Q1: Model

model architecture

Listing 1: model config


```
{
  "_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": "4.6.0.dev0",
  "use_cache": true,
  "vocab_size": 250100
}
```


為 transformer encoder decoder 架構。

forward 會將 source text tokenizer output(src_{input_ids}) 當作 encoder input，輸出為 encoder 每一層的 tranformer block output(1)，將最後一層 encoder output($encoder_{output}[0]$) 當作 decoder 的 encoder hidden states，並將 target text tokenizer output(tgt_{input_ids}) 往右 shift(2)，當作 decoder input，decoder 會透過 input 之間 attention，以及與 encoder hidden states 的 attention(3)，decoder output 的結果在經過 linear layer 出來為每個 subword 的機率。

generate 會則是將前面 decode 的結果當作 decoder 的 input，依序產生結果。

$$encoder_{output} = encoder(src_{input\_ids}) \tag{1}$$

$$decoder\_input[:, 1:] = tgt_{input\_ids}[:, :-1] \tag{2}$$

$$decoder_{output} = decoder(decoder\_input, encoder\_hidden\_states = encoder_{output}[0]) \tag{3}$$

$$decoder_{output} = decoder(past\_decoder\_output, encoder\_hidden\_states = encoder_{output}[0]) \tag{4}$$

Preprocessing

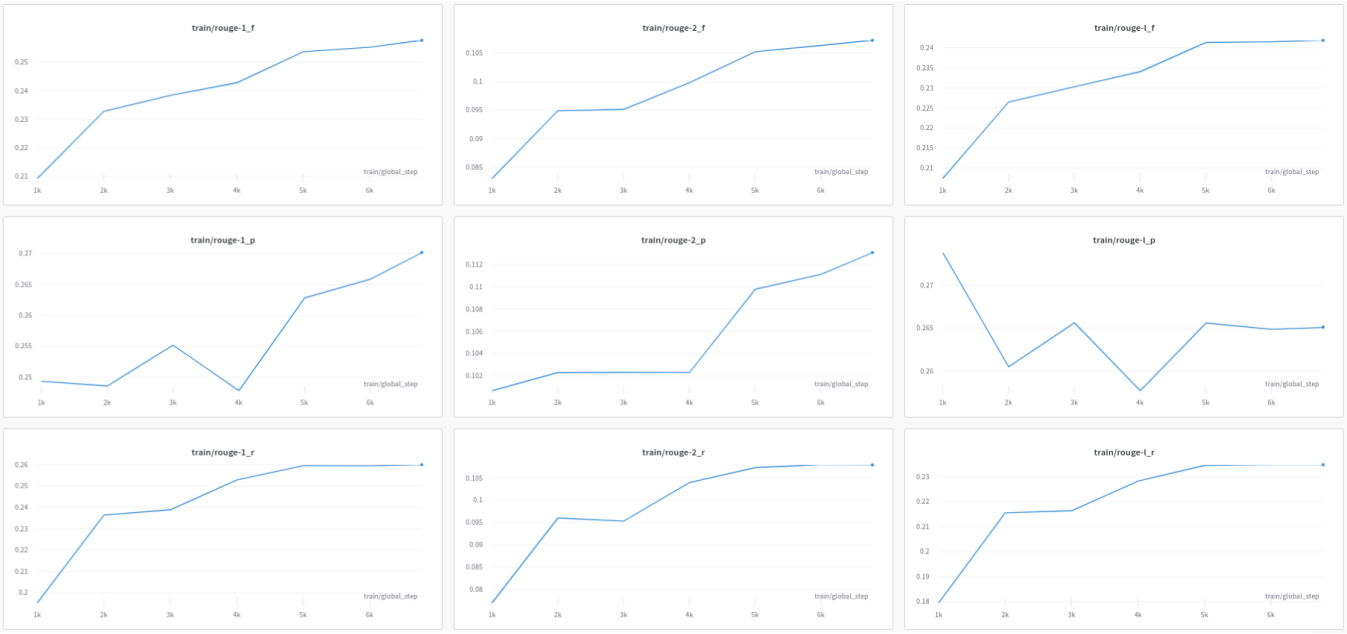
tokenization: t5 tokenizer 為 sentencepiece，是由 byte pair encoding 以及 unigram language model 組成，會將 sentence 切成 subword，並利用 viterbi algorithm 找出機率最大的 path 來做 tokenize。

Q2: Training

Hyperparameter

batch size 為 4，gradient accumulation steps 為 4，batch size 太小會造成結果不好，且可能造成 gradient 計算的誤差，因此設定 gradient accumulation steps；參考助教的建議 optimizer 使用 Adafactor(lr=1e-3)，與 adam 相比，少了 momentum，因此記憶體使用量較少，另外也有使用 fp16 訓練減少記憶體使用量。

Learning Curves



Q3: Generation Strategies

Stratgies

* greedy: 每個 time step 都選 $P(w|w_1...w_{t-1})$ 最大的那一個。

* Beam Search: 每個 time step 都選 num beams 個候選，下一個 time step 則從這一個 time step 的候選中找出前 num beams 機率最大的候選。

* Top-k Sampling: 從 $P(w|w_1...w_{t-1})$ 的機率分佈從機率最大的前 K 名 sample 出 w_t。

* Top-p Sampling: 從 $P(w|w_1...w_{t-1})$ 的機率分佈從機率最大的前幾名累加到 p 的 word sample 出 w_t。

* Temperature: 對 softmax 做 sharpen 或 smoothing，小於 1 為 sharpen，大於則為 smoothing。

Hyperparameters

greedy vs no greedy(sample)

	greedy	sample
rouge-1_f	0.2417	0.1979
rouge-1_p	0.2615	0.2067
rouge-1_r	0.2379	0.2006
rouge-2_f	0.0928	0.0661
rouge-2_p	0.1004	0.0691
rouge-2_r	0.0917	0.0673
rouge-3_f	0.2281	0.1814
rouge-3_p	0.2604	0.1941
rouge-3_r	0.2148	0.1797

greedy 會比起 sample 來的好，因為 sample 還是會有一定的機率 sample 到比較不好的選擇，造成後續 decode 的結果也不好。

beam search vs greedy

	greedy	n beams = 5
rouge-1_f	0.2417	0.2577
rouge-1_p	0.2615	0.2601
rouge-1_r	0.2379	0.2701
rouge-2_f	0.0928	0.1072
rouge-2_p	0.1004	0.1079
rouge-2_r	0.0917	0.1131
rouge-3_f	0.2281	0.2418
rouge-3_p	0.2604	0.2347
rouge-3_r	0.2148	0.2651

有 beam search 會比起 greedy 好，因為如果在前一輪 time step 選錯，會連帶影響後面的 time step，因此 beam search 可以多一層保險，防止該輪 time step 並非機率最高為最後的答案。

top k

	sample	top k = 50	top k = 10
rouge-1_f	0.1979	0.1979	0.2167
rouge-1_p	0.2067	0.2070	0.2286
rouge-1_r	0.2006	0.2006	0.2182
rouge-2_f	0.0661	0.0663	0.0766
rouge-2_p	0.0691	0.0696	0.0803
rouge-2_r	0.0673	0.0673	0.0777
rouge-3_f	0.1814	0.1821	0.2009
rouge-3_p	0.1941	0.1951	0.2193
rouge-3_r	0.1797	0.1801	0.1958

有使用 top k 會比原本的 sample 還要好，但如果 k 過大則會與原本的結果差不多。

top p

	sample	top p = 0.9	top p = 0.6
rouge-1_f	0.1979	0.2091	0.2274
rouge-1_p	0.2067	0.2200	0.2433
rouge-1_r	0.2006	0.2108	0.2252
rouge-2_f	0.0661	0.0739	0.0846
rouge-2_p	0.0691	0.0779	0.0908
rouge-2_r	0.0673	0.0746	0.0839
rouge-3_f	0.1814	0.1926	0.2112
rouge-3_p	0.1941	0.2081	0.2344
rouge-3_r	0.1797	0.1891	0.2025

有使用 top p 會比原本的 sample 還要好，也比起 top k 來的好，因為 output 通常是只有幾個很大其他很小，因此用 top p 會比較好找到候選的範圍，另外如果 p 設太小則會與 greedy 相當。

Temperature

	n beams = 5	n beams = 5 + Temperature = 0.7	n beams = 5 + Temperature = 1.3
rouge-1_f	0.2577	0.2578	0.2576
rouge-1_p	0.2601	0.2702	0.2700
rouge-1_r	0.2701	0.2601	0.2600
rouge-2_f	0.1072	0.1073	0.1072
rouge-2_p	0.1079	0.1131	0.1130
rouge-2_r	0.1131	0.1079	0.1079
rouge-3_f	0.2418	0.2419	0.2418
rouge-3_p	0.2347	0.2651	0.2650
rouge-3_r	0.2651	0.2348	0.2347

temperature 對於 beam search 的 rouge 結果影響不大，但在 generate 文字上，temperature=0.7 會比較多奇怪的句子，像是 id=22047 的結果「川普如何把「核足球控制權」移交給拜登? 川普如何把手中「核足球」移交給拜登?」會在標點符號後產生多餘的句子，可能是因為 sharpen 後會比較難產生 end 或是錯誤的詞機率上升導致後續很難 decode。

	top p = 0.6	top p = 0.6 + Temperature = 0.7	top p = 0.6 + Temperature = 1.3
rouge-1_f	0.2274	0.2385	0.2129
rouge-1_p	0.2433	0.2574	0.2246
rouge-1_r	0.2252	0.2347	0.2134
rouge-2_f	0.0846	0.0915	0.0746
rouge-2_p	0.0908	0.0990	0.0785
rouge-2_r	0.0839	0.0902	0.0750
rouge-3_f	0.2112	0.2231	0.1965
rouge-3_p	0.2344	0.2521	0.2140
rouge-3_r	0.2025	0.2113	0.1912

temperature 在 top p 影響較大，因為 temperature 會影響每個 time step 候選的數量，temperature<1，會將機率 sharpen，造成機率大的更大，小的更小，導致候選數量減少，更容易 sample 到機率大的，比較像 greedy；而 smooth 反之，導致機率平均分佈，因此會像是從 top p 中 random sample。