



Building Transformer from scratch and using pre-trained GPT2 Transformer

Natural Language Processing

Presented by:

Habiba Abdullah Said Hammad-2203167

Supervised by:

Dr. Christine Basta

Engineer Jailan Refaat

Introduction

The task I decided to tackle was machine translation , mainly because there were more resources and it looked a little bit more exciting than text summarization , after thorough searching through Kaggle I decided on an English to German dataset as it was a CSV file and wasn't that large.

<https://www.kaggle.com/datasets/devanshusingh/machine-translation-dataset-de-en>

this is the link for the dataset.

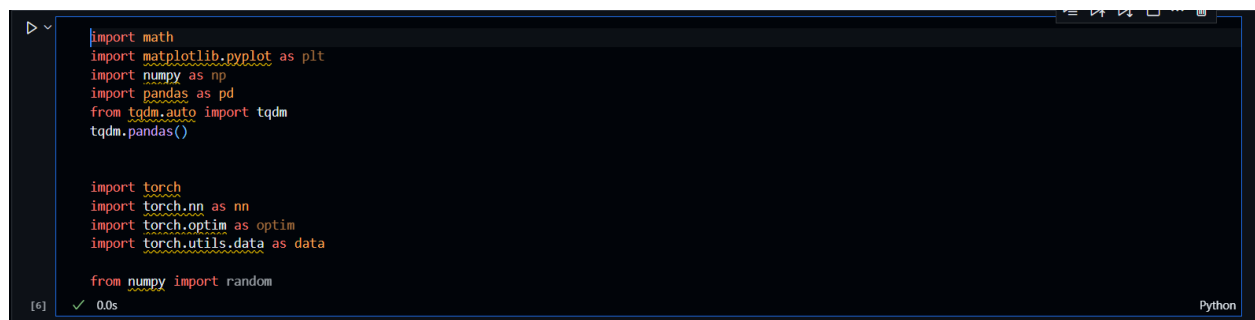
Models

I built 2 models a traditional Seq2Seq model using LSTM (bonus) , a Transformer from scratch and fine tuned GPT2. For all the models Pytorch was used.

Transformer From Scratch (Pytorch)

Imports

- ✓ Math → for math equations
- ✓ Matplotlib.pyplot → for plotting
- ✓ numpy → to deal with arrays
- ✓ Pandas → to deal with csv files
- ✓ Tqdm.auto → for progress bars in loops
- ✓ Tqdm.pandas → for progress bars for pandas data frame
- ✓ Torch, torch.nn, torch.optim, torchdata → to create the custom dataset and some blocks of the transformer

A screenshot of a Jupyter Notebook cell with a dark background. The code is written in a light blue font and includes imports for various libraries. The first block of imports includes math, matplotlib.pyplot (as plt), numpy (as np), pandas (as pd), and tqdm (from tqdm.auto and tqdm.pandas). The second block imports torch, torch.nn (as nn), torch.optim (as optim), and torch.utils.data (as data). The third block imports random from numpy. The cell is numbered [6] and shows a successful execution with a green checkmark and a time of 0.0s. The word 'Python' is visible in the bottom right corner of the cell's interface.

```
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from tqdm.auto import tqdm
tqdm.pandas()

import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data

from numpy import random
```

[6] ✓ 0.0s Python

Loading the dataset

After loading the dataset, I checked to see if there were duplicates or N/A values. Luckily there were no N/A values but there were 3 duplicates, so they were removed.

```
df = pd.read_csv('D:/College/5th Semester/NLP/Assignmnet 1/translation_train.csv')
df_test = pd.read_csv('D:/College/5th Semester/NLP/Assignmnet 1/translation_test.csv')

print(df.shape)
print(df.isnull().sum())
print(df.duplicated().sum())

...
(29000, 2)
english    0
german     0
dtype: int64
3

print(df_test.shape)
print(df_test.isnull().sum())
print(df_test.duplicated().sum())

...
(1000, 2)
english    0
german     0
dtype: int64
0

print(df_test.shape)
print(df_test.isnull().sum())
print(df_test.duplicated().sum())

...
(1000, 2)
english    0
german     0
dtype: int64
0

print(df.isnull().sum())
print(df.duplicated().sum())

...
english    0
german     0
dtype: int64
3

df.drop_duplicates(inplace=True)

df_test.drop_duplicates(inplace=True)
```

Now we can look at the data.

```
for i in range(15):
    print(df['german'][i+1])
    print(df['english'][i+1])
```

[13] ✓ 0.0s Python

...
Mehrere Männer mit Schutzhelmen bedienen ein Antriebsradsystem.
Several men in hard hats are operating a giant pulley system.
Ein kleines Mädchen klettert in ein Spielhaus aus Holz.
A little girl climbing into a wooden playhouse.
Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein Fenster.
A man in a blue shirt is standing on a ladder cleaning a window.
Zwei Männer stehen am Herd und bereiten Essen zu.
Two men are at the stove preparing food.
Ein Mann in grün hält eine Gitarre, während der andere Mann sein Hemd ansieht.
A man in green holds a guitar while the other man observes his shirt.
Ein Mann lächelt einen ausgestopften Löwen an.
A man is smiling at a stuffed lion
Ein schickes Mädchen spricht mit dem Handy während sie langsam die Straße entlangschwebt.
A trendy girl talking on her cellphone while gliding slowly down the street.
Eine Frau mit einer großen Geldbörse geht an einem Tor vorbei.
A woman with a large purse is walking by a gate.
Jungen tanzen mitten in der Nacht auf Pfosten.
Boys dancing on poles in the middle of the night.
Eine Ballettklasse mit fünf Mädchen, die nacheinander springen.
A ballet class of five girls jumping in sequence.
Vier Typen, von denen drei Hüte tragen und einer nicht, springen oben in einem Treppenhaus.
Four guys three wearing hats one not are jumping at the top of a staircase.
Ein schwarzer Hund und ein gefleckter Hund kämpfen.
A black dog and a spotted dog are fighting
Ein Mann in einer neongrünen und orangefarbenen Uniform fährt auf einem grünen Traktor.
...
Mehrere Frauen warten in einer Stadt im Freien.
Several women wait outside in a city.
Eine Frau mit schwarzen, elasthan und Beile, steht auf einem grünen Traktor.
A woman with black, elasthan and Beile, stands on a green tractor.

Train test split

Here instead of using the norm Train_test_split , the ratio was first calculated then it was cut from the total number of sentences in the train dataset and then shuffled. For the test_df I used the test dataset but not all of it just to conserve power and CPU.

```
train_ratio = 0.8
val_ratio = 0.1
test_ratio = 0.1

num_sentences = len(df)
num_train = int(train_ratio * num_sentences)
num_val = int(val_ratio * num_sentences)
num_test = num_sentences - num_train - num_val

# Shuffle the dataset
df = df.sample(frac=1).reset_index(drop=True)
```

[14] ✓ 0.0s Python

```
train_df = df[:num_train]
val_df = df[num_train:num_train+num_val]
test_df = df_test[:num_test]
```

[15] ✓ 0.0s Python

I then viewed the lengths of the data frames, and they were looking good.

```
[16] len(train_df)
✓ 0.0s Python
... 23197

[17] len(val_df)
✓ 0.0s Python
... 2899

[18] len(test_df)
✓ 0.0s Python
... 1000
```

Tokenization

Here I am first training the sentencepiece tokenizer and getting the vocab sizes and also declaring the tokenizers.

```
import sentencepiece as spm

# Train a SentencePiece tokenizer
def train_sentencepiece(corpus, model_prefix, vocab_size=7500): # Adjust vocab_size
    with open(f"{model_prefix}_corpus.txt", "w") as f:
        f.write("\n".join(corpus))

    spm.SentencePieceTrainer.train(
        input=f"{model_prefix}_corpus.txt",
        model_prefix=model_prefix,
        vocab_size=vocab_size,
        character_coverage=0.9995, # Adjust for language diversity
        pad_id=0, unk_id=1, bos_id=2, eos_id=3
    )
    print(f"SentencePiece model trained for {model_prefix}")

# Train tokenizers for English and German
train_sentencepiece(train_df['english'], model_prefix="english")
train_sentencepiece(train_df['german'], model_prefix="german")

# Load SentencePiece models
en_tokenizer = spm.SentencePieceProcessor(model_file="english.model")
ger_tokenizer = spm.SentencePieceProcessor(model_file="german.model")

# Vocabulary sizes
src_vocab_size = en_tokenizer.get_piece_size()
tgt_vocab_size = ger_tokenizer.get_piece_size()
print(f"Source (English) vocab size: {src_vocab_size}")
print(f"Target (German) vocab size: {tgt_vocab_size}")
```

Here I tokenized the sentences using the tokenizer and used special tokens

```

# Tokenizing function with special tokens
def tokenize_sentence_with_specials(sentence, tokenizer):
    tokens = tokenizer.encode(sentence, out_type=int) # Convert sentence to subword token IDs
    tokens = [tokenizer.bos_id()] + tokens + [tokenizer.eos_id()] # Add BOS and EOS
    return tokens

# Tokenizing training data
print("Tokenizing training data:")
train_en_tokens = [tokenize_sentence_with_specials(sent, en_tokenizer) for sent in train_df['english']]
train_gr_tokens = [tokenize_sentence_with_specials(sent, ger_tokenizer) for sent in train_df['german']]
print("Training Data tokenized")

# Tokenizing validation data
print("Tokenizing validation data:")
val_en_tokens = [tokenize_sentence_with_specials(sent, en_tokenizer) for sent in val_df['english']]
val_gr_tokens = [tokenize_sentence_with_specials(sent, ger_tokenizer) for sent in val_df['german']]
print("Validation Data tokenized")

# Tokenizing validation data
print("Tokenizing Test data:")
test_en_tokens = [tokenize_sentence_with_specials(sent, en_tokenizer) for sent in test_df['english']]
test_gr_tokens = [tokenize_sentence_with_specials(sent, ger_tokenizer) for sent in test_df['german']]
print("Test Data tokenized")

```

Dataset and Data loader

By creating a custom data set , I made sure to pad the data with zeros and also had a function of `__len__()` to see the length of the tokens (note that English and german tokens are of same length).

The datasets were then created and the data loaders followed.

```
class TranslationDataset(data.Dataset):
    def __init__(self, en_tokens, ger_tokens):
        self.en_tokens = en_tokens
        self.ger_tokens = ger_tokens
        self.max_len = max(max(len(en), len(gr)) for en, gr in zip(en_tokens, ger_tokens))

    def __len__(self):
        return len(self.en_tokens)

    def __getitem__(self, index):
        en_data = self.en_tokens[index] + [0] * (self.max_len - len(self.en_tokens[index])) # Padding with 0
        ger_data = self.ger_tokens[index] + [0] * (self.max_len - len(self.ger_tokens[index])) # Padding with 0
        return torch.tensor(en_data), torch.tensor(ger_data)

[22] ✓ 0.0s Python

train_dataset = TranslationDataset(train_en_tokens, train_gr_tokens)
val_dataset = TranslationDataset(val_en_tokens, val_gr_tokens)
test_dataset = TranslationDataset(test_en_tokens, test_gr_tokens)

# Create data loaders
train_loader = data.DataLoader(train_dataset, batch_size=2, shuffle=True)
val_loader = data.DataLoader(val_dataset, batch_size=2)
test_loader = data.DataLoader(test_dataset, batch_size=2)
print("Dataset and Dataloders created")

[23] ✓ 0.0s Python

... Dataset and Dataloders created
```

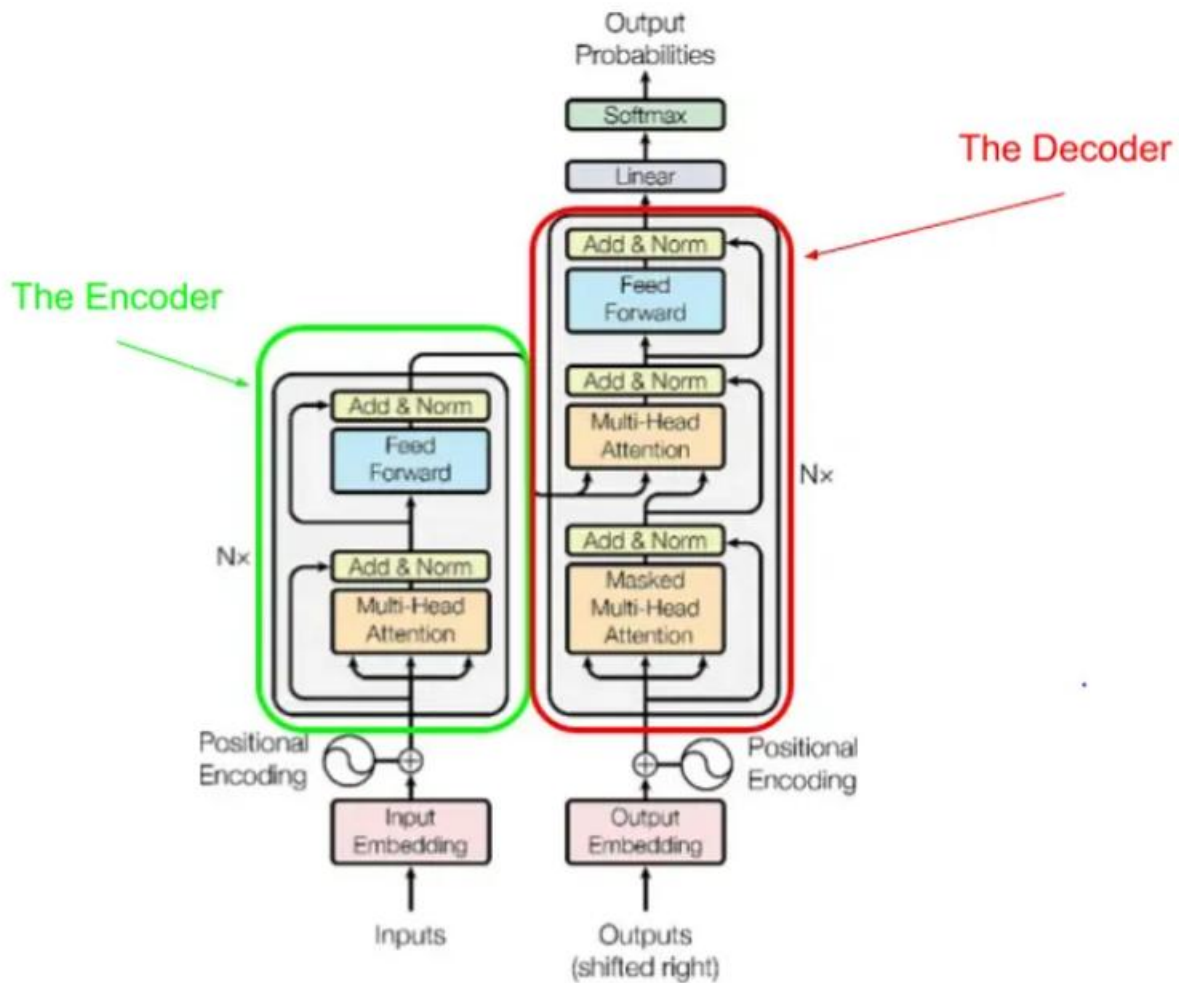
Finally, to validate the pre-processing, the batches were printed. Now we are ready to build the transformer.

```
# Print the first batch of data from train_loader
for batch_idx, (en_data, ger_data) in enumerate(train_loader):
    print("English Data (Batch 0):", en_data)
    print("German Data (Batch 0):", ger_data)
    break # Break after printing the first batch to avoid printing the entire dataset

[24] ✓ 0.0s Python

... English Data (Batch 0): tensor([[ 1,  2, 133,  4,  5, 3270, 61,  5, 872, 1060,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0],
 [ 47, 13, 3089, 240,  5, 2866,  4, 300, 36,  5, 512, 149,
 36, 3509,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0]])
... German Data (Batch 0): tensor([[ 21, 283, 157,  4, 761, 61, 26,  5, 12348,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0],
 [ 47, 13, 385, 125, 16, 49, 14801, 32, 14802,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0]])
```


Transformer: Attention is all you need



Position Encoding and Position Wise Feed Forward

Here this class is only for a feedforward network consisting of 2 linear layers and then a relu activation function

```
class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

[26] ✓ 0.0s Python

Here this class is to create the positional encoding of the input tensors. It first initiates a matrix by 0 for size maximum sequence length and dimension of the model (embedding dimension). The position is then calculated using the a range function so it has the correct position and then a `div_term` is to create a frequency scaling mechanism, which is used to generate positional encodings using the sinusoidal functions. To help with distinct positional information if the index is even , the sin is calculated else the cosine is calculated and then it is saved. The forward function adds the embedding of x to the positional encoding.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
        # Frequency Scaling mechanism
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

Multi-Head Attention

It takes parameters as the `d_model` and the num of heads , we first check if the dimension of the model is divisible by the number of heads using the `assert` function and after initialization , the `Wq`, `Wk` and `Wv` are initialized as linear layers and `debug_str` is for debugging purposes. If we then look at the dot product of the attention function , first the attention score is calculated using matrix multiplication (`matmul`) of the Q and K transpose and then divided by the dimension of k , a debug if statement is there. If there is a mask then the mask is filled with a very large negative number which after SoftMax function would be 0 so that the model doesn't pay attention to the padding, the probabilities are then calculated using the SoftMax function with `dim=-1` to be applied to the last dimension of the tensor then the output is just a matrix multiplication of the value and the attention probabilities and then returned

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, debug_str = None):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)
        self.debug_str = debug_str

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if (self.debug_str == 'cross'):
            print('attn_scores:', attn_scores.shape, mask.shape)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
```

The function `split_heads` is used to split the attention outputs in the multi-head attention to smaller tensors so that attention is applied to each head or small tensor. and the `combine_heads` is used to combine the outputs from the multiple attention heads back into one tensor. The forward function then calculates the Q,K, and V and then performs scaled dot product attention and then multiplies it

```

return output
def split_heads(self, x):
    batch_size, seq_length, d_model = x.size()
    return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

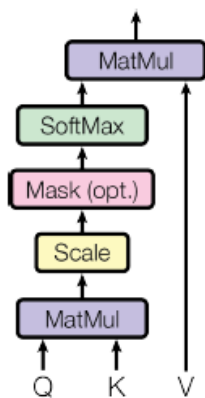
def combine_heads(self, x):
    batch_size, _, seq_length, d_k = x.size()
    return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

def forward(self, Q, K, V, mask=None):
    Q = self.split_heads(self.W_q(Q))
    K = self.split_heads(self.W_k(K))
    V = self.split_heads(self.W_v(V))

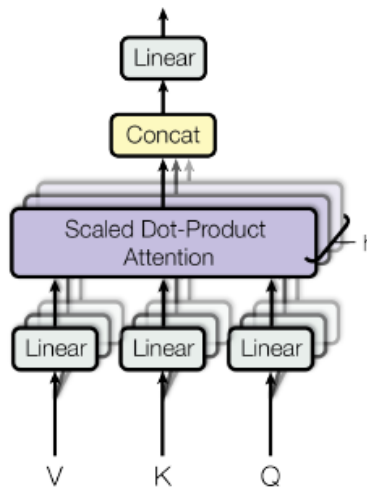
    attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
    output = self.W_o(self.combine_heads(attn_output))
    return output

```

Scaled Dot-Product Attention



Multi-Head Attention



Encoder Layer

For the encoder layer all it needs is the Multi-Head Attention layer, feedforward layer, 2 normalization layers and a dropout layer for regularization. And the forward pass just passes through all the layers in order.

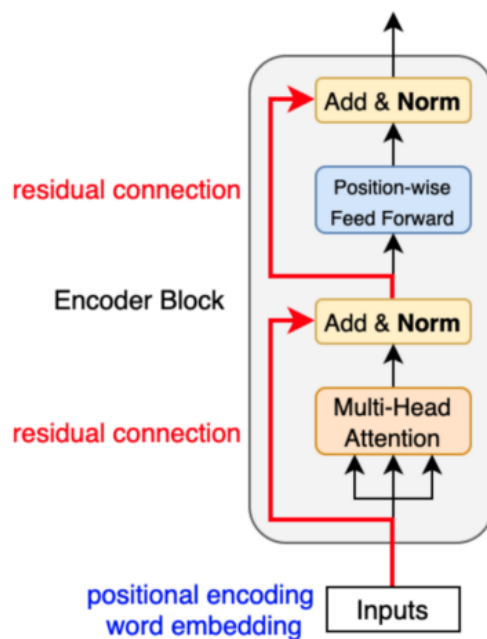
```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

[31]

✓ 0.0s

Python



Decoder Layer

The decoder layer also has 2 attention layers but one is cross-attention which computes attention between the source(encoder) and the target (decoder) sequences and the other is self-attention. We also have 3 normalization layers for residual connections and a dropout layer. for the forward function we first calculate the self-attention where x is the query, key and value at the same time and we use the target mask so it becomes a masked multi-head attention layer. we then pass through a normalization layer before passing through the self-attention layer and after another normalization layer the output is run through a feedforward neural network and then passed through the last normalization layer.

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)#, debug_str="cross")
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

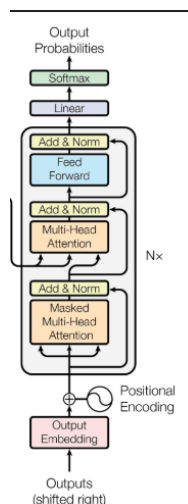
    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))

        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(attn_output))

        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
```

[33] ✓ 0.0s

Python



Full transformer model

Def __init__(self,src_vocab_size,trg_vocab_size,d_model,num_heads,num_layers,d_ff,max_seq_length,dropout,pad_token_src,pad_token_trg,device)

Here what we are passing are the vocab sizes , the hyper-parameters for the transformer and the pad tokens numbers (which is equal to 0)

We then start by initializing the encoder and decoder embeddings and then the positional embeddings, then we define the encoder and decoder layers followed by a fully connected linear layer and a dropout layer

Def generate_mask(self,src_mask,trg_mask)

This function is used to control the flow of information so it doesn't peek to future words. The source mask ensures the model doesn't pay attention to padding tokens and the target mask prevents the model from attending to future tokens and is then modified to include the padding tokens. So two types of masking were used , padding mask in both the encoder and decoder and the casual mask which is only found in the decoder during the self-attention task.

```
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout, pad_token_src = 0, pad_token_tgt = 0):
        super(Transformer, self).__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)

        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])

        self.fc = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

        self.pad_token_src = pad_token_src
        self.pad_token_tgt = pad_token_tgt
        self.device = device
        self = self.to(self.device)

    def generate_mask(self, src_mask, tgt_mask):
        src_mask = src_mask.unsqueeze(1).unsqueeze(2)
        tgt_mask = tgt_mask.unsqueeze(1).unsqueeze(3)
        seq_length = tgt_mask.size(2)
        nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length), diagonal=1)).bool()
        tgt_mask = tgt_mask & nopeak_mask.to(self.device)
        return src_mask, tgt_mask
```

Def decode(self,src,ger_token_id,eos_token_id,mask=None,max_dec_length=25)

Here the target is first initialized with the start of sequencey tokens. The mask are then generated

If no masks are produced the method generates the mask using the generate_mask() function discussed earlier. The source is then passed to the encoder for the number of encoder layers and then the target is equalized to output_labels , the unfinished sequences are sequences without the end of sentence token. And then a while loop keeps looping until either the number of layers are more then the maximum number of decoder layers or the unfinished sequences have been resolved.

```

def decode(self, src, bos_token_id, eos_token_id, mask=None, max_dec_length=50):
    # Initialize target with bos_token_id
    tgt = torch.tensor([[bos_token_id]] * src.shape[0]).to(self.device)

    if mask:
        src_mask = mask.get('src_mask')
        tgt_mask = mask.get('tgt_mask', self.generate_mask(tgt.size(1))) # Default to causal mask if missing
    else:
        src_mask = src != self.pad_token_src
        tgt_mask = self.generate_mask(tgt.size(1))

    src_mask, tgt_mask = self.generate_mask(src_mask, tgt_mask)

    for _ in range(max_dec_length):
        # Pass source and target through the transformer layers
        logits = self.forward(src, tgt, src_mask, tgt_mask)

        # Select the most probable token for the current step
        next_token = logits.argmax(-1)[0, -1].unsqueeze(1)

        # Append the selected token to the target sequence
        tgt = torch.cat([tgt, next_token], dim=1)

        # Break if eos_token_id is generated
        if torch.any(next_token == eos_token_id):
            break

    return tgt

```

Def forward()

Here this is just the forward propagation by first generating the masks and then getting the source and target embeddings by following the dropout layer of the positional encoding layer of the encoder embedding of the source/target. Then the output is passed through the encoder and decoder layers and finally a fully connected layer.

```

def forward(self, src, tgt, mask = None):
    if mask:
        src_mask, tgt_mask = self.generate_mask(mask['src_mask'], mask['tgt_mask'])
    else:
        src_mask, tgt_mask = self.generate_mask(src!=self.pad_token_src, tgt!=self.pad_token_tgt)

    src_embedded = self.dropout(self.positional_encoding(self.encoder_embedding(src)))
    tgt_embedded = self.dropout(self.positional_encoding(self.decoder_embedding(tgt)))

    enc_output = src_embedded
    for enc_layer in self.encoder_layers:
        enc_output = enc_layer(enc_output, src_mask)

    dec_output = tgt_embedded
    for dec_layer in self.decoder_layers:
        dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

    output = self.fc(dec_output)
    return output

```

✓ 0.0s

Python

Hyper-parameters and initialization of model

Here the device is initialized as cuda , the source and target vocabulary sizes are declared. Then the hyper-parameters are initialized along with the model. We are also using Cross Entropy Loss and Adam optimizer.

```

device = 'cuda' if torch.cuda.is_available() else 'cpu'
# Hyperparameters
src_vocab_size = len(en_vocab)
tgt_vocab_size = len(ger_vocab)
# Reduce Model Size
d_model = 64 # Decrease the model dimensionality
num_heads = 2 # Decrease the number of attention heads
num_layers = 2 # Decrease the number of layers
d_ff = 512 # Decrease the size of the feed-forward layers
max_seq_length = max(train_dataset.max_len, val_dataset.max_len, test_dataset.max_len) # Maximum sequence length
dropout = 0.1 # Dropout probability
num_epochs=10 # Number of epochs
# Instantiate the Transformer model
transformer_model = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout, device = device)

```

37] ✓ 0.1s Python

```

optimizer=torch.optim.Adam(transformer_model.parameters()),lr=0.001)
criterion=nn.CrossEntropyLoss()

```

[39] ✓ 0.9s Python

Training Loop

Note that the training and validation loop are both in the same cell so I imported rouge_scorer so that we can find the rouge score for the validation dataset before the test dataset.

```

from rouge_score import rouge_scorer
from tqdm import tqdm
import torch
import torch.nn as nn

# Early stopping parameters
patience = 3 # Number of epochs to wait for improvement before stopping
best_val_loss = float('inf')
epochs_without_improvement = 0

# Create a ROUGE scorer
scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)

```

```

# Training loop
for epoch in range(num_epochs):
    transformer_model.train() # Set the model to training mode
    total_loss = 0

    # Create a progress bar for training
    progress_bar = tqdm(enumerate(train_loader), total=len(train_loader), desc=f"Epoch {epoch+1}/{num_epochs}", unit="batch")

    # Iterate through batches for training
    for batch_idx, (src, tgt) in progress_bar:
        src, tgt = src.to(device), tgt.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        output = transformer_model(src, tgt[:, :-1]) # Exclude the <eos> token from input
        # Flatten the output and target tensors to compute loss
        output_flat = output.view(-1, output.size(-1))
        tgt_flat = tgt[:, 1:].contiguous().view(-1) # Exclude the <bos> token from target

        # Calculate loss
        loss = criterion(output_flat, tgt_flat)

        # Backward pass
        loss.backward()

        # Clip gradients to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(transformer_model.parameters(), max_norm=1)

        # Update parameters
        optimizer.step()

```



```

# Add batch loss to total loss
total_loss += loss.item()

# Update progress bar description
progress_bar.set_postfix({"Loss": loss.item()})

# Calculate average loss for the epoch
avg_loss = total_loss / len(train_loader)
print(f"Epoch {epoch + 1}/{num_epochs}, Average Loss: {avg_loss:.4f}")

```

Validation loop

```

transformer_model.eval() # Set the model to evaluation mode
val_loss = 0
val_rouge_scores = {'rouge1': [], 'rouge2': [], 'rougeL': []}

with torch.no_grad():
    # Create a progress bar for validation
    val_progress_bar = tqdm(enumerate(val_loader), total=len(val_loader), desc="Validation", unit="batch")

    for batch_idx, (src, tgt) in val_progress_bar:
        src, tgt = src.to(device), tgt.to(device)

        # Forward pass
        output = transformer_model(src, tgt[:, :-1]) # Exclude the <eos> token from input

        # Flatten the output and target tensors to compute loss
        output_flat = output.view(-1, output.size(-1))
        tgt_flat = tgt[:, 1:].contiguous().view(-1) # Exclude the <bos> token from target

        # Calculate loss
        loss = criterion(output_flat, tgt_flat)

        # Add batch loss to total validation loss
        val_loss += loss.item()

        # Decode the predictions to text (argmax over the vocab dimension)
        pred_tokens = output.argmax(dim=-1) # Taking the predicted tokens (removes vocab dimension)

        # Loop through each pair of predicted and true targets for ROUGE calculation
        for pred, true in zip(pred_tokens, tgt):
            pred_text = ger_tokenizer.decode(pred.tolist())
            true_text = ger_tokenizer.decode(true.tolist())

            score = scorer.score(true_text, pred_text)
            for key in val_rouge_scores:
                val_rouge_scores[key].append(score[key].fmeasure)

    # Update progress bar description
    val_progress_bar.set_postfix({"Validation loss": loss.item()})

# Calculate average validation loss
avg_val_loss = val_loss / len(val_loader)
print(f"Validation Loss: {avg_val_loss:.4f}")

# Calculate the average ROUGE scores
avg_rouge_scores = {key: sum(values) / len(values) for key, values in val_rouge_scores.items()}
print(f"Validation ROUGE Scores: {avg_rouge_scores}")

# Early stopping logic
if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    epochs_without_improvement = 0 # Reset the counter
    # Save the model
    torch.save(transformer_model.state_dict(), 'transformer_model.pth')
    print(f"Model saved at epoch {epoch+1}")
else:
    epochs_without_improvement += 1

# Check for early stopping
if epochs_without_improvement >= patience:
    print(f"Early stopping at epoch {epoch+1}. Validation loss did not improve.")
    break

```

```
Epoch 1/10: 100%|██████████| 11599/11599 [07:03<00:00, 27.38batch/s, Loss=0.881]
Epoch 1/10, Average Loss: 1.0510
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 103.38batch/s, Validation Loss=0.906]
Validation Loss: 0.9542
Validation ROUGE Scores: {'rouge1': 0.43283835715091595, 'rouge2': 0.19386879035341922, 'rougeL': 0.4162730873912906}
Model saved at epoch 1
Epoch 2/10: 100%|██████████| 11599/11599 [07:20<00:00, 26.34batch/s, Loss=0.701]
Epoch 2/10, Average Loss: 0.8318
Validation: 100%|██████████| 1450/1450 [00:13<00:00, 105.22batch/s, Validation Loss=0.785]
Validation Loss: 0.8407
Validation ROUGE Scores: {'rouge1': 0.5023131110557396, 'rouge2': 0.2570274930347473, 'rougeL': 0.4851970226480362}
Model saved at epoch 2
Epoch 3/10: 100%|██████████| 11599/11599 [07:20<00:00, 26.32batch/s, Loss=0.802]
Epoch 3/10, Average Loss: 0.7503
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 100.94batch/s, Validation Loss=0.729]
Validation Loss: 0.7880
Validation ROUGE Scores: {'rouge1': 0.5347638125299945, 'rouge2': 0.28935758130210404, 'rougeL': 0.5160682260691485}
Model saved at epoch 3
Epoch 4/10: 100%|██████████| 11599/11599 [07:26<00:00, 25.99batch/s, Loss=0.156]
Epoch 4/10, Average Loss: 0.6993
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 101.25batch/s, Validation Loss=0.718]
Validation Loss: 0.7479
Validation ROUGE Scores: {'rouge1': 0.5607894043232357, 'rouge2': 0.32053016506502635, 'rougeL': 0.5438058675530372}
Model saved at epoch 4
Epoch 5/10: 100%|██████████| 11599/11599 [07:39<00:00, 25.23batch/s, Loss=0.249]
Epoch 5/10, Average Loss: 0.6625
Validation: 100%|██████████| 1450/1450 [00:13<00:00, 107.01batch/s, Validation Loss=0.637]
Validation Loss: 0.7135
```

Show hidden icons

```
Epoch 6/10: 100%|██████████| 11599/11599 [07:46<00:00, 24.86batch/s, Loss=1.1]
Epoch 6/10, Average Loss: 0.6332
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 102.57batch/s, Validation Loss=0.759]
Validation Loss: 0.6885
Validation ROUGE Scores: {'rouge1': 0.5894155829211623, 'rouge2': 0.3569996679065852, 'rougeL': 0.5734363165611484}
Model saved at epoch 6
Epoch 7/10: 100%|██████████| 11599/11599 [07:35<00:00, 25.47batch/s, Loss=0.897]
Epoch 7/10, Average Loss: 0.6103
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 97.45batch/s, Validation Loss=0.673]
Validation Loss: 0.6725
Validation ROUGE Scores: {'rouge1': 0.5962022664634847, 'rouge2': 0.362150133390866, 'rougeL': 0.5807996008775139}
Model saved at epoch 7
Epoch 8/10: 100%|██████████| 11599/11599 [21:46<00:00, 8.88batch/s, Loss=0.606]
Epoch 8/10, Average Loss: 0.5922
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 101.19batch/s, Validation Loss=0.643]
Validation Loss: 0.6596
Validation ROUGE Scores: {'rouge1': 0.605010966631962, 'rouge2': 0.3746338651756154, 'rougeL': 0.5899880129688385}
Model saved at epoch 8
Epoch 9/10: 100%|██████████| 11599/11599 [09:41<00:00, 19.94batch/s, Loss=0.724]
Epoch 9/10, Average Loss: 0.5757
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 102.56batch/s, Validation Loss=0.695]
Validation Loss: 0.6558
Validation ROUGE Scores: {'rouge1': 0.6099110882217844, 'rouge2': 0.3763702754133648, 'rougeL': 0.5932270568663334}
Model saved at epoch 9
Epoch 10/10: 100%|██████████| 11599/11599 [11:07<00:00, 17.38batch/s, Loss=0.487]
Epoch 10/10, Average Loss: 0.5644
Validation: 100%|██████████| 1450/1450 [00:14<00:00, 103.08batch/s, Validation Loss=0.624]
Validation Loss: 0.6454
Validation ROUGE Scores: {'rouge1': 0.6123742827673571, 'rouge2': 0.3820343653436769, 'rougeL': 0.5971016769158085}
Model saved at epoch 10
```

Testing (Inference)

```
# Testing
from rouge_score import rouge_scorer
from tqdm import tqdm
import torch
import torch.nn as nn

transformer_model.eval() # Set the model to evaluation mode
test_loss = 0
test_rouge_scores = {'rouge1': [], 'rouge2': [], 'rougeL': []}
# Create a ROUGE scorer
scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)
with torch.no_grad():
    # Create a progress bar for validation
    test_progress_bar = tqdm(enumerate(test_loader), total=len(test_loader), desc="Test", unit="batch")

    for batch_idx, (src, tgt) in test_progress_bar:
        src, tgt = src.to(device), tgt.to(device)

        # Forward pass
        output = transformer_model(src, tgt[:, :-1]) # Exclude the <eos> token from input

        # Flatten the output and target tensors to compute loss
        output_flat = output.view(-1, output.size(-1))
        tgt_flat = tgt[:, 1:].contiguous().view(-1) # Exclude the <bos> token from target

        # Calculate loss
        loss = criterion(output_flat, tgt_flat)

        # Add batch loss to total validation loss
        test_loss += loss.item()
```

```
# Decode the predictions to text (argmax over the vocab dimension)
pred_tokens = output.argmax(dim=-1) # Taking the predicted tokens (removes vocab dimension)

# Loop through each pair of predicted and true targets for ROUGE calculation
for pred, true in zip(pred_tokens, tgt):
    pred_text = ger_tokenizer.decode(pred.tolist())
    true_text = ger_tokenizer.decode(true.tolist())

    # Compute ROUGE score for each pair and append the F-measure values
    score = scorer.score(true_text, pred_text)
    for key in test_rouge_scores:
        test_rouge_scores[key].append(score[key].fmeasure)

# Update progress bar description
test_progress_bar.set_postfix({"Test Loss": loss.item()})

# Calculate average validation loss
avg_loss = test_loss / len(test_loader)
print(f"Test Loss: {avg_loss:.4f}")

# Calculate the average ROUGE scores
avg_rouge_scores = {key: sum(values) / len(values) for key, values in test_rouge_scores.items()}
print(f"Test ROUGE Scores: {avg_rouge_scores}")
```

```
... Test: 100%|██████████| 500/500 [00:04<00:00, 108.20batch/s, Test Loss=0.216]
Test Loss: 0.7769
Test ROUGE Scores: {'rouge1': 0.6174154518047238, 'rouge2': 0.38755095282171126, 'rougeL': 0.6033373757267394}
```

The model here has produced great results and can understand long sequences efficiently but needs some training when it comes to word pair relationships and context.

Transformer GPT-2

Essentially, I did the same pre-processing but with different approaches.

```
[43] import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torch.nn import LayerNorm
from itertools import chain
import numpy as np
import spacy
import random
import pandas as pd
import matplotlib.pyplot as plt

✓ 0.0s Python
```

```
[44] df=pd.read_csv("D:/College/5th Semester/NLP/Assignmnet 1/translation_train.csv")
df.head()

✓ 0.1s Python
```

	english	german
0	Two young. White males are outside near many b...	Zwei junge weiße Männer sind im Freien in der ...
1	Several men in hard hats are operating a giant...	Mehrere Männer mit Schutzhelmen bedienen ein A...
2	A little girl climbing into a wooden playhouse.	Ein kleines Mädchen klettert in ein Spielhaus ...
3	A man in a blue shirt is standing on a ladder ...	Ein Mann in einem blauen Hemd steht auf einer ...
4	Two men are at the stove preparing food.	Zwei Männer stehen am Herd und bereiten Essen zu.

```
[45] df_test=pd.read_csv("D:/College/5th Semester/NLP/Assignmnet 1/translation_test.csv")
df_test.head()

✓ 0.0s Python
```

```
[46] df.duplicated().sum()

✓ 0.0s Python
```

```
3
```

```
[47] df.drop_duplicates(inplace=True)

✓ 0.0s Python
```

```
[48] df.dropna(inplace=True)

✓ 0.0s Python
```

```
[49] df_test.duplicated().sum()

✓ 0.0s Python
```

```
0
```

```
[50] df_test.dropna(inplace=True)

✓ 0.0s Python
```

```
[51] BATCH_SIZE=64
english=df['english'].tolist()
german=df['german'].tolist()

✓ 0.0s Python
```

```
MAX_SEQ_LEN=20 # For computation
```

```
[52]
```

```
✓ 0.0s
```

```
Python
```

Data Pre-Processing

```
▷
```

```
from collections import Counter
from itertools import chain
def create_tokenizer(text_corpus, vocab_limit=10000):
    # Create a counter to count the word frequencies in the corpus
    word_counter = Counter(chain.from_iterable(sentence.split() for sentence in text_corpus))

    # Build a vocabulary with a limit
    vocabulary = ["<pad>", "<eos>", "<eos>", "<unk>"] + [word for word, _ in word_counter.most_common(vocab_limit - 4)]

    # Build mappings for word-to-index and index-to-word
    word_to_index = {word: idx for idx, word in enumerate(vocabulary)}
    index_to_word = {idx: word for word, idx in word_to_index.items()}

    # Tokenize function will return a list of integers (the word indices)
    def tokenize(sentence):
        # Use get() to prevent key errors (if a word is not in the vocabulary, it maps to <unk>)
        return [word_to_index.get(word, word_to_index["<unk>"]) for word in sentence.split()]

    return tokenize, word_to_index, index_to_word
```

```
[53]
```

```
import torch
from torch.utils.data import Dataset

class MTDataset(Dataset):
    def __init__(self, input_texts, output_texts, input_tokenizer, output_tokenizer, sequence_limit):
        # Use list comprehension to process the input and output sequences
        self.input_sequences = [input_tokenizer(text)[:sequence_limit] for text in input_texts]
        self.output_sequences = [output_tokenizer(text)[:sequence_limit] for text in output_texts]

    def __len__(self):
        return len(self.input_sequences)

    def __getitem__(self, index):
        input_sequence = torch.tensor(self.input_sequences[index], dtype=torch.long)
        output_sequence = torch.tensor(self.output_sequences[index], dtype=torch.long)
        return input_sequence, output_sequence
```

```
[54]
```

```
✓ 0.0s
```

```
Python
```

```
from torch.nn.utils.rnn import pad_sequence

def batch_collate_fn(batch):
    input_batch, output_batch = zip(*batch)
    # Pad the input and output sequences
    input_padded = pad_sequence(input_batch, batch_first=True, padding_value=0)
    output_padded = pad_sequence(output_batch, batch_first=True, padding_value=0)
    return input_padded, output_padded
```

```
[55]
```

```
✓ 0.0s
```

```
Python
```

```
from sklearn.model_selection import train_test_split
training_input, validation_input, training_output, validation_output = train_test_split(english, german, test_size=0.2, random_state=42)

[56] ✓ 0.0s Python

testing_input = df_test['english'].tolist()
testing_output = df_test['german'].tolist()

[57] ✓ 0.0s Python

eng_tokenizer, eng_word2idx, eng_idx2word = create_tokenizer(training_input)
ger_tokenizer, ger_word2idx, ger_idx2word = create_tokenizer(training_output)

[58] ✓ 0.0s Python

train_dataset = MTDataset(training_input, training_output, eng_tokenizer, ger_tokenizer, MAX_SEQ_LEN)
val_dataset = MTDataset(validation_input, validation_output, eng_tokenizer, ger_tokenizer, MAX_SEQ_LEN)
test_dataset = MTDataset(testing_input, testing_output, eng_tokenizer, ger_tokenizer, MAX_SEQ_LEN)

[59] ✓ 0.1s Python

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=batch_collate_fn)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=batch_collate_fn)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=batch_collate_fn)

[60] ✓ 0.0s Python
```

Now that loading the dataset and pre-processing are out of the way let's look at the pre-trained model. First I prepared the dataset from the data frame by appending them together with separator token in between and then loaded the tokenizer and the model from gpt2 (after downloading it from Hugging Face) I then added the separator token and pad token to the tokenizer and resized the model so it takes the full tokens with the special characters and finally configured the pad token.

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel, AdamW
from torch.utils.data import Dataset, DataLoader
import torch

# Define the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Function to prepare dataset
def prepare_gpt2_dataset(df, sep_token="<|SEP|>"):
    formatted_data = []
    for _, row in df.iterrows():
        src = row["english"]
        trg = row["german"]
        formatted_data.append(f"{src}{sep_token}{trg}")
    return formatted_data

# Load tokenizer and model
tokenizer_gpt = GPT2Tokenizer.from_pretrained("gpt2")
model_gpt = GPT2LMHeadModel.from_pretrained("gpt2")

# Add special tokens
special_tokens = {"sep_token": "<|SEP|>", "pad_token": "<|PAD|>"}
tokenizer_gpt.add_special_tokens(special_tokens)
tokenizer_gpt.pad_token = "<|PAD|>"
model_gpt.resize_token_embeddings(len(tokenizer_gpt))  # Update model embeddings
model_gpt.config.pad_token_id = tokenizer_gpt.pad_token_id

# Prepare dataset
gpt2_texts = prepare_gpt2_dataset(df)

[61] ✓ 3.6s Python
```

Here we are creating the GPT2 dataset by including the tokenizer , the texts and the maximum length. I then initialized the dataset and the train loader and began training

```
# Custom dataset class
class GPT2Dataset(Dataset):
    def __init__(self, texts, tokenizer, max_length):
        self.tokenizer = tokenizer
        self.texts = texts
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        encoding = self.tokenizer(
            self.texts[idx],
            max_length=self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )
        input_ids = encoding["input_ids"].squeeze(0)
        attention_mask = encoding["attention_mask"].squeeze(0)
        return input_ids, attention_mask

# Create dataset and dataloader
max_length = 128
batch_size = 16
dataset = GPT2Dataset(gpt2_texts, tokenizer_gpt, max_length=max_length)
train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

✓ 0.0s Python
```

After moving the model to device and deciding on an Adam optimizer I started fine-tuning by only 3 epochs since it is a sweet medium of fine-tuning and not taking much power , the loop was normal but after each epoch I would print the loss and save the model and the tokenizer.

```
# Move model to device and define optimizer
model_gpt.to(device)
optimizer = AdamW(model_gpt.parameters(), lr=5e-5)
# Training loop
num_epochs = 3
for epoch in range(num_epochs):
    model_gpt.train()
    epoch_loss = 0
    for input_ids, attention_mask in train_loader:
        input_ids = input_ids.to(device)
        attention_mask = attention_mask.to(device)
        # Forward pass
        outputs = model_gpt(input_ids, attention_mask=attention_mask, labels=input_ids)
        loss = outputs.loss
        epoch_loss += loss.item()
        # Backpropagation and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Print epoch loss
    model_gpt.save_pretrained("gpt2_finetuned_checkpoint")
    tokenizer_gpt.save_pretrained("gpt2_finetuned_checkpoint")
    print(f"Epoch {epoch + 1}, Loss: {epoch_loss / len(train_loader):.4f}")

[76] ✓ 279m 45.9s Python
```

```
... d:\College\5th Semester\NLP\Assignmet 1\myenv\Lib\site-packages\transformers\optimization.py:591: FutureWarning: This implementation of AdamW is deprecated
warnings.warn(
Epoch 1, Loss: 1.0662
Epoch 2, Loss: 0.6081
Epoch 3, Loss: 0.5361
```

To then evaluate the model, I used the test data and found that ROUGE score would be the most descriptive as it compares the over-lapping between the generated text and the reference text. I used 3 types of ROUGE (ROUGE1 for n-grams (precision), ROUGE2 for bigrams (recall), ROUGE-L which focuses on the longest common subsequence between the reference and the generated text (f1-score))

```
from rouge_score import rouge_scorer

# Function to generate text using the trained model
def generate_translation(model, tokenizer, src_sentence, max_length=128):
    model.eval() # Set model to evaluation mode
    inputs = tokenizer.encode(src_sentence, return_tensors="pt", truncation=True, max_length=max_length)

    # Create attention mask for padding (1 for non-padding, 0 for padding)
    attention_mask = (inputs != tokenizer.pad_token_id).long()

    # Ensure that pad_token_id is properly set during generation
    generated_ids = model.generate(
        input_ids=inputs.to(device),
        attention_mask=attention_mask.to(device),
        max_length=max_length,
        num_return_sequences=1,
        no_repeat_ngram_size=2,
        do_sample=True,
        pad_token_id=tokenizer.pad_token_id # Explicitly set the pad_token_id
    )

    return tokenizer.decode(generated_ids[0], skip_special_tokens=True)

# Function to evaluate ROUGE scores
def evaluate_with_rouge(model, tokenizer, test_df, max_length=128):
    model.eval()
    scorer = rouge_scorer.RougeScorer(["rouge1", "rouge2", "rougeL"], use_stemmer=True)

    total_rouge1 = 0
    total_rouge2 = 0
    total_rougeL = 0
    num_samples = len(test_df)

    # Generate predictions and compare to target
    with torch.no_grad():
        for i, row in test_df.iterrows():
            src_sentence = row["english"]
            trg_sentence = row["german"]

            # Generate predicted translation
            pred_translation = generate_translation(model, tokenizer, src_sentence, max_length)

            # Compute ROUGE scores between the predicted and target translation
            scores = scorer.score(trg_sentence, pred_translation)

            total_rouge1 += scores["rouge1"].fmeasure
            total_rouge2 += scores["rouge2"].fmeasure
            total_rougeL += scores["rougeL"].fmeasure

    # Calculate average ROUGE scores
    avg_rouge1 = total_rouge1 / num_samples
    avg_rouge2 = total_rouge2 / num_samples
    avg_rougeL = total_rougeL / num_samples

    print(f"Average ROUGE-1: {avg_rouge1:.4f}")
    print(f"Average ROUGE-2: {avg_rouge2:.4f}")
    print(f"Average ROUGE-L: {avg_rougeL:.4f}")

    return avg_rouge1, avg_rouge2, avg_rougeL

# Evaluate using ROUGE on test data
evaluate_with_rouge(model_gpt, tokenizer_gpt, df_test)
```

✓ 36m 36.7s Python

```
... Average ROUGE-1: 0.2352
Average ROUGE-2: 0.1188
Average ROUGE-L: 0.1987
```

So this model has the understanding of basic content and some understanding of longer phrases but the context understanding is very poor and has limited coherent phrasing.

Bonus: try to translate a sentence

So this function is used to translate a sentence.

```
def translate_sentence(model, tokenizer, input_sentence, max_length=128, max_new_tokens=50):
    model.eval()

    # Tokenize input
    inputs = tokenizer(
        input_sentence,
        return_tensors="pt",
        padding="max_length",
        truncation=True,
        max_length=max_length
    ).to(device)

    # Move tensors to the device
    inputs = {key: value for key, value in inputs.items()}
    model.to(device)

    # Generate output
    outputs = model.generate(
        inputs['input_ids'].to(device),
        attention_mask=inputs['attention_mask'],
        max_length=max_length + max_new_tokens, # Extend max_length
        max_new_tokens=max_new_tokens,         # Limit number of new tokens
        pad_token_id=tokenizer.pad_token_id
    )

    # Decode the result
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

```
sentence = "Hello my name is Habiba"

[81] ✓ 0.0s Python

sentence_translated=translate_sentence(model_gpt,tokenizer_gpt,sentence)
sentence_translated

[82] ✓ 0.8s Python

... Both `max_new_tokens` (=50) and `max_length` (=178) seem to have been set. `max_new_tokens` will take precedence. Please refer to the documentation for more
... 'Hello my name is Habiba Ein Hände ist Habiba.'
```

and as you can see it got have the sentence correct (ist Habiba) but the text itself is wrongfully translated and needs more fine-tuning.

Bonus: Seq2Seq Model

This is the OG machine translation sequence to sequence , it has the same pre-processing as the GPT2 transformer so let's dive into the embedding layer created using skip-gram.

Embedding Layer

```
from gensim.models import Word2Vec
import numpy as np

embed_dim = 256
vocab = len(eng_word2idx)

# Training the Word2Vec model
model_2 = Word2Vec(sentences=[sentence.split() for sentence in df['english'].tolist()],
                    vector_size=embed_dim, window=5, min_count=1, sg=1)

# Initialize embedding matrix with random values
embedding_matrix = np.random.uniform(-0.05, 0.05, (vocab, embed_dim))

# Update the embedding matrix with pre-trained Word2Vec vectors
for word, idx in eng_word2idx.items():
    if word in model_2.wv:
        embedding_matrix[idx] = model_2.wv[word]
```

61] ✓ 1.7s Python

```
embedding_matrix.shape
```

62] ✓ 0.0s Python

.. (10000, 256)

Encoder

The encoder is just an LSTM cell which carries the embedding of x , the hidden and cell states.

```
class Encoder(nn.Module):
    def __init__(self, input_dim, embed_dim, hidden_dim, num_layers, embedding_matrix):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(input_dim, embed_dim)
        if embedding_matrix is not None:
            self.embedding.weight.data.copy_(torch.tensor(embedding_matrix, dtype=torch.float32))
            self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers, bidirectional=True, batch_first=True)

    def forward(self, x):
        embedded = self.embedding(x)
        outputs, (hidden, cell) = self.lstm(embedded)
        return hidden, cell
```

[63] ✓ 0.0s Python

Decoder

Here it is an LSTM cell followed by a fully connected linear layer which would compute the predictions.

```
class Decoder(nn.Module):
    def __init__(self, output_dim, embed_dim, hidden_dim, num_layers, embedding_matrix):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(output_dim, embed_dim)
        if embedding_matrix is not None:
            self.embedding.weight.data.copy_(torch.tensor(embedding_matrix, dtype=torch.float32))
            self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x, hidden, cell):
        x = x.unsqueeze(1) # Add time-step dimension
        embedded = self.embedding(x)
        outputs, (hidden, cell) = self.lstm(embedded, (hidden, cell))
        predictions = self.fc(outputs.squeeze(1))
        return predictions, hidden, cell
```

[64] ✓ 0.0s Python

This is an early stopping class for the training loop.

```

class Early_Stopping:
    def __init__(self, patience=3, delta=0):
        self.patience = patience
        self.delta = delta
        self.best_score = None
        self.early_stop = False
        self.counter = 0
        self.best_model_state = None

    def call(self, val_loss, model):
        score = -val_loss
        if self.best_score is None:
            self.best_score = score
            self.best_model_state = model.state_dict()
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.best_model_state = model.state_dict()
            self.counter = 0

    def load_best_model(self, model):
        model.load_state_dict(self.best_model_state)

```

And this is the final Seq2Seq Model

```

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.num_layers = decoder.lstm.num_layers

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        batch_size = src.size(0)
        trg_len = trg.size(1)
        trg_vocab_size = self.decoder.fc.out_features

        outputs = torch.zeros(batch_size, trg_len, trg_vocab_size).to(src.device)

        # Encoder forward pass
        hidden, cell = self.encoder(src)

        # Use the last forward states from the encoder
        hidden = hidden[-2].unsqueeze(0).repeat(self.num_layers, 1, 1) # Repeat for num_layers
        cell = cell[-2].unsqueeze(0).repeat(self.num_layers, 1, 1) # Repeat for num_layers

        # Decoder forward pass
        x = trg[:, 0] # Start token (<sos>)
        for t in range(1, trg_len):
            output, hidden, cell = self.decoder(x, hidden, cell)
            outputs[:, t, :] = output
            teacher_force = torch.rand(1).item() < teacher_forcing_ratio
            x = trg[:, t] if teacher_force else output.argmax(1)

        return outputs

```

Now let's start training

```
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"

[67] ✓ 0.0s Python

device=torch.device("cuda" if torch.cuda.is_available() else "cpu")

[68] ✓ 0.0s Python

input_dim=len(eng_word2idx)
output_dim=len(ger_word2idx)
embed_dim=256
hidden_dim=512
num_layers=2
num_epochs=10

[69] ✓ 0.0s Python

▶ ~
encoder=Encoder(input_dim,embed_dim,hidden_dim,num_layers,embedding_matrix)
decoder=Decoder(output_dim,embed_dim,hidden_dim,num_layers,embedding_matrix)
model=Seq2Seq(encoder,decoder).to(device)
optimizer=optim.Adam(model.parameters(),lr=0.001)
criterion=nn.CrossEntropyLoss(ignore_index=0)

[70] ✓ 0.2s Python + Code + Markdown

for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0

    for src, trg in train_loader:
        src, trg = src.to(device), trg.to(device)
        optimizer.zero_grad()

        output = model(src, trg)
        output_dim = output.shape[-1]

        output = output[:, 1:].reshape(-1, output_dim)
        trg = trg[:, 1:].reshape(-1)

        loss = criterion(output, trg)
        loss.backward()

        optimizer.step()
        epoch_loss += loss.item()
    avg_loss=epoch_loss/len(train_loader)
    print(f"Epoch {epoch+1} epoch loss: {avg_loss:.4f}")

[71] ✓ 10m 43.7s Python

... Epoch 1 epoch loss: 6.0927
Epoch 2 epoch loss: 5.6066
Epoch 3 epoch loss: 5.4055
Epoch 4 epoch loss: 5.1414
Epoch 5 epoch loss: 4.8698
Epoch 6 epoch loss: 4.6085
Epoch 7 epoch loss: 4.3740
Epoch 8 epoch loss: 4.1734
Epoch 9 epoch loss: 3.9913
Epoch 10 epoch loss: 3.8138
```

Of course, the losses are horrendous but this goes to show the importance of attention and the transformer model.

For the evaluation I used the BLEU score which measures the quality of the text generated calculated by this equation

$$\text{BLEU} = BP \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

Where:

- BP : Brevity penalty to discourage overly short outputs.
- p_n : Precision for n -grams.
- w_n : Weight for each n -gram (default is equal weighting for unigrams, bigrams, etc.).
- N : Maximum n -gram order (often $N = 4$ for BLEU-4).

```
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu
model.eval()
predicted_sentences, reference_sentences = [], [] # Lists to store sentences
bleu_results = []

with torch.no_grad():
    for src, trg in test_loader:
        src, trg = src.to(device), trg.to(device)
        model_output = model(src, trg, 0)
        predicted_indices = model_output.argmax(2)
        for i in range(len(predicted_indices)):
            predicted_words = [gr_idx2word.get(idx.item(), '<unk>') for idx in predicted_indices[i]]
            reference_words = [gr_idx2word.get(idx.item(), '<unk>') for idx in trg[i]]
            predicted_words = [word for word in predicted_words if word not in ("<pad>", "<sos>", "<eos>")]
            reference_words = [word for word in reference_words if word not in ("<pad>", "<sos>", "<eos>")]
            predicted_sentences.append(predicted_words)
            reference_sentences.append(reference_words)
            bleu_results.append(sentence_bleu([reference_words], predicted_words))

avg_bleu_score = sum(bleu_results) / len(bleu_results)
print(f"The average sentence-level BLEU score is {avg_bleu_score:.4f}")
corpus_bleu_score = corpus_bleu(reference_sentences, predicted_sentences)
print(f"The corpus-level BLEU score is {corpus_bleu_score:.4f}")
```

```
Warning: warn(_msg)
The average sentence-level BLEU score is 0.0150
The corpus-level BLEU score is 0.0395
```

The scores are bad, and this goes to show that the Seq2seq model is way too simple for this kind of task.

Final Comparison and Conclusion

The worst performance must be the Seq2Seq Model which showed a very low BLEU score as it is way too simple for this complicated task followed by the pre-trained GPT transformer and I think this is for 2 main reasons with the first one being that there wasn't that much data to begin with and two that GPT-2 transformer is not trained specifically for machine translation task so while it gives ok scores maybe using another transformer would have produced better results. The best performance must be the Transformer from scratch, but it is really complex to build from scratch and is only useful so you can grasp the transformer's idea and how they work but nevertheless the scores were great.

In the end, for the machine translation task you can either choose a pre-trained transformer specifically trained for machine translation tasks or if you have the time build your own transformer from scratch.