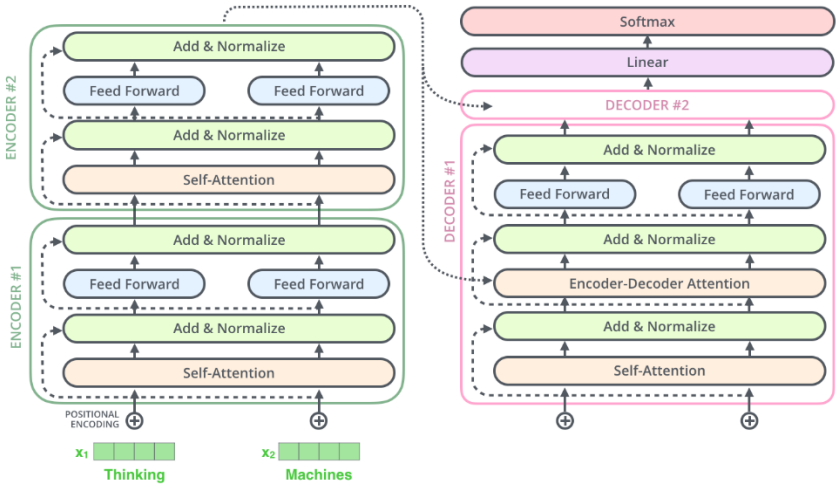
Q1: Model

1. Model



上圖為 Model 的 Architecture 示意圖 (From: http://jalammar.github.io/illustrated-transformer/)

Architecture: mt5 的 Architecture 與 t5 十分相似，都是一種 encoder-decoder transformer。

在 Encoder 端部分，有多個 Encoder Layer 組成。每個 Encoder Layer 中有兩個 sub layer，分別是 Self-Attention layer 及 Feed Forward NN。在 sub layer 之間會在 Add Residual 後，進行 layer Normalize 的動作。

在 Decoder 端部分，有多個 Decoder Layer 組成。每個 Decoder Layer 中有三個 sub layer，分別是 Self-Attention layer、Encoder-Decoder Attention layer、Feed Forward NN。Encoder 端的輸出會做為 Encoder-Decoder Attention

layer 的輸入，用於幫助 focus on important places in input sequence。在 sub layer 之間會在 Add Residual 後，進行 layer Normalize 的動作。

在最後加上 Linear Layer 及 Softmax Layer，將 Decoder 端輸出 Linear 後再經過 softmax 變成 probability 輸出。

WorkFlow:

在 data preprocessing 後，source text 轉變成 source sequences 作為 Encoder 的 input。

Encoder 的 Output 會用於幫助 focus on important places in input sequence。

在 Decoder 端，decoder 端的 input 會根據 step t 而有所不同，Decoder 會使用由 t=0 至 t=t-1 的 generated token 作為輸入，即為 step=t 前的 output token 作為 decoder 的 input。

Decoder 的 output 為 a vector of float，我們需要將此 output 透過一個 linear layer mapping to token set size。Mapping 後的值再透過 softmax 使其代表此 token 為 ans 的機率。

根據 ans 的機率以及 Q3 所討論的 Generation Strategies 生

成 Ans。

2. Preprocessing

Mt5 model 透過 sentencepiece 進行 tokenize。

我透過 truncation 設定 feautre 的 maximum length 為 256、target 的 maximum length 為 64，以降低 GPU 記憶體需求量。如長度不達 256，則會進行 padding 動作。我並沒有進行 data cleaning。

Q2: Training

1. Hyperparameter

我是根據 sample code 的 hyperparameter 去做修改。

(url:https://github.com/huggingface/transformers/tree/v4.24.0/examples/pytorch/summarization/run_summarization_no_trainer.py)

我修改了 max_source_length、max_target_length 以降低 GPU 使用量。

我修改了 per_device_train_batch_size、gradient_accumulation_steps，以降低 GPU 使用量同時能夠維持較高的 batch_size。

我新增了 warmup_ratio 用於提供一定比例的 steps 用於 warmup

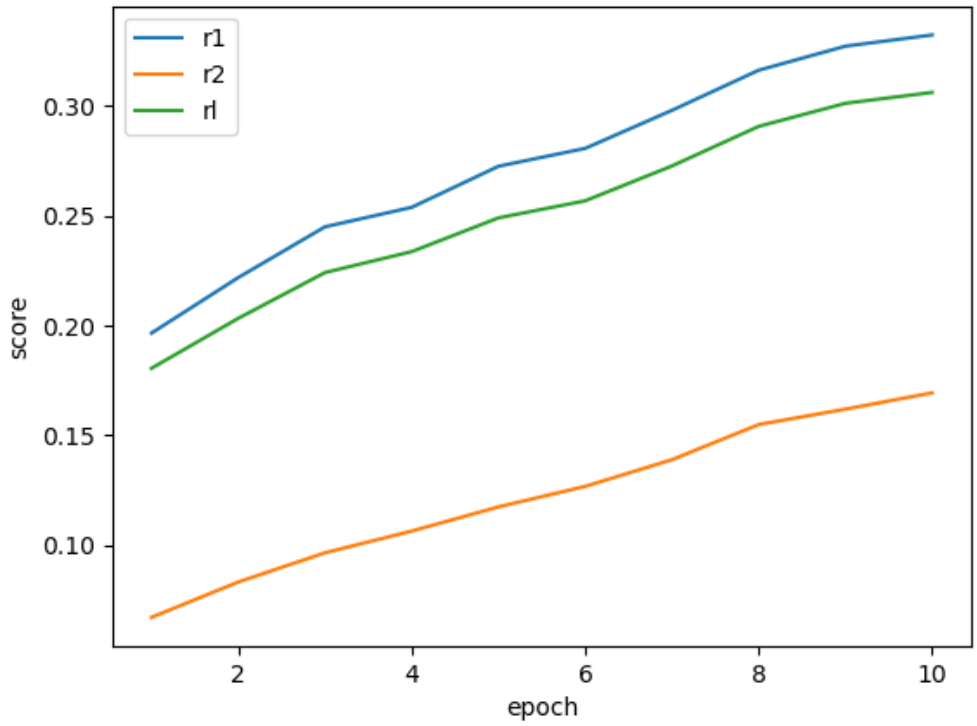
for scheduler。

我修改了 learning_rate，default_lr 為 5e-5，但在觀察了幾個 epoch 後，我發現 performace metrics 一直沒有太大變化，因此我選擇提升 lr 至 2e-3，同時觀察 10 個 epoch 的訓練情況。

以下是我的 Hyperparameter(不討論 Generation Strategies)：

1. max_source_length:256
2. max_target_length:64
3. per_device_train_batch_size:2
4. gradient_accumulation_steps:8
5. learning_rate:2e-3
6. num_train_epoch:10
7. lr_scheduler_type:linear
8. warmup_ratio:0.1(10%的 train_steps 用於 warmup)
9. optimizer:adamw

2. Learning Curves



我使用 train.jsonl 中的前 20%data 作為驗證集，Generation Strategy 為 Beam_sample。

Q3: Generation Strategies

1. Stratgies

1.Greedy:最簡單的 generation strategy，每次都選擇最高機率的 token 作為輸出。效果不佳，高機率重複性地出現相同字詞。

2.Beam Search:每次追蹤前 k 個最高機率的 path，最後選擇

機率最高的 path。如何設定 k 值是 beam search 最大的問題，如果設定太小，效果不佳。如果設定太大，計算時間會過長。

3.Top-k sampling:考慮前 k 個最高機率的 token，依據其機率選擇，使較低機率的 token 也有機會被選到。問題在於每次都選擇固定前 k 個最高機率的 token，可能選入機率非常低的 token，但其不小心被選到成為答案，使得辨識結果變差。

4.top-p sampling:將 token 依出現機率排序，由高至低累加機率值，直到累計機率值大於 p，將這些 token 定義為 token set，依據機率選擇 token。Top-p sampling 如同動態地調整 top-k sampling 中的 k 值，一定程度上避免選到機率非常低的 token。

5.Temperature:設定 temperature(T)改變 probability distribution，在 T 較高時會讓 probability distribution 更加平緩，讓選字結果 more diversity。在 T 較低時讓 probability distribution 更加 spiky，讓選字結果 less diversity。T 並不會影響 token 的 probability 的高低順序，只會影響 token 之間機率的差值。

2. Hyperparameters

1. Greedy:

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #0
        "top_p":args.top_p, #1.0
        "do_sample":args.do_sample, #False

}
```

以上為用於 Greedy 的 hyperparameter，因為 greedy 無法調整，因此只提供一種參數。

R1-score	R2-score	RL-score
[0.24029761234363498]	[0.08652187884918097]	[0.21548871413724327]

上圖為 Greedy Search 的結果。

2. Beam Search:

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #5
        "top_k":args.top_k, #0
        "top_p":args.top_p, #1.0
        "do_sample":args.do_sample, #False

}
```

以上為用於 Beam Search 的第一組參數。

R1-score	R2-score	RL-score
[0.2575639078777072]	[0.10108539007701262]	[0.23237538683517672]

上圖為 Beam Search 第一組參數的結果。

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #3
        "top_k":args.top_k, #0
        "top_p":args.top_p, #1.0
        "do_sample":args.do_sample, #False

}
```

以上為用於 Beam Search 的第二組參數。

R1-score	R2-score	RL-score
[0.2544770984868151]	[0.09822871141718005]	[0.22990413242195587]

上圖為 Beam Search 第二組參數的結果。

3. Top-k Sampling:

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #10
        "top_p":args.top_p, #1.0
        "do_sample":args.do_sample, #False

}
```

以上為用於 Top-k Sampling 的第一組參數。

R1-score	R2-score	RL-score
[0.2143135830707568]	[0.07028025312992127]	[0.1908928339148647]

上圖為 Top-k Sampling 第一組參數的結果。

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #25
        "top_p":args.top_p, #1.0
        "do_sample":args.do_sample, #False

}
```

以上為用於 Top-k Sampling 的第二組參數。

R1-score	R2-score	RL-score
[0.20358839770033182]	[0.06718703886551237]	[0.18131375698316662]

上圖為 Top-k Sampling 第二組參數的結果。

4. Top-p Sampling:

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #0
        "top_p":args.top_p, #0.5
        "do_sample":args.do_sample, #True
}
```

以上為用於 Top-p Sampling 的第一組參數。

R1-score	R2-score	RL-score
[0.22469707648214607]	[0.07863368441244742]	[0.20104698445833305]

上圖為 Top-p Sampling 的第一組參數的結果。

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #0
        "top_p":args.top_p, #0.8
        "do_sample":args.do_sample, #True
}
```

以上為用於 Top-p Sampling 的第二組參數。

R1-score	R2-score	RL-score
[0.20326364718557496]	[0.06869856429213875]	[0.1823876930593733]

上圖為 Top-p Sampling 的第二組參數的結果。

5. Temperature:

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #10
        "top_p":args.top_p, #1
        "do_sample":args.do_sample, #True
        "temperature":args.temperature, #0.5
}
```

以上為用於 Temperature 的第一組參數。

R1-score	R2-score	RL-score
[0.23091394980737556]	[0.08212670013672592]	[0.20740991048487672]

上圖為 Temperature 的第一組參數的結果。

```
gen_kwargs = {
        "max_length": args.val_max_target_length if args is not None else config.max_length, #64
        "num_beams": args.num_beams, #1
        "top_k":args.top_k, #0
        "top_p":args.top_p, #0.5
        "do_sample":args.do_sample, #True
        "temperature":args.temperature, #0.5
}
```

以上為用於 Temperature 的第二組參數。

R1-score	R2-score	RL-score
[0.23875883447424867]	[0.08521270235747302]	[0.21395482261136428]

上圖為 Temperature 的第二組參數的結果。

我最後根據這幾組參數的結果選擇了 Beam Search 作為我的生成 Summary 的方式，因為在 rogue 的評分方式中此方法表現最佳。

Bonus: Applied RL on Summarization