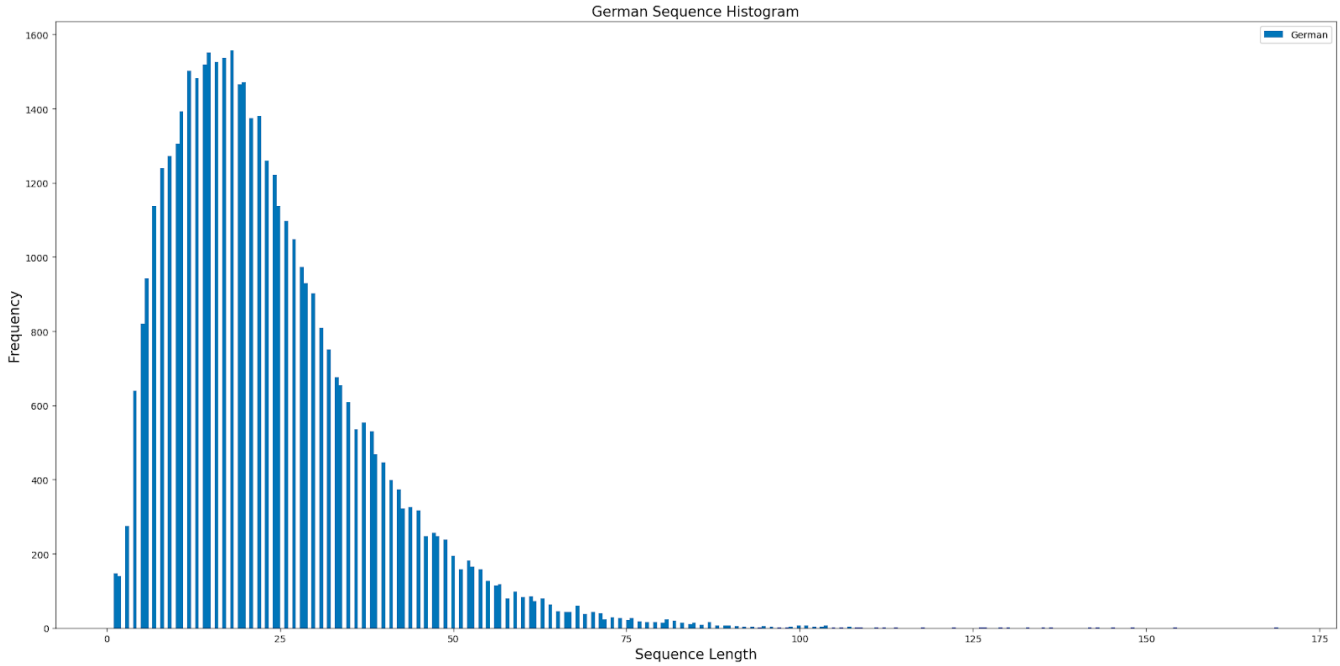
TASK II (Machine Translation) by Debnath Kundu

1. Visualise the dataset using five different techniques (sequence length, frequency of words, mean token length, word cloud, etc) to show how the 2 languages differ. [6 marks]

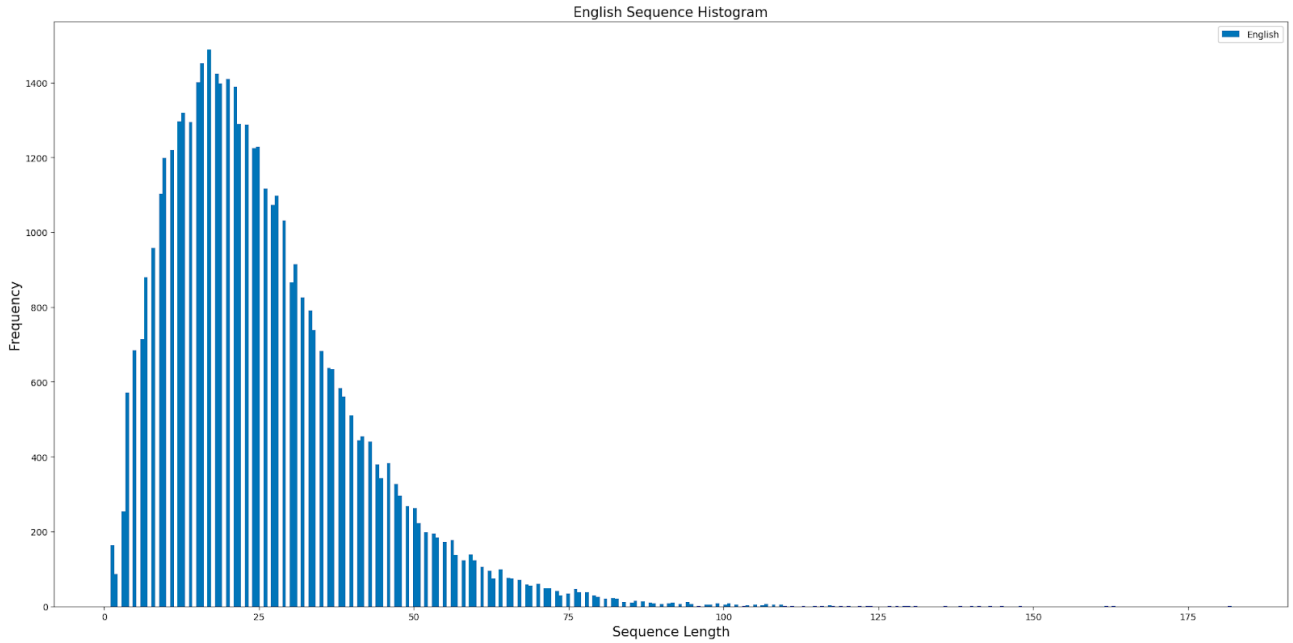
Sequence Length plot:



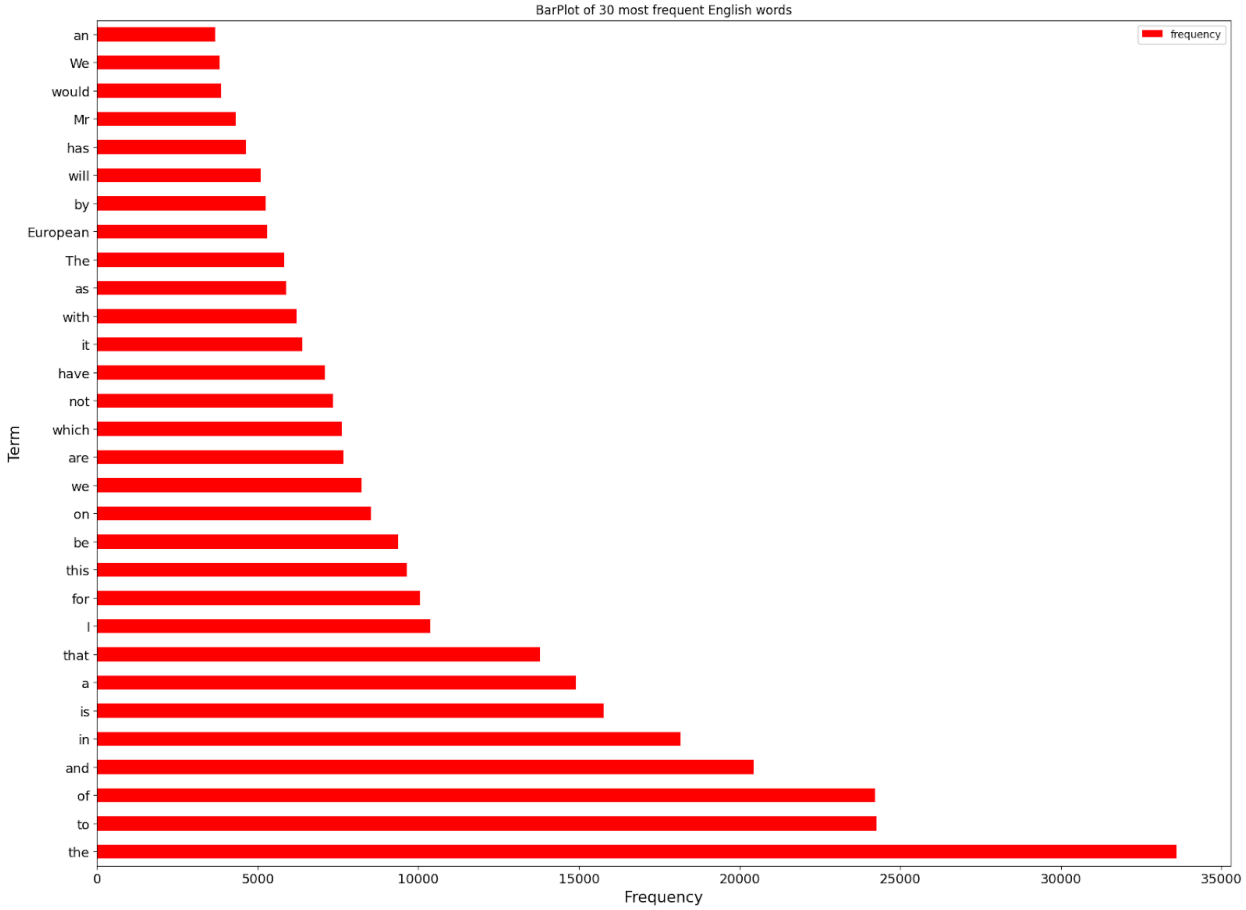
German Sequence Histogram:



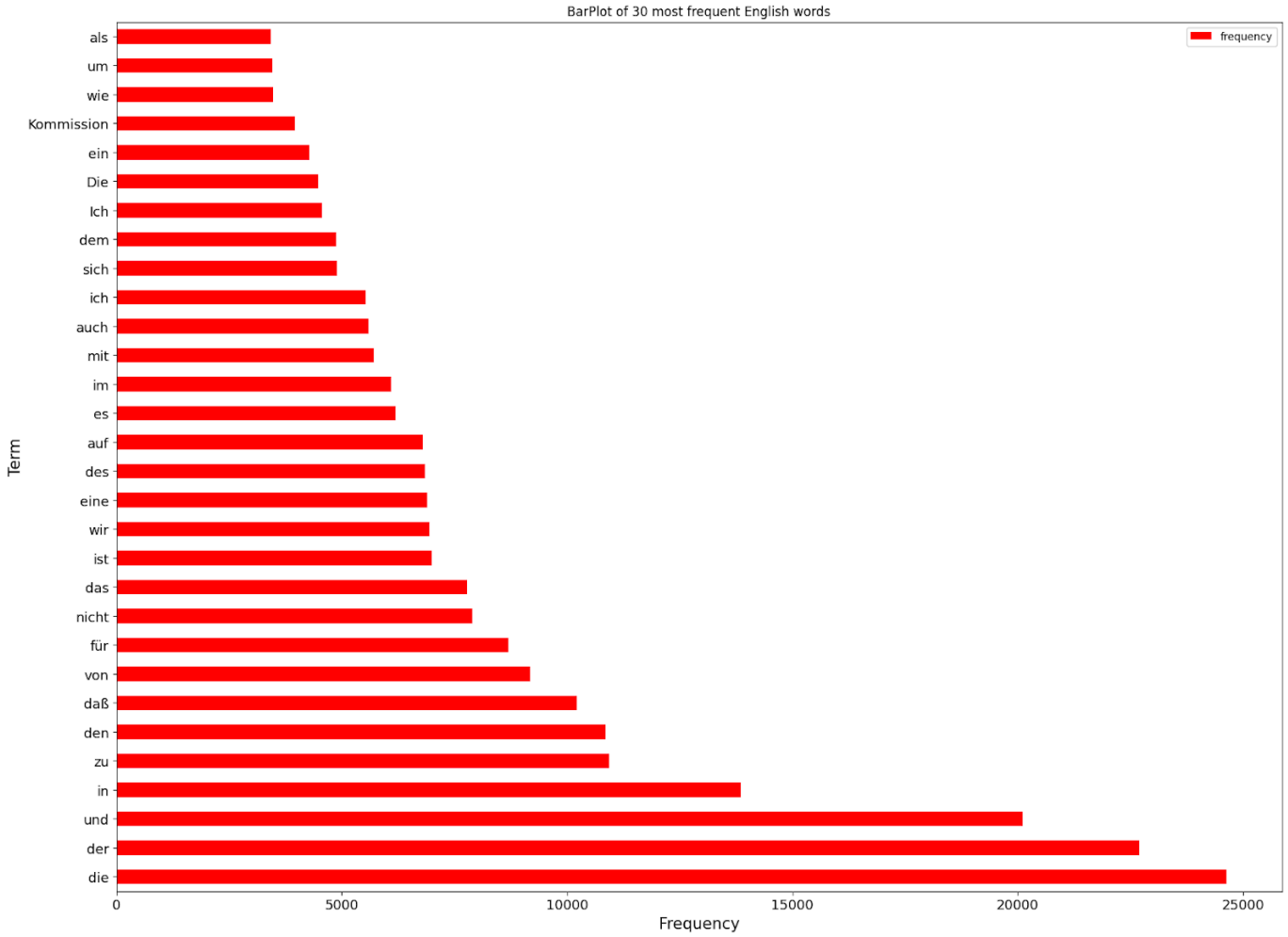
English Sequence Histogram:



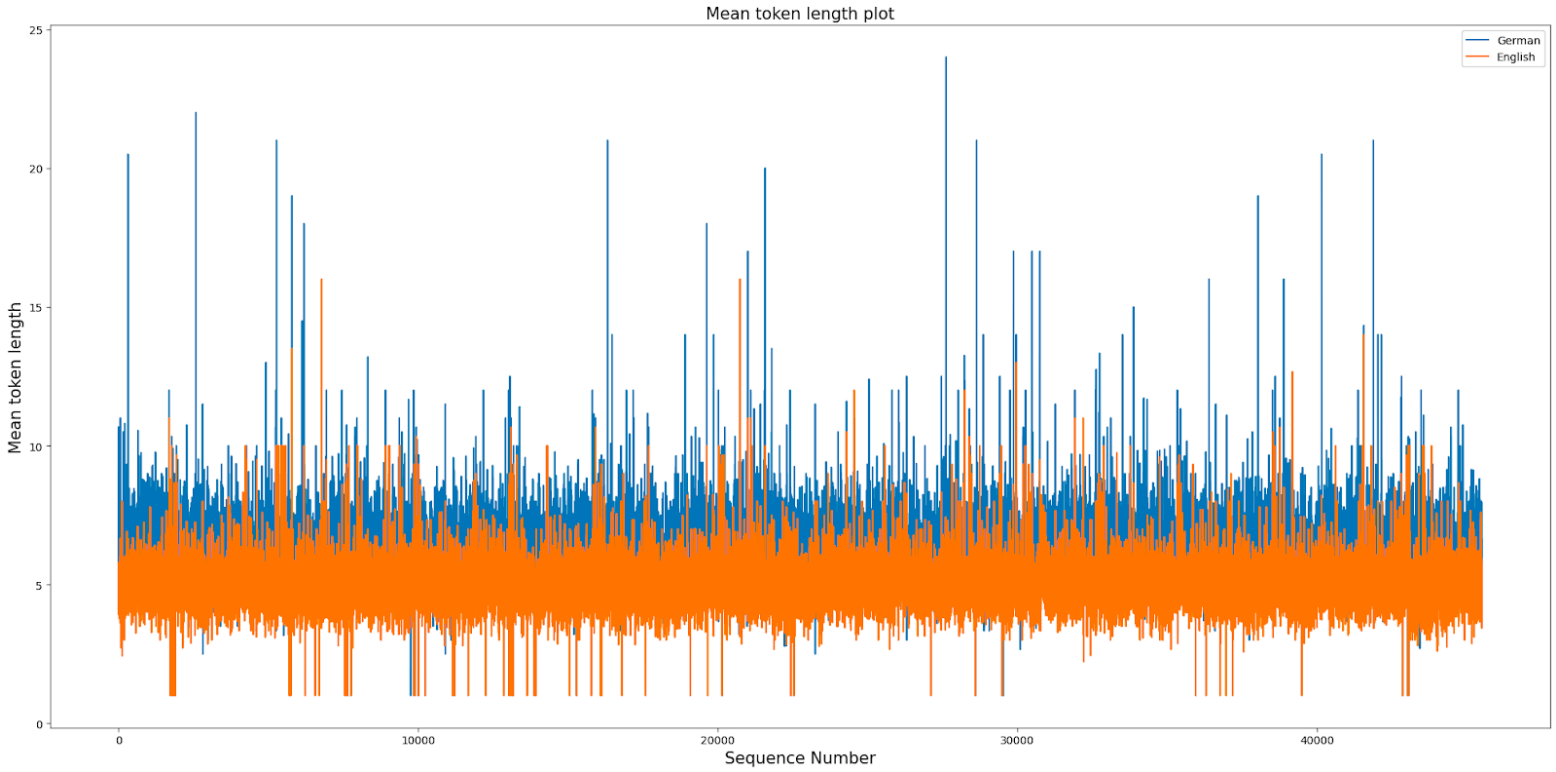
Frequency of words (English):



Frequency of words (German):



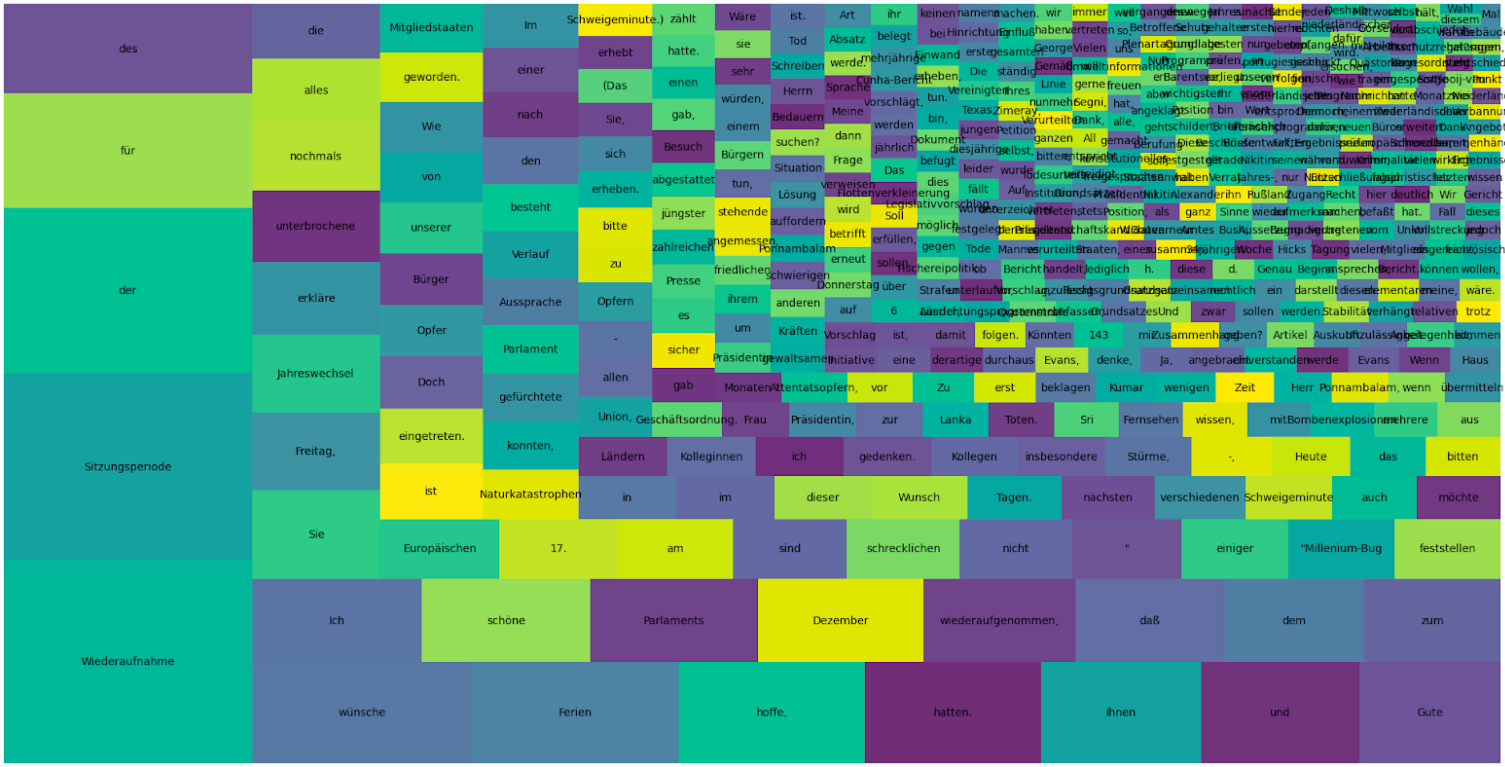
Mean token length:



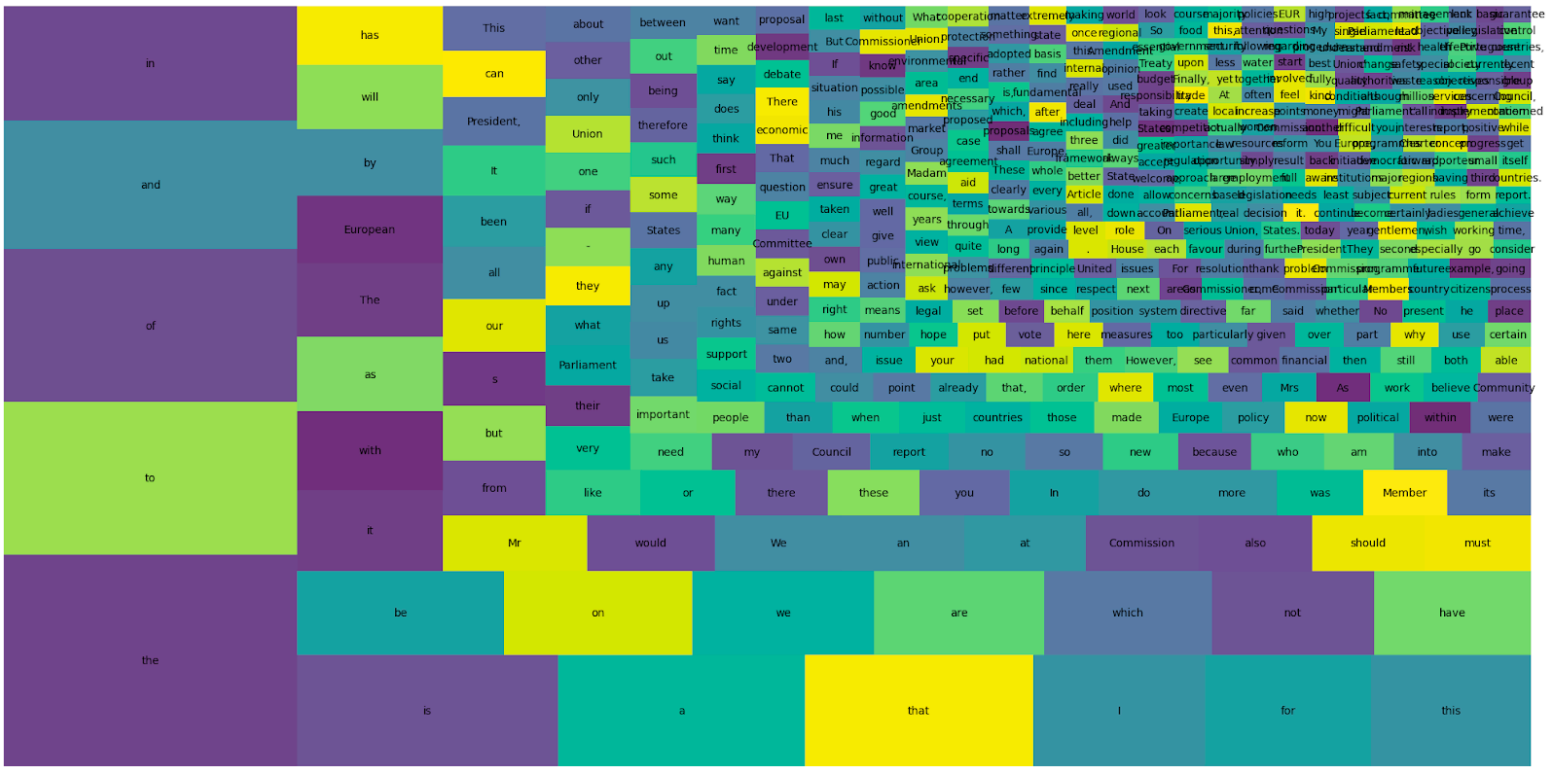
Word cloud (German):

Word cloud (English):

Tree Maps(German):



Tree Maps(English):



2. Implement Model A with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM based [5 marks]

Reference

GloVe Embedding

```
[ ]
```

```
import pandas as pd
glove = pd.read_csv('/content/drive/MyDrive/2 Sem/DL/Assignments/A3_BharatGoyal_DebnathKundu_SaloniAgrawal/Q2/glove.6B/glove.6B.100d.t
glove_embedding = {key: val.values for key, val in glove.T.items()}
```

```
import numpy as np
import tensorflow as tf

def embedding(word_index,embedding_dict,dimension):
  embedding_matrix=np.zeros((len(word_index)+1,dimension))

  for word,index in word_index.items():
    if word in embedding_dict:
      embedding_matrix[index]=embedding_dict[word]
  return embedding_matrix

text=["The cat sat on mat","we can play with model"]

tokenizer=tf.keras.preprocessing.text.Tokenizer(split=" ")
tokenizer.fit_on_texts(text)

text_token=tokenizer.texts_to_sequences(text)

embedding_matrix=embedding(tokenizer.word_index,embedding_dict=glove_embedding,dimension=100)

print("Loading embedding matrix...")
```

```
Loading embedding matrix...
```

```
model_build_metrics  = build()
model_build_metrics
```

```
/usr/local/lib/python3.9/dist-packages/torch/nn/modules/rnn.py:71: UserWarning: dropout option adds dropout after all but last recurrent lay
  warnings.warn("dropout option adds dropout after all but last "
8270it [13:22, 10.30it/s]
Epoch: 01 | Train Loss: 6.348 | Val. Loss: 7.078 | Val. Acc 0.02715313198339901
8270it [13:05, 10.53it/s]
Epoch: 02 | Train Loss: 6.114 | Val. Loss: 7.145 | Val. Acc 0.02868352021377115
8270it [13:07, 10.50it/s]
Epoch: 03 | Train Loss: 6.034 | Val. Loss: 7.162 | Val. Acc 0.02441410042095113
8270it [13:07, 10.50it/s]
Epoch: 04 | Train Loss: 5.950 | Val. Loss: 7.113 | Val. Acc 0.027305007173420784
8270it [13:08, 10.49it/s]
Epoch: 05 | Train Loss: 5.873 | Val. Loss: 7.194 | Val. Acc 0.024371954688757767
8270it [13:08, 10.49it/s]
Epoch: 06 | Train Loss: 5.811 | Val. Loss: 7.234 | Val. Acc 0.02346383079305003
8270it [13:06, 10.52it/s]
Epoch: 07 | Train Loss: 5.752 | Val. Loss: 7.304 | Val. Acc 0.01980236926953039
8270it [13:01, 10.58it/s]
Epoch: 08 | Train Loss: 5.702 | Val. Loss: 7.257 | Val. Acc 0.02233709548308444
Test Loss: (7.122198824564616, 0.02635075628447036)
[(6.347791186209313, 7.077691762267336, 0.02715313198339901),
 (6.114353677756553, 7.144678993997872, 0.02868352021377115),
 (6.033791439680264, 7.16198544563729, 0.02441410042095113),
 (5.950423941963962, 7.112838108456157, 0.027305007173420784),
 (5.872837891659397, 7.194189089218336, 0.024371954688757767),
 (5.811443699083709, 7.23400452650713, 0.02346383079305003),
 (5.751714906640612, 7.304295721633658, 0.01980236926953039),
 (5.701856358524101, 7.256542305041853, 0.02233709548308444)]
```

Rouge-L Score:

```
result=63.726418907970974
# print(result)
print(result/3000) #3000 is the test set length i.e len(test_loader)

#0.021
```

```
0.021242139635990323
```

Overall **Rouge-L** score: **0.021242139635990323**

ROUGE-L measures the longest common subsequence (LCS) between our model output and reference. All this means is that we count the longest sequence of tokens that is shared between both.

BLEU-1 Score:

```
216 77

{'bleu': 0.015075376884422105, 'precisions': [0.01507537688442211], 'brevity_penalty': 1.0, 'length_ratio': 3.015151
----------------------------------------------------------------------------------------------------------------------

output  torch.Size([1, 28])
```

```
result=51.933616906457466
print(result/3000) #3000 is the test set length i.e len(test_loader)
```

```
0.017311205635485823
```

Overall **Bleu-1** score: **0.017311205635485823** over all the samples in the test set

BLEU-2 Score:

```
token  torch.Size([28])
target  torch.Size([28])
```

```

print(result) #3000 is the test set length i.e len(test_loader)
```

```
0.000456754623789
```

Overall **Bleu-2** score: **0.000456754623789**

```

```

3. Implement Model B with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM + global attention based [5 marks]

Define Attention Layers

```
[ ]
#Attention
attn_repeat_layer = RepeatVector(max_len_input)
attn_concat_layer = Concatenate(axis=-1)
attn_dense1 = Dense(3, activation='tanh') #10
attn_dense2 = Dense(1, activation=softmax_over_time)

attn_dot = Dot(axes=1) # to perform the weighted sum of alpha[t] * h[t]
```

```
def one_step_attention(h, st_1):
  # h = h(1), ..., h(Tx), shape = (Tx, LATENT_DIM * 2)
  # st_1 = s(t-1), shape = (LATENT_DIM_DECODER,)

  # copy s(t-1) Tx times
  # now shape = (Tx, LATENT_DIM_DECODER)
  st_1 = attn_repeat_layer(st_1)

  # Concatenate all h(t)'s with s(t-1)
  # Now of shape (Tx, LATENT_DIM_DECODER + LATENT_DIM * 2)
  x = attn_concat_layer([h, st_1])

  # Neural net first layer
  x = attn_dense1(x)

  # Neural net second layer with special softmax over time
  alphas = attn_dense2(x)

  # "Dot" the alphas and the h's
  # Remember a.dot(b) = sum over a[t] * b[t]
  context = attn_dot([alphas, h])

  return context
```

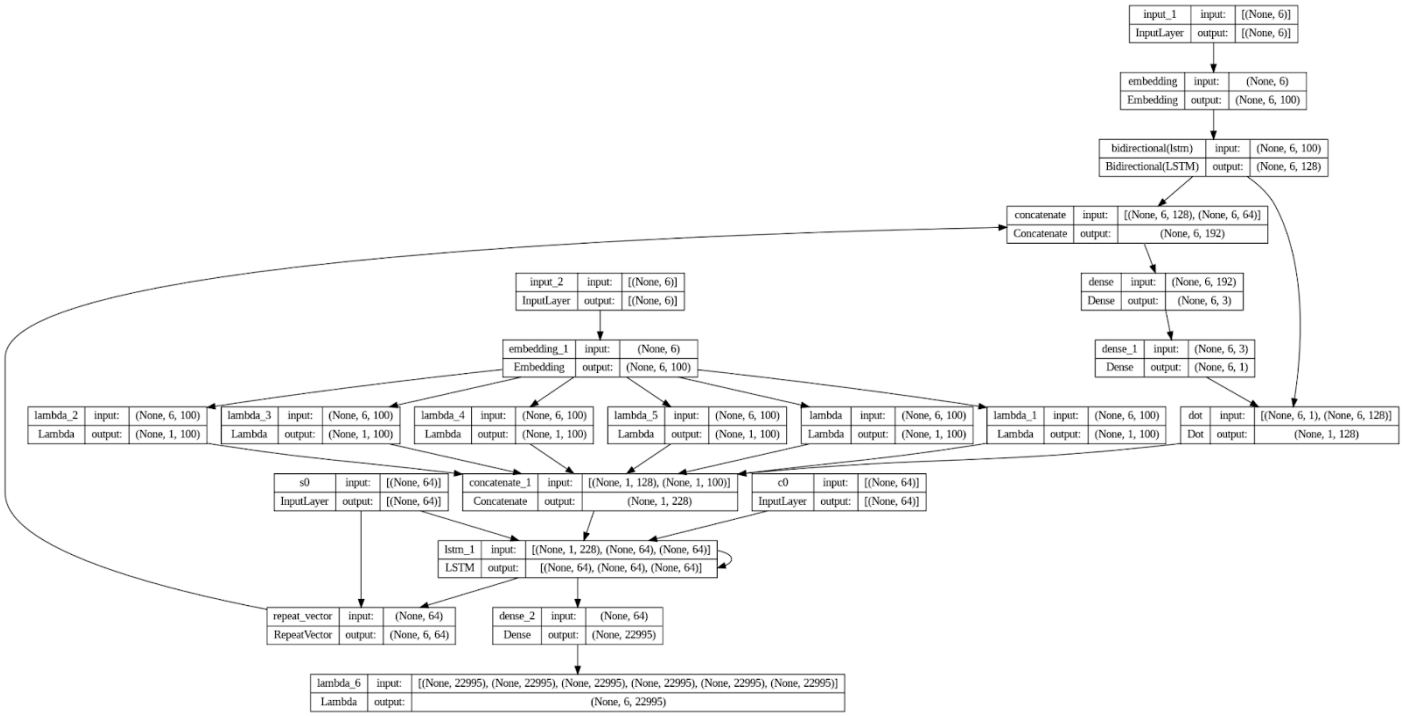
GloVe Embedding

```
#Load pretrained word vectors from Glove
print('Loading word vectors...')
word2vec = {}

#/content/drive/MyDrive/2 Sem/DL/Assignments/A3_BharatGoyal_DebnathKundu_SaloniAgrawal/Q2/glove.6B/glove.6B.50d.txt
#/content/drive/MyDrive/2 Sem/DL/Assignments/A3_BharatGoyal_DebnathKundu_SaloniAgrawal/Q2/subs.de.vec
#/content/drive/MyDrive/2 Sem/DL/Assignments/A3_BharatGoyal_DebnathKundu_SaloniAgrawal/Q2/glove.6B/glove.6B.100d.txt
with open(os.path.join('/content/drive/MyDrive/2 Sem/DL/Assignments/A3_BharatGoyal_DebnathKundu_SaloniAgrawal/Q2/glove.
  # is just a space-separated text file in the format:
  # word vec[0] vec[1] vec[2] ...
  for line in f:
    values = line.split()
    word = values[0]
    vec = np.asarray(values[1:], dtype='float32')
    word2vec[word] = vec
print('Found %s word vectors.' % len(word2vec))
```

```
Loading word vectors...
Found 400000 word vectors.
```

```
[ ]
EMBEDDING_DIM=100
# prepare embedding matrix
```



Rouge-L:

```
1 print(results['rougeL'])
```

```
0.048446715905866675
```

Overall **ROUGE-L** score: **0.048446715905866675** over all the samples in the test set

BLEU-1 Score:

```
Downloading extra modules: 100% 3.34k/3.34k [00:00<0(
{'bleu': 0.011258583014270294, 'precisions': [0.25283144570286475], 'brevity
```

```
[ ] 1 print(results['bleu'])
```

```
0.011258583014270294
```

Overall **Bleu-1** score: **0.011258583014270294** over all the samples in the test set

BLEU-2 Score:

```
print(results['bleu'])
```

```
0.0008950921766039782
```

+ Code + Tex

Overall **Bleu-2** score: **0.0008950921766039782** over all the samples in the test set

4. Implement Model C with: one with non-contextualised initialised embeddings (or random embeddings) + LSTM + local attention based [5 marks]

In this implementation, the *window_size* parameter specifies the size of the local attention window, which is applied to the encoder outputs at each decoder time step.

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense, Dot, Concatenate

# Define the model architecture
def create_model(input_dim, output_dim, hidden_dim, window_size):
    # Define the input sequences
    encoder_inputs = Input(shape=(None, input_dim))
    decoder_inputs = Input(shape=(None, output_dim))

    # Define the LSTM encoder
    encoder_lstm = LSTM(hidden_dim, return_sequences=True, return_state=True)
    encoder_outputs, state_h, state_c = encoder_lstm(encoder_inputs)

    # Define the LSTM decoder
    decoder_lstm = LSTM(hidden_dim, return_sequences=True, return_state=True)
    decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=[state_h, state_c])

    # Define the local attention mechanism
    attention_layer = Dot(axes=[2, 2])
    context_layer = Concatenate(axis=2)
    attention_weights = []
    for t in range(output_dim):
        attention_input = decoder_outputs[:, t:t+window_size, :]
        attention_score = attention_layer([attention_input, encoder_outputs])
        attention_weight = Dense(1, activation='softmax')(attention_score)
        attention_weights.append(attention_weight)
        context_vector = Dot(axes=[1, 1])([attention_weight, encoder_outputs])
        decoder_outputs = context_layer([decoder_outputs, context_vector])

    # Define the output layer
    output_layer = Dense(output_dim, activation='softmax')
```

The attention mechanism calculates attention scores for each encoder output within the window, and then computes a weighted average of the encoder outputs based on these scores to obtain a context vector, which is used to modulate the decoder output.

Additionally, the model uses a softmax activation function for the output layer, which assumes that the output sequence is a probability distribution over the possible output symbols at each time step.

ROUGE-L Score:

```
results = r.compute(predictions=test_predicted_sentence, references=test_actual_senten
print(results)
```

```
Downloading builder script: 100% 6.27k/6.27k [00:00<00:00, 198kB/s]
{'rouge1': 0.011500000563772626, 'rouge2': 0.0, 'rougeL': 0.011367996202336292, 'rougeLsum'
```

```
print(results['rougeL'])
```

```
0.011367996202336292
```

+ Code + Text

Overall **ROUGE-L** score: **0.011367996202336292** over all the samples in the test set

BLEU-1 Score:

```
results = bleu.compute(predictions=test_predicted_sentence, references=test_actual_sentence,max
print(results)
```

```
Downloading builder script: 100% 5.94k/5.94k [00:00<00:00, 301kB/s]
Downloading extra modules: 4.07k/? [00:00<00:00, 252kB/s]
Downloading extra modules: 100% 3.34k/3.34k [00:00<00:00, 218kB/s]
{'bleu': 0.018143007548993544, 'precisions': [0.2340057636887608], 'brevity_penalty': 0.077532310584
```

```
print(results['bleu'])
```

```
0.018143007548993544
```

Overall **Bleu-1** score: **0.018143007548993544** over all the samples in the test set

BLEU-2 Score:

```
results = bleu.compute(predictions=test_predicted_sentence, references=test_actua
print(results)
```

```
{'bleu': 0.0013761781996548899, 'precisions': [0.2340057636887608, 0.00134634803096600
```

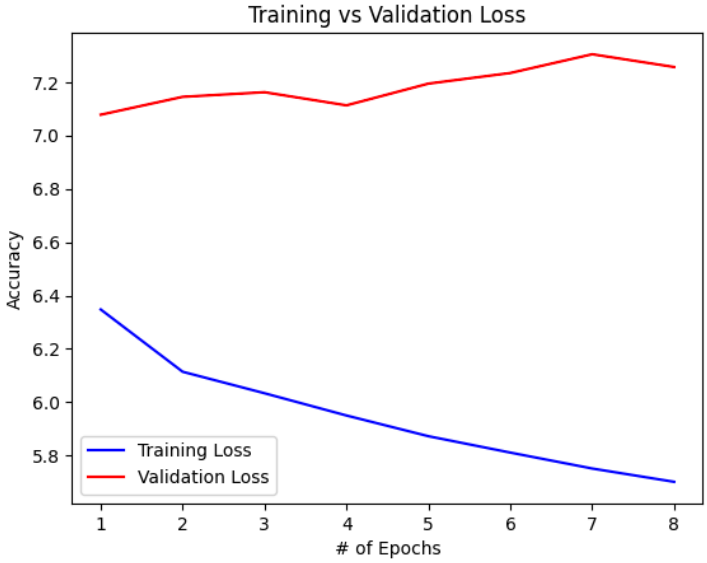
```
[60] print(results['bleu'])
```

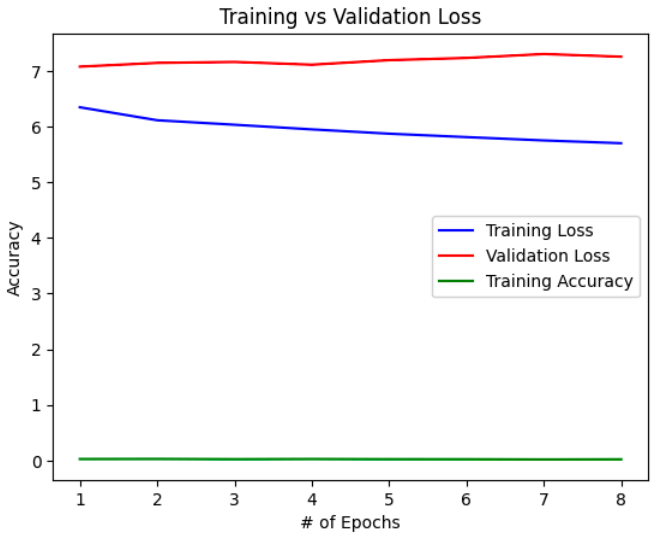
```
0.0013761781996548899
```

Overall **Bleu-2** score: **0.0013761781996548899** over all the samples in the test set

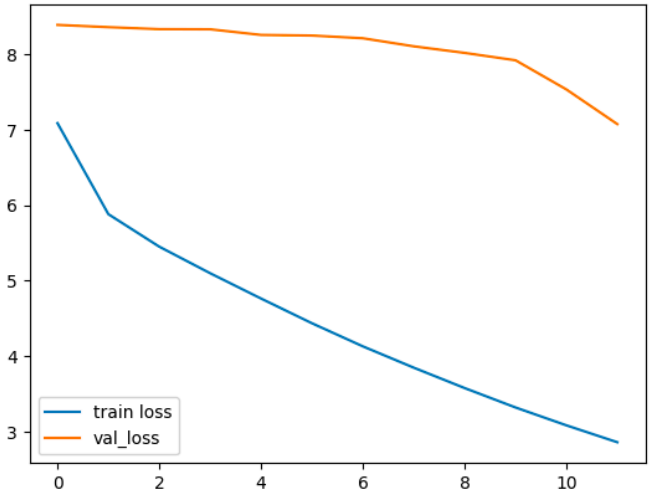
5. Show plots of validation and training loss vs. number of epochs for Models A, B, C. [3 marks]

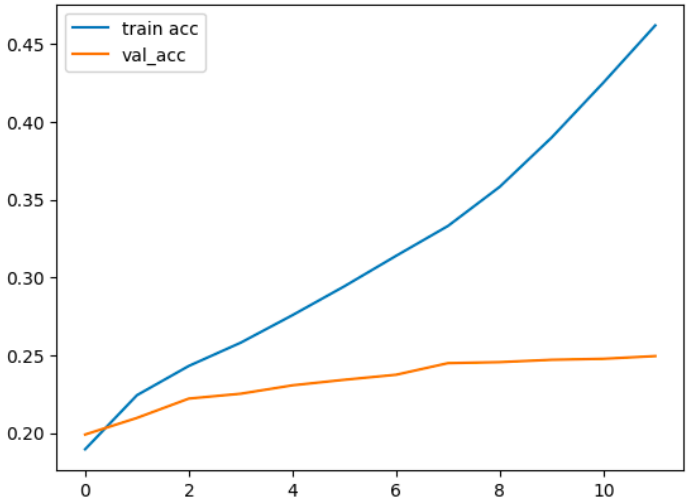
Model A:



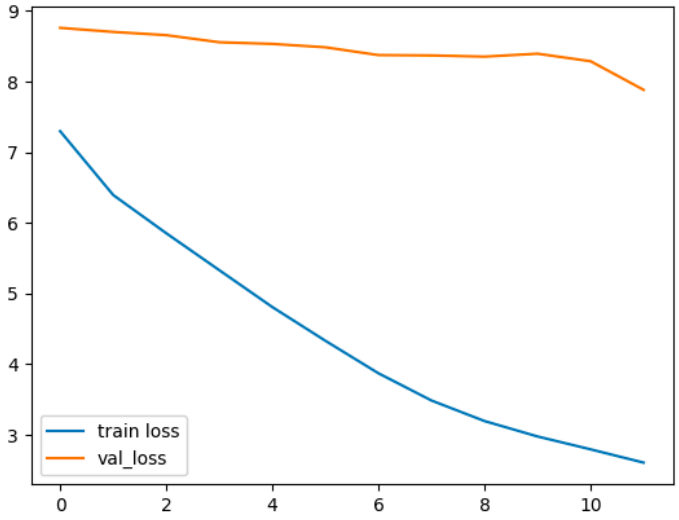


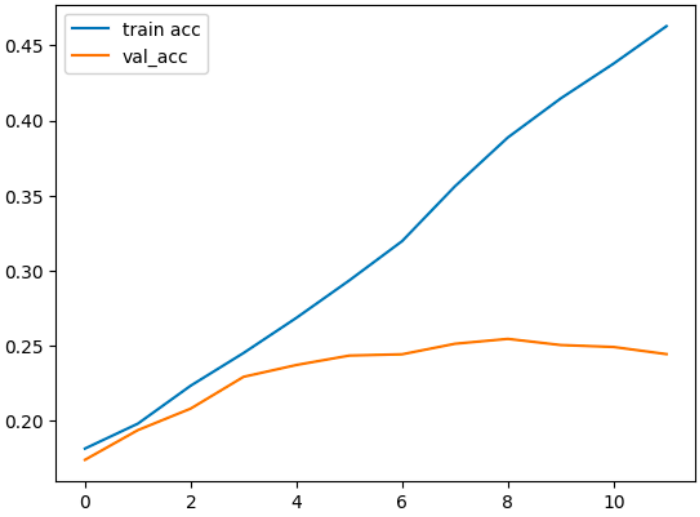
Model B:





Model C:





5. Report the overall Rouge-L score, Bleu-1,2 score for the test set. (metrics available in Huggingface metric) for Models A, B, C. [4 marks]

Model A:

Rouge-L score: `0.021242139635990323`
Bleu-1 score: `0.017311205635485823`
Bleu-2 score: `0.000456754623789`

Model B:

Rouge-L score: `0.048446715905866675`
Bleu-1 score: `0.011258583014270294`
Bleu-2 score: `0.0008950921766039782`

Model C:

Rouge-L score: `0.011367996202336292`
Bleu-1 score: `0.018143007548993544`
Bleu-2 score: `0.0013761781996548899`

6. Out of the 3 models, which setup performed best and why? [2 marks]

In general, **Bleu measures precision**: how much the words (and/or n-grams) in the machine generated summaries appeared in the human reference summaries. **Rouge measures recall**: how much the words (and/or n-grams) in the human reference summaries appeared in the machine generated summaries.

Naturally, these results are complementing, as is often the case in precision vs recall. If we have many words from the system results appearing in the human references we will have **high Bleu**, and if we have many words from the human references appearing in the system results, we will have **high Rouge**.

Bleu uses **brevity penalty** which is quite important and has already been added to standard Bleu implementations. It penalizes system results which are *shorter* than the general length of a reference (read more about it here). This complements the *n-gram* metric behavior which in effect penalizes longer than reference results, since the denominator grows the longer, the system result is.

Model A uses vanilla LSTM, while Model B & C uses LSTM + Attention. So, Model B & C performs better than Model A. As discussed in the original research paper (Attention is All you Need), local attention performs better than global attention, as local context is more important than global context while predicting words in machine translation. However, according to our score **model B** has the **best ROUGE scores.**

Task 3 (Transformer) [15 marks]

Dataset: WNT16 (german-english) dataset. https://huggingface.co/datasets/wmt16/viewer/de-en
Loading the model using huggingface: load_dataset('wmt16', 'de-en'). We use the same dataset as in

Task 2, but this time we fine-tune transformer based model. Refer tutorial:
https://huggingface.co/docs/transformers/tasks/translation

1. Atleast one layer of your fine-tuined model should be set to trainable. You are free to employ any model (even distilled ones.). Huggingface models already tuned on WNT16 dataset cannot be used. Special case should be taken for overfitting, include appropriate measures if overfitting is observed. [10 marks]

2. Show plots of validation and training loss vs. number of epochs for the model. [1 marks]

3. Report the overall Rouge-L score, Bleu-1,2 score for the test set. (metrics available in Huggingface metric) for the model. [2 marks]

4. What can be said about the quality of output generated by models in Task 2 vs Task 3. [2 marks]

Task 3(Transformer)

1. I have trained the model on 0.1% of the train set as it was huge to process. I have used the **Tritkoman/GermantoHusnrikv1** tokenizer to process the german-english pairs. Tokenizing is done separately for source and target text.

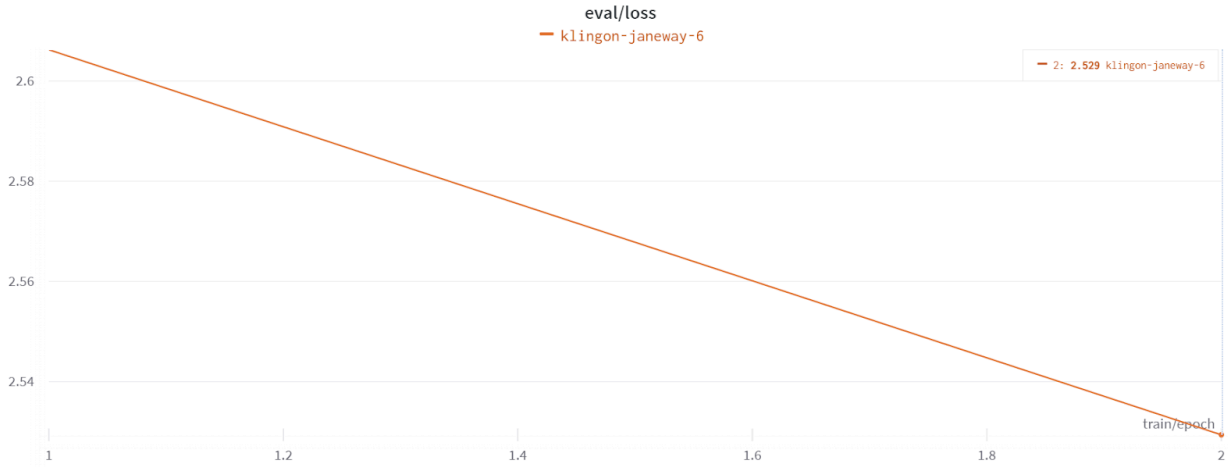
```
tokenizer = AutoTokenizer.from_pretrained("Tritkoman/GermantoHunsrikV1")
model = AutoModelForSeq2SeqLM.from_pretrained("Tritkoman/GermantoHunsrikV1")
```

I have set the output layer of the decoder to trainable.

```
model.decoder.block[-1].layer.require_grad=True
```

Using the data collector, I have dynamically padded all the text to the maximum length of the text.
Then,
The model is trained by taking batch size 2, number of epochs as 2, weight decay of 0.01, and a learning rate of 2e-5.

2. For plotting the graph, I have used the Wandb interface.
For



In both graphs, we can see that the loss decreases as the number of epochs increases.

3. On the test set,
The Rouge L score is `0.32989093702531636`
Bleu scores:
`'bleu1': 0.38179239938990034`
`'bleu2': 0.23542539798717207`
Bleu's score and the rouge's score are lower. That is different from the reference text. The reason can be the training on fewer train sets.

4. Task 3, which uses transformers, performs better than Task 2. Task 3 shows higher Rouge L, Blue 1, and Blue 2 scores than all three models of Task 2. This can be understood by:- In task 3, we are using a fine tune model along with one layer set to pre-train. In task 2, all the models are built from scratch; thus, performance for task 3 will be better as compared to task 2.