ADL HW3

資工AI所碩一 r11922a05 林聖硯

Q1: Model

- Model: Describe the model architecture and how it works on text summarization.

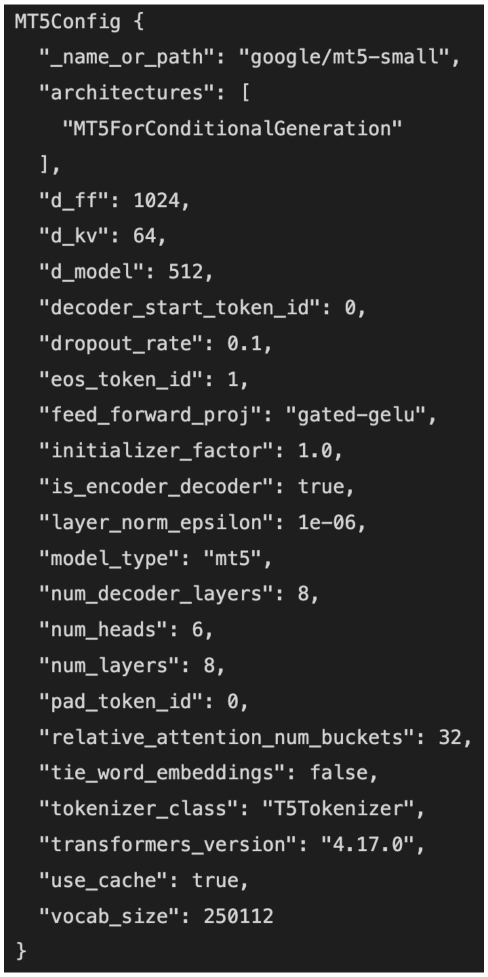
Model architecture:

The model used in this task is **mT5: A Mussively Multilingual Pre-trained Text-to-Text Transformer** proposed by google brain in 2021. It is an **encoder-decoder** (i.e. a sequence-to-sequence) model comparing to the model (BERT, which is only an encoder) that was used in the last homework. The tokenizer used in mT5 is called **SentencePiece**, which is lighter than the WordPiece in BERT. However, it is not quite convenient for Chinese input since the tokenizer will change full-width punctutations into half-width punctutations. The specific model used in this task is mt5-**small**, which is the lightest model in the mt5 family. It has only 8 layers in both encoder and decoder, and it has 6 heads in its multi-head attention module.

Text summarization:

In the text summarization task, the input sentence will be tokenized into several tokens, and then these tokens will be turned into `input_ids` by the mt5 tokenizer. During the encoding phase, the tokenized sentence will be fed into the encoder of mt5 and then the encoder will output its`memory`, which can be viewed as a compressed information of the original input sequence modeled by encoder. During the decoding **training** phase, we'll input both the `memory` from encoder and the summarized sentence into decoder, and used the right-shited summarized sentence as ground truth to compute loss. In this way, the entire model is trained via **Teacher Forcing.** During the inference phase, the encoding process is the same as training phase; however, we'll use the `start` token as the starting input of decoder, and then autoregressively complete the whole sentence, i.e. use the output of previous timestep as the input of next timestep.

The config that I used for mt5-small



```
MT5Config {
  "_name_or_path": "google/mt5-small",
  "architectures": [
    "MT5ForConditionalGeneration"
  ],
  "d_ff": 1024,
  "d_kv": 64,
  "d_model": 512,
  "decoder_start_token_id": 0,
  "dropout_rate": 0.1,
  "eos_token_id": 1,
  "feed_forward_proj": "gated-gelu",
  "initializer_factor": 1.0,
  "is_encoder_decoder": true,
  "layer_norm_epsilon": 1e-06,
  "model_type": "mt5",
  "num_decoder_layers": 8,
  "num_heads": 6,
  "num_layers": 8,
  "pad_token_id": 0,
  "relative_attention_num_buckets": 32,
  "tie_word_embeddings": false,
  "tokenizer_class": "T5Tokenizer",
  "transformers_version": "4.17.0",
  "use_cache": true,
  "vocab_size": 250112
}
```

- Preprocessing: describe your preprocessing (e.g. tokenization, data cleaning and etc.)

ref: SentencePiece github

I didn’t do any data cleaning, and just used the tokenizer (i.e. SentencePiece) provided by the mt5 model. SentencePiece is an unsupervised text tokenizer mainly for Neural Network-based text generation systems where the vocabulary size is predetermined prior to the neural model training. It implements subword units (i.e. Byte Pair Encoding, BPE) and unigram language model with the extension of direct training from raw sentences. Precisely, it uses the Viterbi Algorithm (a kind of dynammic programming algorithm) to search for the best token pairs.

Q2: Training

- Hyperparameter: Describe your hyperparameter you use and how you decide it.

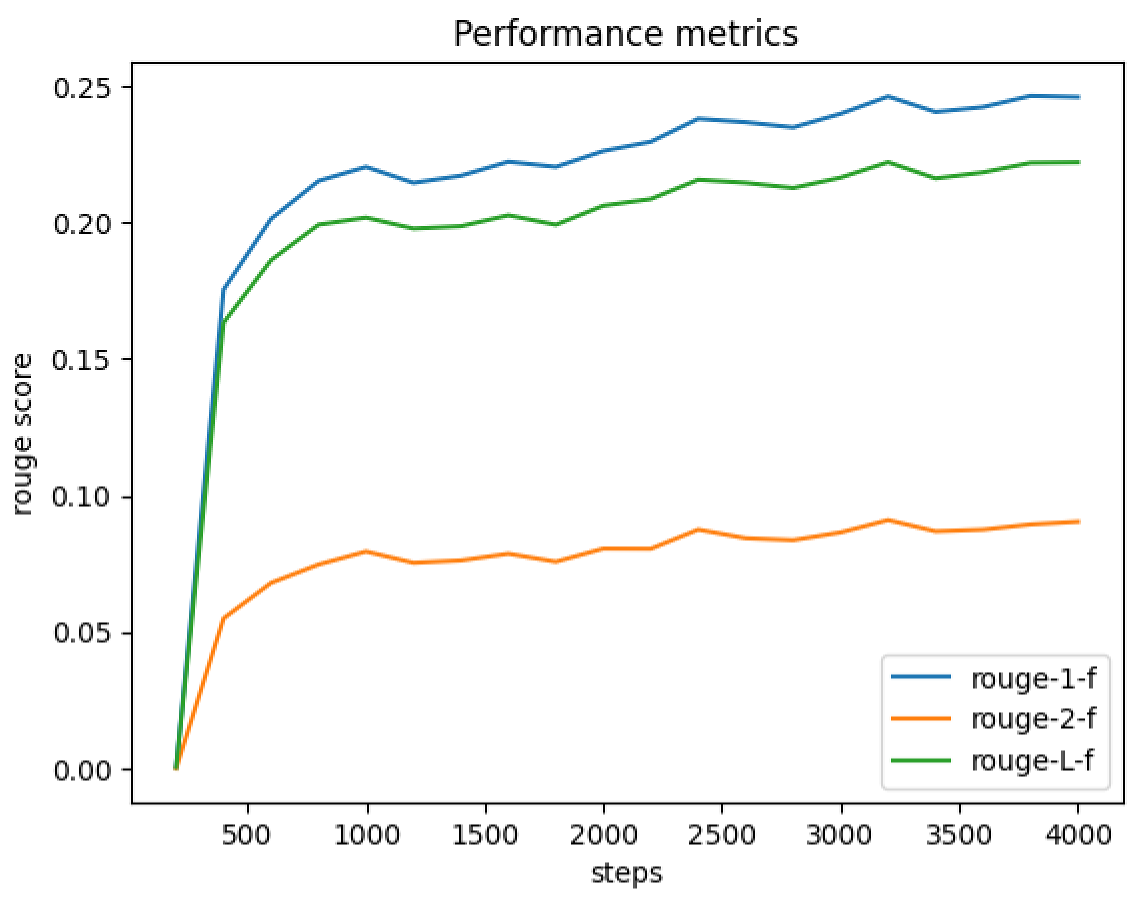
The hyperparamters that I used are the followings:

- learning rate: 1e-3
- max_source_length: 1024
- max_target_length: 128
- Effective batch size: 16 (2*8 = per_device_batch_size * per_device_accumulation_step)
- Epoch: 3
- Optimizer: Adafactor
- warmup_ratio: 0.1

The default learning rate was 5e-4, but after several experiments, I found it leaded to bad performance result and the learning efficiency was low during training phases. Thus, I tried several learning rate (1e-2, 5e-3, 1e-3, 5e-4, 1e-4) and found that the best performance result happened when the learning rate is 1e-3.

In addition, after changing the optimizer to adafactor and setting the warmup ratio at 0.1, the validation loss and performance metrics improved but the other hypermeters only make the performace worse. Therefore, I spend more time on decoding strategy experiments.

- Learning Curves: Plot the learning curves (ROUGE versus training steps)



Q3: Generation Strategies

- Stratgies: describe the detail of the following generation strategies

Reference: How to generate text: using different decoding methods for language generation with Transformers

1. **Greedy**

   The idea of greedy search is to select to the word with the highest probability as its next word at each timestep. However, it has many drawbacks. During the auto-regressive process, the words repeat itself easily and sometimes this strategy may lead to the result of missing high probability words hidden behind a low probability word.

2. **Beam Search**

   The idea of bean search is to reduce the risk of missing hidden high probability word sequences by keeping the most likely `num_beams` of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability.

It has many advantages comparing to greedy search. First, it will always find an output sequence with higher probability than greedy search. Second, it can work very well in tasks where the length of the desired generation is more or less predictable as in machine translation or summarization. Nevertheless, it usually have bad result for open-ended generation where the desired output length can vary greatly and it is not guaranteed to find the most likely output. In addition, researchers found that high quality human language does not follow a distribution of high probability next words.

3. **Top-k Sampling**

   The idea of Top-k Sampling is to select K most likely next words and the probability mass is redistributed among only those K next words. By using top-k sampling, we could add some randomness in the auto-regressive process, which could let the whole sentences more human-like.

   However it still has some disadvantanges. Limiting the sample pool to a fixed size K could endanger the model to produce gibberish for sharp distributions and limit the model's creativity for flat distribution. In addition, it does not dynamically adapt the number of words that are filtered from the next word probability distribution.

4. **Top-p Sampling (nuclues sampling)**

   The invention of top-p sampling was to overcome the problems of top-k sampling. The idea of top-p sampling is to choose from the smallest possible set of words whose cumulative probability exceeds the probability `p`. The probability mass is then redistributed among this set of words. Thus, we could avoid sampling words with low probability

5. **Temperature** (ref: stackoverflow)

   Temperature is a hyperparameter in softmax function that controls the “sharpness” of the next word distribution in an auto-regressive process. If we **increase** the temperature in softmax function, the likelihood of high probability words will **decrease** and the likelihood of low probability word will **increase**. Thus, the next word probability distribution is softer.

   With regards to why higher temperatures lead to softer distributions, that has to do with the exponential function. The temperature parameter penalizes bigger logits more than the smaller logits. The exponential function is an 'increasing function'. So if a term is already big, penalizing it by a small amount would make it much smaller (%-wise) than if that term was small.

- Hyperparameters
  - Try at least 2 settings of each strategies and compare the result.

I do the following experiment to decide which generation strategy to use for my final submission. The experiment results are listed in the following table. In addition, I try different hyperparameters according to different generation strategies used (E.g. change number of beams in beam search, try different value of K in Top-k sampling, etc.

Rouge-f score / decode strategy	Rouge-1-f	Rouge-2-f	Rouge-l-f
Greedy	0.2085	0.0683	0.1903
Beam search + num_beams = 2	0.2153	0.0922	0.2301
Beam search + num_beams = 4	0.2617	0.1040	0.2364
Beam search + num_beams = 8 (Long inference time)	0.2639	0.1078	0.2401
Top-k sampling + k = 10	0.1893	0.0661	0.1872
Top-k sampling + k = 20	0.1902	0.0728	0.1957
Top-k sampling + k = 30	0.1728	0.0672	0.1792
Top-k sampling + k = 20 + tmp = 1.2	0.2051	0.0852	0.1903
Top-k sampling + k = 20 + tmp = 0.8	0.2015	0.0831	0.1834
Top-p sampling + p = 0.8	0.2187	0.0781	0.1900
Top-p sampling + p = 0.85	0.2108	0.0853	0.2018
Top-p sampling + p = 0.9	0.2063	0.0701	0.1829

Observation

Beam search is the best strategy among all generation strategies and the next best is top-p sampling. As k increases, top-k sampling reaches its best performance when k equals to 20. Thus, I changed the temperature in this situation and found that when temeprature is high (next word distribution is softer), the result is better. In top-p sampling, the best performance result happens when p equals to 0.8

- What is your final generation strategy? (you can combine any of them)

According to my experiment result, beam search with 8 beams gets the best performance result in public dataset. However, the inference time may exceed the time limit. Therefore, I chose beam search with 4 beams as my final generation strategy.