ADL HW3

Model

model

- **model architecture** :
    - **name** : multilingual variant of Text-to-Text Transfer Transformer
    - **Frame** : encoder-decoder
    - **pretrain** :
        - **BERT-style** : mask part of text and try to generate original text.
        - **Replace spans** : replace a sequence of text by a single mask token. In the original T5 model, the masked sequence length is 3.
        - **Mask ratio** : 15% of text
- **model config** :

```
{
  "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",
  "vocab_size": 250112
}
```

- **For summarization** :
  MT5 can be use for many different task by adding different prefix on input data. I use "summarize: " as the training frefix.
- **resource** : **https://huggingface.co/google/mt5-small** (https://huggingface.co/google/mt5-small)

Preprocessing :

- **tokenizer** : T5Tokenizer
  - **resource** : **https://github.com/huggingface/transformers/blob/main/src/transformers/models/t5/tokenization_t5.py** (https://github.com/huggingface/transformers/blob/main/src/transformers/models/t5/tokenization_t5.py)
- **padding and truncate** :
  - **text** : max_len = 1024
  - **title** : max_len = 128

- **valid input** :
  - **utf-8 to chinese** :
    Take public.jsonl to example:

```
import json
with open('./data/public.jsonl', 'r', encodi
    data = [json.loads(line) for line in f]
with open( './data/public.json', 'w') as f:
    for i in data:
        count+=1
        json.dump( i, f, ensure_ascii=False
        f.write('\n')
```

- **training and predicting args** :
  Let main program know which column to use.
  - **text_column** : maintextt
  - **summary** : title

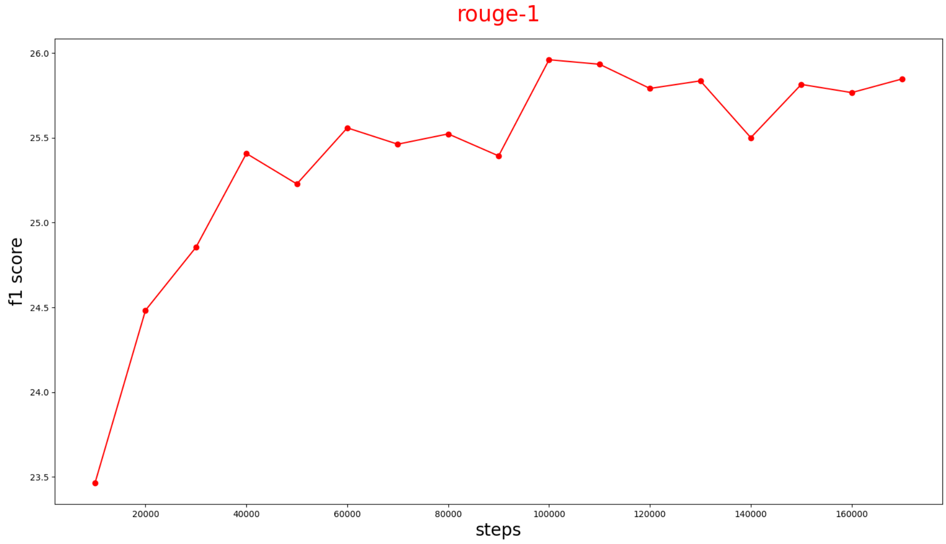
Training

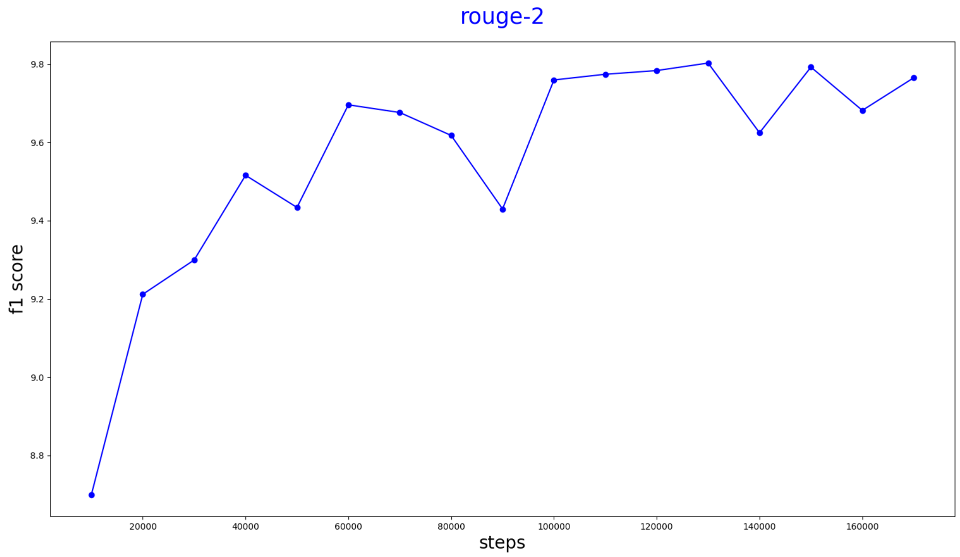
Hyperparameter

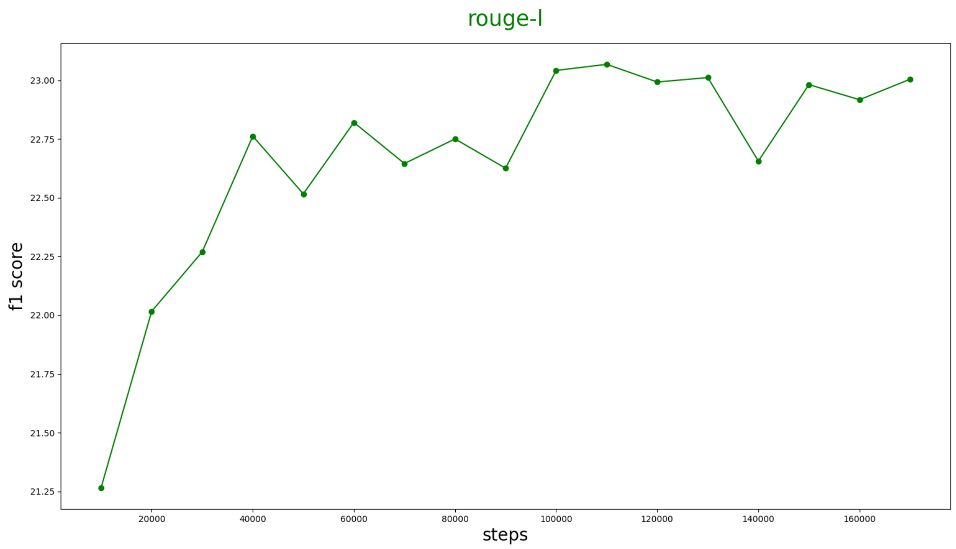
- **num_train_epochs** : 32
- **per_device_train_batch_size** : 2
- **gradient_accumulation_steps** : 2
- **learning_rate** : 5e-5
- **optimizer** : AdamW

Learning Curves

There are 173664 steps for 32 epochs, ploting every 10000 steps.







Generation Strategies

Stratgies

- **Greedy** :
  Always choose the most possible word to generate.

- **Beam Search** :
  Decided number of beams, n_b.Then always keep the most likely n_b of sequence to generate next word.

- **Top-k Sampling** :
  Decided number of k, n_k. Then randomly choose next word from the most likely n_k word.

- **Top-p Sampling** :
  Decided number of p, n_p. Then choose next word form the set
  $\{w_0 ... w_i \mid \sum_{k=0}^{i} P(w_k) \leq n_p < \sum_{k=0}^{i+1} P(w_k),\ P(w_k) > P(w_{k+1})\}$
  .
  Different from Top-K Sampling, word w_i with higher $P(w_i)$ has more posibility being choosed.

- **Temperature** :
  The original softmax :

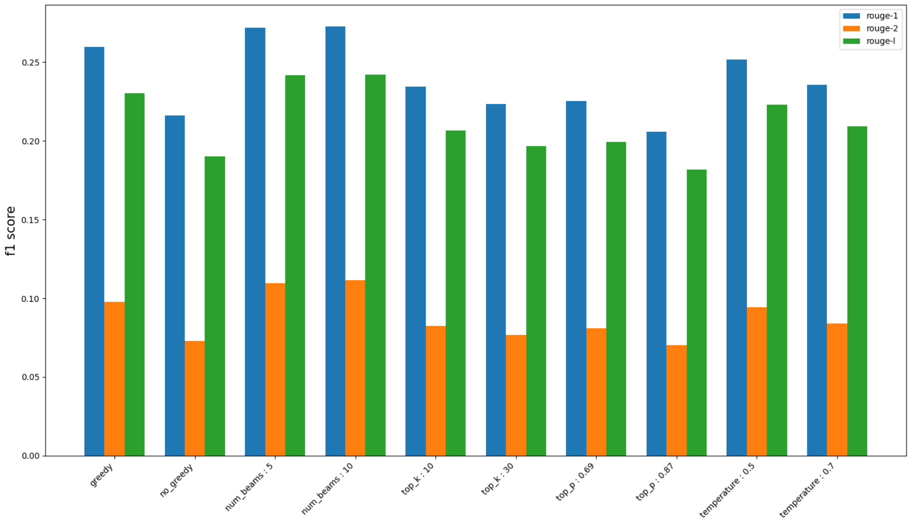
$$q_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

  By adding the temperature T into the equation:

$$q_i = \frac{exp(\frac{z_i}{T})}{\sum_j exp(\frac{z_j}{T})}$$

  We can decided the shape of the distribution: the bigger temperature, the sharper distribution.
  Hyperparameters

I choose the model saved from the 100000 steps while training to compare the different generating strategy.



- **Greedy** : **greedy** vs **no-greedy**
  No-greedy here is actually top-k sampling where $n_k = 50$, becuase transformer.generate() take top_k = 50 as default setting.
  Greedy strategy is much more better than no-greedy, I think it's reasonable because randomly choosing may cause bad outcome.

- **Beam Search** : **num_beams = 5** vs **num_beams = 10**
  Beam Search is the best of all strategies, and the higher num_beams comes higher score becuase it consider more posibility.

- **Top-k Sampling** : **top_k = 10** vs **top_k 30**
  With lower top_k, I got higher score. I think it's becuuse that the more words to be consider, the more posibility to choose bad word.

- **Top-p Sampling** : **top_p = 0.69** vs **top_p = 0.87**
  The same reason and outcome as Top-k sampling. Considering more posibility doesn't guarantee the better outcome.

- **Temperature** : **0.5** vs **0.7**
  For this case, I choose $n_p = 0.69$ to compare the difference from temperature. The lower temperature

has better outcome. I think it's because that the sharper distribution let the word with higher posibility can be choose more often.

- **My final generation strategy**
  I choose **num_beams = 10** as my final strategy. Though it is simple, but it has the best preformance in my test.

Implement
I basicly follow hugginface example summarization to complete this home work.
Sample code :
https://github.com/huggingface/transformers/tree/main/examples/pytorch/summarization
(https://github.com/huggingface/transformers/tree/main/examples/pytorch/summarization)
Because the trainer used by huggingface only support mum_beams and max_len.
Original trainer:
https://github.com/huggingface/transformers/blob/main/examples/legacy/seq2seq/seq2seq_trainer.py
(https://github.com/huggingface/transformers/blob/main/examples/legacy/seq2seq/seq2seq_trainer.py)
So, I modify the sample code and trainer in the my local python lib to support top_k, top_p, early_stopping, do_sample and temperature. So that I can test the different generate strategy.