I am forced to submit this report in English due to my limited skills in Chinese. I apologize about this. I hope that it will not be a problem and I will be grateful if I can get feedback about the assignment in English, too.

## Model

The model used for text summarization was the small multilingual T5 model (`mt5-small`). On a high level of abstraction, when applied to text summarization, it can be decomposed into three units: the encoder, decoder, and generator. The encoder is fed the text which is to be summarized. For each timestep, the decoder one is fed the output of the encoder as well as an additional input token. The first of those additional input tokens is the artificial beggining-of-sequence token. Based on these inputs, the decoder generates a probability distribution of tokens from the vocabulary, which is meant to estimate how likely a given token is to be the next token in the generated sequence. Based, in turn, on this distribution, the generator chooses the next token of the output sequence, which in turn is fed to the decoder as the next additional input token. This behavior (feeding the outputs back to the decoder) is called autoregression. The cycle between the decoder and generator repeats until an end-of-sequence token is generated.

Let us now consider each part in a bit more detail. The encoder is composed of 8 identical layers (as opposed to the normal, not `small`, `mt5` model, having 12 layers). The input of the first layer is the input sequence (after tokenization and conversion to word embeddings). The input of each subsequent layer is the output of the previous layer. Each layer can be further decomposed into two sub-layers, with the output of the first one being the input of the second one. Each sub-layer implements a function

$$f_i(x) = LN(x + g_i(x))$$

where LN is the layer normalization function as described here, and $g_i$ is a function specific to the $i$-th layer. For the first layer, $g_0$ is the multi-head attention function. For the second layer, $g_1$ is a function implemented by a feed-forward network, performing two linear transformations with a GeGLU acivation in between:

$$FF(x) = W_2 \cdot ReLU(W_1 x + b_1) + b_2$$

I hope it is fine to omit specific equations for each of those functions, since they would be almost identical to the ones we already provided in the report for this semester's second task.

The decoder works similarly, also being composed of 8 layers, but each of those layers includes additional sublayer between the attention sublayer and the feed-forward sublayer. This sublayer, to quote the original BERT paper, 'performs multi-head attention over the output of the encoder stack'. In addition, the attention mechanism is modified so that positions cannot attend to later positions in the sequence.

The decoder is followed by a linear transformation, converting its hidden state to logits, which are then converted to a probability distribution over tokens from the vocabulary by applying the softmax function.

Finally, the internal mechanism of the generator depends on the generation algorithm. There are numerous such algorithms, and the ones I tried for this task are described below in the 'Generation strategies' section.

## Preprocessing

An input sequence, before being fed to the model, undergoes three preprocessing steps: tokenization, padding, and conversion to token embeddings.

A pretrained fast `T5Tokenizer` is used for the first of those tasks. It implements the Sentencepiece algorithm, with fixed vocabulary trained with the Unigram algorithm.

Let us first consider the vocabulary creation. The Unigram algorithm, given a corpus, initializes the vocabulary to a vary large set of tokens, and removes tokens iteratively until a desired vocabulary size is reached. Each token is assigned a probability equal to its frequency in the corpus divided by the sum of frequencies of all the tokens in the vocabulary.

To decide which tokens to remove at each iteration, at each timestep the loss function, equal to the sum of negative log-likelihoods of all the tokens, is computed. For each token, it is then calculated how much the los swould increase if it were removed from the vocabulary. Finally then the token whose removal would increase the loss the least is removed.

It should be noted that although this is the general idea of Unigram, in practice certain heuristics, like removing a number of tokens at once, are applied to speed the process up.

Learning a vocabulary by means of the Unigram algorithm is not sufficient to tokenize a sequence, since the sequence may have multiple possible tokenizations into tokens from that vocabulary. Sentencepiece decides on the best tokenization for each individual sequence. If I understand correctly, this means selecting the tokenization with the best product of probabilities of all its tokens.

After tokenization, each sequence in a given batch is dynamically padded to the length of the longest sequence in that batch.

Finally, as is usually the case for NLP models based on Transformers, tokens are replaced with their respective pretrained word embeddings.

# Generation strategies

## Algorithms

The following generation strategies have been tried out for the task:

- Greedy search. This simple algorithm consists in choosing, at every timestep, the token with the highest probability in the probability distribution output by the decoder part. It does not have any adjustable parameters.

- Beam search. This algorithm has an adjustable parameter $k$. At each timestep $n$, it keeps track of the $k$ sequences of length $n$ judged to be the best, i.e. those assigned the highest scores. A sequence's score is equal to the sum of log probabilities of all the tokens it contains (probabilities from the moment of adding them to the sequence). At timestep $n$, for each of those $k$ sequences, we want to find the $k$ tokens most likely to come next in the given sequence. Therefore, for each sequence we compute the probability distribution over all the tokens, and consider the $k$ most probable tokens. This means that at this moment, we have $k^2$ candidate sequences. Each of those is assigned a new score equal to the log probability of its newly-added token, added to the score previously assigned to its parent sequence (without the last token), and then the $k$ sequences with the best scores are kept. If the special END token is predicted for a sequence, the sequence is set aside and we no longer attempt to add new tokens to it. After a certain criterion is met, usually either that we have set aside a given number of sequences, or we have executed a given number of timesteps, the algorithm stops and we choose the best one from the sequences which have been set aside. We normalize (divide) each of their scores by length and eventually return the sequence with the highest score.

- Top-k sampling. In this algorithm, each consecutive token is sampled randomly. We take the probability distribution over all the vocabulary tokens as output by the decoder. We only consider the $k$ tokens with the highest probabilities, with $k$ being a hyperparameter. We normalize their probabilities so that they sum up to 1. Then we sample from this distribution. This continues until an END token is sampled.

- Top-p sampling (or nucleus sampling). It is analogous to top-k sampling, with $p$ being a hyperparameter. Instead of considering the $k$ most probable tokens, we consider the $n$ most probable tokens, with $n$ being the smallest possible integer such that the probabilities of the $n$ most probable tokens sum up to at least $p$.

These strategies can be further modified by a temperature setting. Temperature is a number by which all the logits output by the decoder are divided before being fed to the softmax function. This

increases the probability of the more probable tokens, while decreasing the probability of the less probable tokens. It can be used to modify the behavior of, among others, the decoding strategies described above. It is worth noting, though, that it has no effect on the greedy decoding algorithm - it does not change which token is the most probable, only its probability.

### Testing in this task

I tried out the following decoding strategies (ROUGE-L scores, obtained by the `evaluation/strategies/eval_generation.ipynb` script, are provided for each of them):

- Beam search with beam size 3, score 17.261,
- Beam search with beam size 4, score 17.45
- Beam search with beam size 5, score 17.1292
- Greedy search, score 15.5877
- Top-k with $k = 120$, score 15.6331
- Top-k with $k = 32$, score 15.6243
- Top-p with $p = 0.6$, score 15.6082
- Top-p with $p = 0.85$, score 15.6248

A few conclusions can be drawn from this:

- All the scores are below the baseline. However, when I manually tested the model by running the prediction and evaluating it with the program provided for this purpose in the `ADL22-HW3` repository, the scores were considerably better and above the baseline (they are further discussed in the Evaluation section below). I am not entirely sure what caused this discrepancy. I decided to keep this methodology because I reckoned that even if the scores are different, their order from best to worst still reflects the performance of each strategy.
- There is a wide gap between the strategies based on beam search and all the others.
- The scores of the other strategies are hardly different from each other, although it can be seen that greedy search has the worst performance, and for both top-k and top-p, the higher tried-out values of the respective hyperparameters gave marginally better results.
- The best beam size was 4. This was quite lucky - out of the first three beam sizes that I tried out, the middle one proved to be the best. I assumed that values below 3 and above 5 would only get progressively worse, and did not test any other values.

Since the results obtained by beam search were superior, I decided to only try changing the temperature hyperparameter with this strategy. I also kept the best beam size 4, because I guessed such setting would be the most likely to give a good final score. I tried out two temperature values:

- 0.8, giving a score of 17.4918,
- 0.9, giving a score of 17.4548

This shows that temperature, too, has a small impact on the final score. For the final strategy, I decided to keep beam search with a beam size of 4 and no temperature setting. (defaulting to 1).

The precise scores for all the strategies mentioned here can be seen in the `evaluation/train_rouge` directory.

A small final remark: the `generate` function used for generating sequences based on these strategies has a default setting to only consider the 50 most probable tokens at each step. I kept this setting, except for the top-k searches. This means it might have slightly changed the scores obtained for the top-p strategy.

## Training hyperparameters

I chose the batch size of 2 to save GPU memory, and applied 16-bit floats for the same reason. I did not adjust any other parameters, so, among others, the following defaults applied:

- Learning rate: 5e-05
- Optimizer - Adam
- Attention heads - 6

## Evaluation

The final F1 for the ROUGE-L metric for the model is equal to 0.22. More precise results for the final model and a few training checkpoints can be seen in the `evaluation/train_rouge` directory. Based on these results, the learning curve below has been drawn.