Task 4

**Getting the dataset ready**

* Import wmt14 dataset for german to English translation, split into train, test and validation sets
* Take the first 50k pairs of sentences from the train datset and store English and german in 2 separate lists. Do the same for all the sentences in validation dataset

**Training tokenizer**

* We create a set words, such that duplicates are automatically deleted. For every word in the sentence, we separate it from other words by using spaces and punctuation marks and then add it to words. The length of the set words is returned in the end
* We define a function to implement tokenization.
  + - byte pair encoding model with all unknown tokens (out of vocabulary words) set to “unk”.
    - normalization used is normalization form decomposed (characters that are composed of multiple glyphs into their base character and the diacritical mark), converts to lowercase and strips all accents.
    - Pretokenizer – whitespace
    - Special tokens:

<unk> for out of vocabulary tokens

<pad> to maintain constant length of sentences

<cls> is used as the overall sentence level representation,

<sep> to represent punctuation

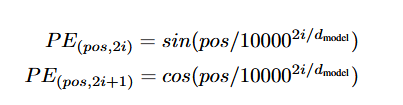
<mask> used to hide some tokens and used in masked language modeling to predict missing words

* Train the tokenizer on these sentences using BPE. The tokenizer learn up to 100,000 of the most frequent subword pieces. These learned pieces will then be used to tokenize any future text. Special tokens will be dealt with as mentioned above.

**Model**

**Positional encoding**

We use positional encoding so that the embedding vector captures both the semantic meaning of the token and its position in the sentence — positional information is important because we're training on sequences, where word order matters. The model does not make use of recurrence so positional encoding is how we make use of the fact that our data is part of a sequence.



This is how positional encoding is implemented.

**Transformer with tokenizer integration**

* We load a tokenizer to go from text to token and vice versa.
* Then, we create the embedding matrices, one for the source language and one for the target language. They convert the input tokens to dense vectors of size d\_model.
* Then, we add the positional encoding to these embedding vectors.

Then, we define **the core transformation architecture.**

* We specify the embedding size (d\_model), which determines the dimensionality of all the internal representations.
* nhead parameter sets the number of parallel self-attention heads in each multi-head attention layer. Each head can focus on different parts of the sentence, helping the model capture more nuanced relationships between words.
* We choose how many layers to stack in both the encoder and decoder (num\_encoder\_layers and num\_decoder\_layers). Stacking multiple layers allows the model to build up deeper understanding.
* The dim\_feedforward value controls the size of the hidden layer inside the feedforward network in each encoder/decoder block. It temporarily expands the dimensionality (e.g., from 512 to 2048) to help the model learn more complex patterns before bringing it back down.
* dropout=0.1 introduces regularization by randomly turning off 10% of the neurons during training. This prevents the model from relying too heavily on any one feature and helps reduce overfitting.

Then there is the **fc.out** layer, which is the final linear layer that helps you predict the next token.

Next, we write the **forward pass** of the Transformer model.

* This function embeds the source and target sequences and applies positional encoding to it.
* Then we pass these vectors to the transformer, using the given masks. These masks allow the model to focus on specific parts of the input or to ignore padding tokens.
* Finally, the output is passed through the fc.out layer to generate the final output (predict next word)

Next, the **encode\_input** function encodes the input text using the source tokenizer. The **decode\_output** function, on the other hand, converts a sequence of token IDs to human readable strings.

Then, we define the **generate function.**

* We set the model to evaluation mode
* Encode the input text while also moving the input to the correct device.
* It initialises the target tokens. The target sequence starts with the special token <cls>
* Then we run a loop and the function starts generating tokens one at a time. In the loop,
  + - It makes use of the masking technique to prevent the mode from peeking ahead.
    - Then, we pass the source and current target to the model, which processes the input and predicts the next token in sequence.
    - We then choose the token with maximum probability of coming next (doen using argmax(dim=-1)
    - If an end of sequence token is generated, we break out of the loop early. Otherwise, we add this token to the target tokens list.
    - The loop continues till eos token is generated or till we reach the max length of the target sequence.
* Then, the final output is decoded and returned.

**Pytorch Dataset**

The class here mainly works with dataset creation and tokenization.

* The init function initializes the src and tgt sentences, as well as the tokenizer, max-length and special tokens
* The len function returns the number of samples (pairs of source and target sentences) in the dataset, which is the same as the length of src\_sentences.
* The getitem function fetches a single sample from the dataset, and performs tokenization and padding on this sample.

The training and validation datasets are created using this class.

**Pytorch Dataloader**

Batch size of 32 is used fr both training and validation sets. Training set is shuffled, whereas validation set isn’t.

what can i reduce so that the efficiency remains more or less same but parameteres come down

**Training**

* The creat\_ padding\_mask function is used to identify padding tokens in a sequence and return a mask that can be used to ignore padded positions during model computation.
* The model is trained using Adam optimiser, and the loss function is cross entropy loss. The loss function ignores the <pad> token. It is trained for 10 epochs.
* For every epoch, we perform teacher forcing. This makes sure the model doesn’t see the word before predicting it.
* The tgt\_mask prevents the model from seeing future tokens in the decoder. We also mask the padded tokens, so they can be ignored during training.
* We perform the forward pass by passing the required parameters.
* Then, the output is reshaped to fit the input of the loss function
* Loss is calculated between predicted and actual tokens
* We perform backpropagation to update model weights, and then display epoch summary.

**Testing**

We test our model on a single example sentence.

Note on hyperparameter tuning: tried but seems that the current values work best. In any case, we concluded that our dataset was too small for the model. So there can’t be a lot of improvement anyway.