ADL HW3 @NTU, 2021 spring

B06902135 資工四 蔡宜倫

1. Model (2%)

- **Describe the model architecture and how it works on text summarization. (1%)**
  - MT5是一個為了tackle text-based 的NLP 問題而產生的 language model，他的全名是 `Multilingual Text-to-text Transfer Transformer`。
  - 這次NLG summarization的task我們使用`"goolge/mt5-small"`這個pre-trained model來實現，model 架構的configuration在下表呈現。
  - 這個model是用teacher-forcing的方式train，也就是decode完一個step後不以model的output作為next state的輸入而是直接以ground truth作為輸入，因為這個模型是設計成可以利用text產生text，也就是input和output都是文字。
  - MT5模型的input是一段文章，而利用summarization得到這段文章可能的title，我們提供模型這些資料作為training-data，並且fine-tune讓這個model可以fit到我們這次task的domain上。
  - MT5是一個auto-regressive的model，並且是根據word sequence的機率分佈可以由product of conditional next word distribution組成的假設。他根據前一個decode出來的token決定下一個要decode的token，另外在decode的時候也有一些不同的generate策略，包含greedy、beam search、top-k 和top-p sampling。
  - **Configuration:**

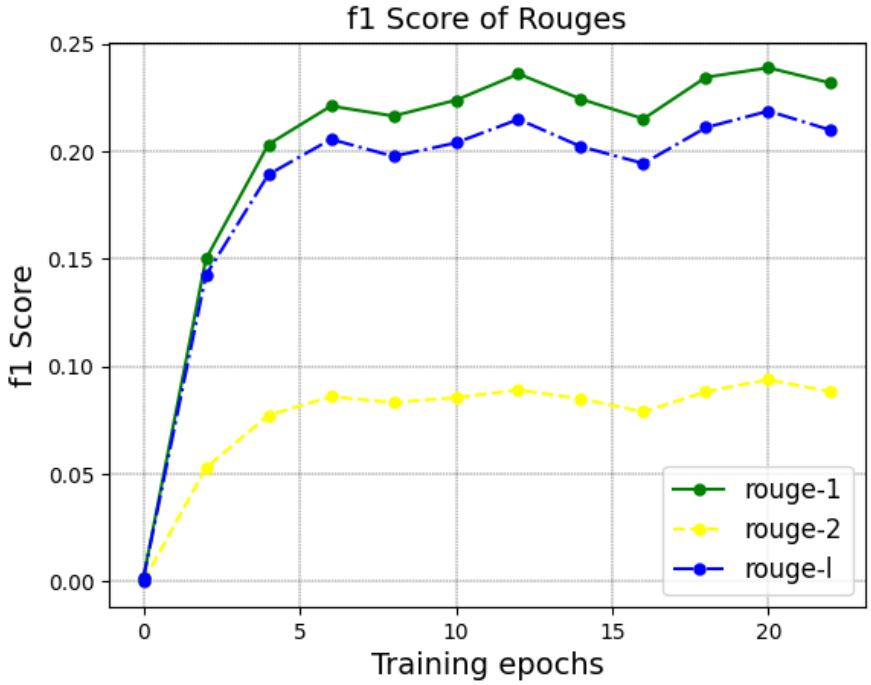
```
{
    "_name_or_path": "google/mt5-small",
    "architectures": [
        "MT5ForConditionalGeneration"
    ],
    "d_ff": 1024,
    "d_kv": 64,
    "d_model": 512,
    "decoder_start_token_id": 0,
    "dropout_rate": 0.1,
    "eos_token_id": 1,
    "feed_forward_proj": "gated-gelu",
    "initializer_factor": 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": "4.5.0",
    "use_cache": true,
    "vocab_size": 250112
}
```


- **Describe your preprocessing (e.g. tokenization, data cleaning and etc.) (1%)**
  - 我分別去tokenize了maintext和title，因為這兩者是模型的輸入，並且使用的即是`"google/mt5-small"`的tokenizer，設定上，我使用`truncation=True, max_length=args.max_len, padding=True`，這樣當輸入超過max_len我就會clip掉，而當不足的時候會做padding。
    - Maintext: 設定max_len為256 tokens。
    - Title: 設定max_len為64 tokens。

2. Training (2%)

- **Describe your hyperparameter you use and how you decide it. (1%)**
  - **Hyperparameters**:
    - Optimization Algorithm: `transformers.Adafactor`
    - Learning Rate: `1e-3` (fix learning rate)
    - Batch Size: `32`
    - Epoch: `50`
    - Gradient Accumulation Step: `16`
  - 選擇`transformers.Adafactor`作為optimizer是照投影片的建議用來減少GPU記憶體，而原本adafactor會根據training step調整learning rate，後來在網路上發現有人推薦固定learning rate可以有不錯的成效，發現`1e-3`可以取得比time-dependent learning rate更好的成績。
  - 因為我的GPU記憶體蠻大的，因此我調整batch_size讓GPU佔量到接近20G，並且通常在train到30 epoch就收斂了，但我有存下最低的loss的model，因此實際上可以不用跑到50 epochs。
- **Plot the learning curves (ROUGE versus training steps) (1%)**
  - learning curve是使用`"goolge/mt5-small`做在`public.jsonl`所得，並根據gram的不同算不同的rouge 的 f1 score。
  - 設定上我使用32作為batch_size，因此一個epoch有總training-data數量（21710）除以batch_size(32)個steps，因此一個epoch有679 training steps。
    - Highest F1 score of rouge-1: 24.1%
    - Highest F1 score of rouge-2: 9.36%
    - Highest F1 score of rouge-l: 22.0%



3. Curves (1%)

- **Strategies (2%)**
  - **Greedy**
    - 這是最heuristic的decoding策略，每次都選擇輸出機率最大的next word：$w_t = argmax_w P(w_t|w_{1:t-1})$。
    - 但是缺點是，有可能有一種word sequence的排列組合有更大的 P 乘積，但是卻因為有一個word的機率相對較小而沒有走到那條路，因此beam search出現，解決了這個問題。
  - **Beam Search**
    - Beam search的好處是會一次追蹤好幾個機率高的可能sequence，這樣就不會因為高機率的組合隱藏在一個低機率的word就miss掉，比如說若 `num_beams` 設定為5，則在每一個decoding step都會追蹤五個最大機率值的sequence，其餘的路則不考慮。
    - Beam search可以得到較好的結果，但因為一次要track的路線比較多，因此會耗費較長的時間。
  - **Top-k Sampling**
    - Top-k是一個sampling的方法，他的作用是redistribute下一個word的k個候選的probability mass，這樣可以刪掉probability比較低的字，因此model可以有比較好的decoding結果。
  - **Top-p Sampling**
    - Top-p是top-k的升級版，不同於top-k選擇固定數量的word作考慮，top-p是取某些字accumulated的機率沒有大於threshold p 的所有字去做decode。
    - 因為通常最好的結果可能會需要dynamically的去調整，不應直接用固定數量的字去做選擇，所以通過改變threshold改變考慮的候選字數量可以有更好的generation結果。
  - **Temperature**
    - Temperature這個strategy是用在改變整個probability distribution，讓distribution變得更

平滑或變得更集中
    - 平滑代表更有機會sample到機率比較小的字，但是得到的結果可能會太general沒辦法對應到在處理的task；集中則是會sample到最有可能的幾個word，這樣結果可能比較specific但是文法可能不太對。
    - 每一個token是根據這個公式去重新scale：$P(w_t|w_{1:t-1}) = \frac{e^{\frac{y_t}{T}}}{\Sigma_j e^{\frac{y_j}{T}}}$

- **Hyperparameters (4%)**
  - **Try at least 2 settings of each strategies and compare the result.**
    - 根據不同的generation策略得到不同的f1-score，如下表所示：

	Greedy	Beam Search				
		num_beam=3	num_beam=5	top_k=10	top_p=0.9	temp=0.5
ROUGE-1	0.2170	0.2359	0.2408	0.2388	0.2376	0.2229
ROUGE-2	0.0803	0.0904	0.0953	0.0924	0.0918	0.0817
ROUGE-l	0.2079	0.2145	0.2198	0.2198	0.2155	0.2030

    - **Greedy**：比較發現在有設定Beam search的情況下分數會比只用greedy的效果好。
    - **Beam Search**：再確定用beam search會比較好後，我比較了num_beam=3和5的情況，發現beam越大得到的結果越好，但相對的也會有比較久的計算時間。
    - **Top-k Sampling**：我比較了top-k為10和預設為50（num_beam=3那欄）的結果，發現考慮較多candidate的結果比較好。
    - **Top-p Sampling**：把p改為0.9取得了比預設1.0更高的rouge成績，一樣是和num_beam=3那欄去比較。
    - **Temperature**：我將temperature調大調小都取得了比較爛的成績，表上顯示temp=0.5的情況，可以看到也輸給預設temp=1的情況。
  - **What is your final generation strategy? (you can combine any of them)**

    改變top-k和top-p可能可以取得更好的成績，但是相對要花更多時間decode，因此我只用了搭一點的beam size去decode。
    - Num_beam: `5`
    - Top-k: `50`
    - Top-p: `1.0`
    - Temperature: `1.0`

4. Bonus: Applied RL on Summarization (2%)

- **Algorithm (1%)**
  - **Describe your RL algorithms, reward function, and hyperparameters.**
- **Compare to Supervised Learning (1%)**
  - **Observe the loss, ROUGE score and output texts, what differences can you find?**