

# Report

資訊工程所碩一 R12922050 郭子麟

## Q1: Data Processing (2%)

### Tokenizer (1%)

- Describe in detail about the tokenization algorithm you use. You need to explain what it does in your own ways.

English Bert tokenizer 是使用 subword tokenization，但在 Chinese Bert 中，大致上使用 character 做切分。舉例來說：深度學習之應用就會被切成 {深,度,學,習,之,應,用}，而有些字可能常與其他字作組合，那就會是以 word-piece 分（通常在 token 表示會用 `##`）做前綴。

除了將中文做切分並轉成 token，我所使用的 tokenizer 是 load `bert-base-chinese`，而這個模型還會在斷詞中加入其他資訊。這次是參考以下連結的回答：

- `[CLS]`：在做分類任務時其最後一層的 repr. 會被視為整個輸入序列的 repr.
- `[SEP]`：有兩個句子的文本會被串接成一個輸入序列，並在兩句之間插入這個 token 以做區隔
- `[UNK]`：沒出現在 BERT 字典裡頭的字會被這個 token 取代
- `[PAD]`：zero padding 遮罩，將長度不一的輸入序列補齊方便做 batch 運算
- `[MASK]`：未知遮罩，僅在預訓練階段會用到

[https://leemeng.tw/attack\\_on\\_bert\\_transfer\\_learning\\_in\\_nlp.html](https://leemeng.tw/attack_on_bert_transfer_learning_in_nlp.html)

#### 進擊的 BERT：NLP 界的巨人之力與遷移學習

這篇是給所有人的 BERT 科普文以及操作入門手冊。文中將簡單介紹知名的語言代表模型 BERT 以及如何用其實現兩階段的遷移學習。讀者將有機會透過 PyTorch 的程式碼來直觀理解 BERT

🔗 [https://leemeng.tw/attack\\_on\\_bert\\_transfer\\_learning\\_in\\_nlp.html](https://leemeng.tw/attack_on_bert_transfer_learning_in_nlp.html)



## Answer Span (1%)

- How did you convert the answer span start/end position on characters to position on tokens after BERT tokenization?

在成功轉成 tokenizer 接受的資料格式後，tokenizer 在 tokenize 每個 sentence 後，會得到一個 `offset_mapping` property. 這個 offset mapping 其實就是對應每個被切開的中文 character，其開始與結束位置。

而 tokenization 也會將每個 character 轉成一個 token。因此在 tokenization 過程結束後，透過這個 offset mapping 機制就可以找到每個 character 其對應 token 的開始與結束位置。

這樣的話我們在原本 character-level 得到的 answer span 起始結束資訊也可以透過 offset mapping 去得到對應的 token-level 的位置資訊。

- After your model predicts the probability of answer span start/end position, what rules did you apply to determine the final start/end position?

有以下幾個步驟會納入考慮：

- 首先 model 會 output 每個 start index 及 end index 的機率，而我們會設定一個 `n_best_size` 的 threshold 只取最多為 `n_best_size` 個 start, end position 排列組合
- 我們也會設定 `max_answer_length`，如果說有一組 start, end index 組合超過這個大小，我們會把這個組合排除
- model 會有一個 `token_is_max_context` property。針對 start, end 組合出的 substring，如果其 token 沒有 maximum token available，我們也不考慮這個 substring
- 接下來我們再透過 `n_best_size` 確保可能的 substring 數量不超過這個 threshold
- 再來我們就 iterate through 每一個 substring，去計算其 score (e.g., EM)。這樣就可以知道 best start/end position 的組合所產生的 substring 是哪一個
- 最終的 final start/end position 就是 score 最高的

## Q2: Modeling with BERTs and their variants (4%)

### Describe (2%)

- Your model.

- The performance of your model.
- The loss function you used.
- The optimization algorithm (e.g. Adam), learning rate and batch size.

針對這次的 QA 任務，共有兩個模型：Multiple Choice 以及 Question Answering。我會分別敘述兩者的 model, loss function 以及 optimization algorithm，並說明使用這樣的配置，在 Kaggle 上的 performance 為何：

- Multiple Choice

- Model: 我使用的是 `transformers` 提供的 `AutoModelForMultipleChoice` 模型，而我所使用的 pre-trained LM 是 `bert-base-chinese`
- Performance: 這邊的計算方式是使用 accuracy，而在最後 evaluation 的結果如下：

```
{"eval_accuracy": 0.9571286141575274}
```

- Loss function: 因為 Multiple Choice 任務基本上就是 classification task，因此是使用 cross entropy 去計算 labels & predictions 的 loss
- Optimization Algorithm:
  - Optimizer: AdamW，並且將權重分為兩組，一組有 weight decay，另一組則沒有

以下是我在 training 時所使用的參數設定：

```
--max_seq_length=512 \
--per_device_train_batch_size=1 \
--gradient_accumulation_steps=2 \
--num_train_epochs=1 \
--learning_rate=3e-5 \
```

- Question Answering

- Model: 我使用的是 `transformers` 提供的 `AutoModelForQuestionAnswering` 模型，而我所使用的 pre-trained LM 是 `hf1/chinese-roberta-wwm-ext`

- Performance: 這邊的計算方式是使用 exact mach 以及 f1 score，而在最後 evaluation 的結果如下：

```
{
  "eval_exact_match": 82.38617480890662,
  "eval_f1": 82.38617480890662
}
```

- Loss function: Total span extraction loss is the sum of a Cross-Entropy for the start and end positions.
- Optimization Algorithm:
  - Optimizer: AdamW，並且將權重分為兩組，一組有 weight decay，另一組則沒有

以下是我在 training 時所使用的參數設定：

```
--max_seq_length=512 \  
--per_device_train_batch_size=8 \  
--gradient_accumulation_steps=2 \  
--num_train_epochs=3 \  
--learning_rate=3e-5 \  

```

- Performance: **0.78842** (public score on Kaggle)

## Try another type of pre-trained LMs and describe (2%)

- Your model.
- The performance of your model.
- The difference between pre-trained LMs (architecture, pretraining loss, etc.)

因為要替換不同的 pre-trained LM，因此我在兩個 models 上各自選了不同的 LMs:

- Multiple Choice: 因為我原本是使用 `bert-base-chinese`，在這邊就換成了 `hf1/chinese-roberta-wwm-ext`。而在其餘最佳化參數不變的情況下，local 跑完的 performance 為

```
{"eval_accuracy": 0.9594549684280492}
```

- Question Answering: 因為我原本是使用 `hfl/chinese-roberta-wwm-ext`，在這邊就換成了 `bert-base-chinese`。而在其餘最佳化參數不變的情況下，local 跑完的 performance 為

```
{  
  "eval_exact_match": 79.16251246261217,  
  "eval_f1": 79.18466821756952  
}
```

- Performance: **0.75587** (public score on Kaggle)

可以看出，performance 直接從 0.78842 掉到 0.75587。我認為主要的原因是因為在第一階段的 model (Multiple Choice)，因為 task 相對單純，因此僅使用 `bert-base-chinese` 也能達到跟 `roberta` 相近的效果。而在第二階段的模型，任務較為複雜。在原本使用 `hfl/chinese-roberta-wwm-ext` 的情況下，模型能夠處理的較好，因此 performance 不錯。但換成較簡單的 `bert-base-chinese` 後，performance 的落差就顯而易見

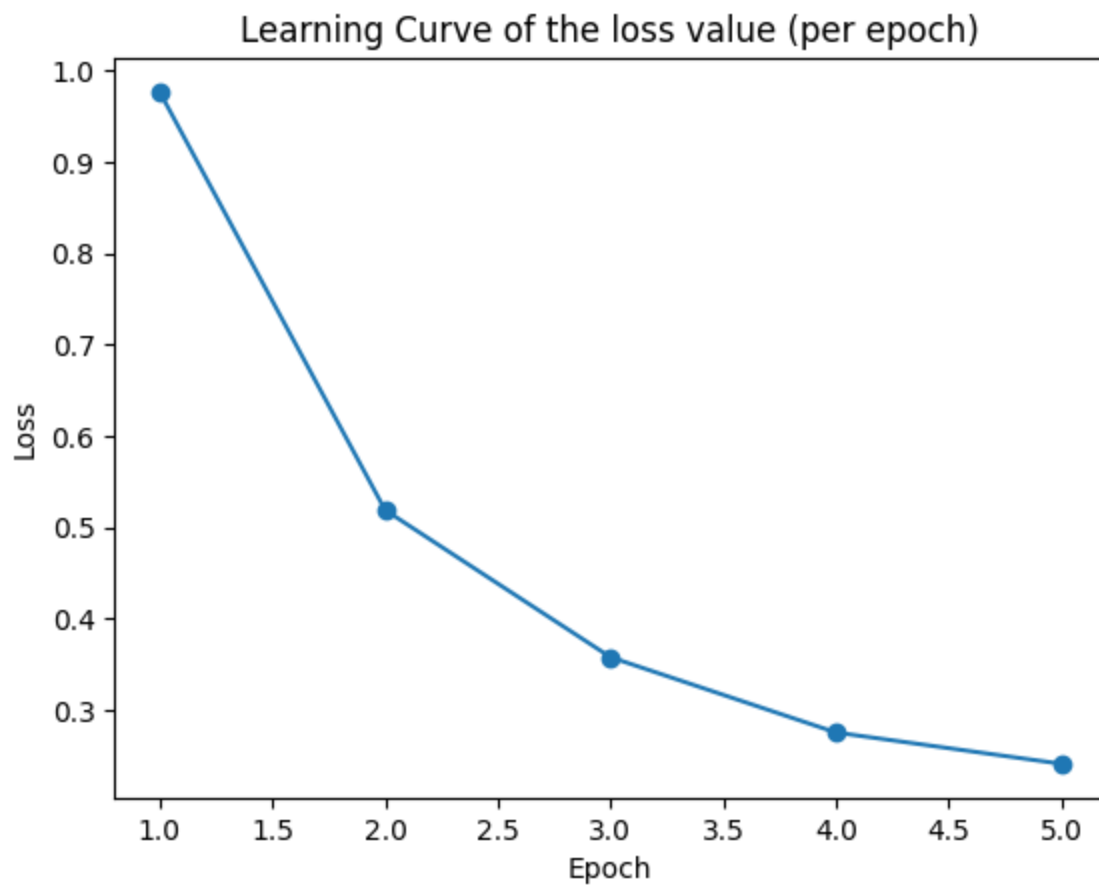
這邊主要就是比較了 `hfl/chinese-roberta-wwm-ext` 及 `bert-base-chinese` 的差別。我認為最主要的差異是：`hfl/chinese-roberta-wwm-ext` follows an optimized pre-training approach, such as larger batch sizes and longer training time。從上述可以想像成前者就是訓練較完善的 pre-trained LM，且就是特別為 Chinese 所設計，因此我認為這是造成其表現較好的原因。

### Q3: Curves (1%)

Plot the learning curve of your span selection (extractive QA) model. **You can plot both the training and validation set or only one set.** Please make sure there are at least **5 data points** in each curve.

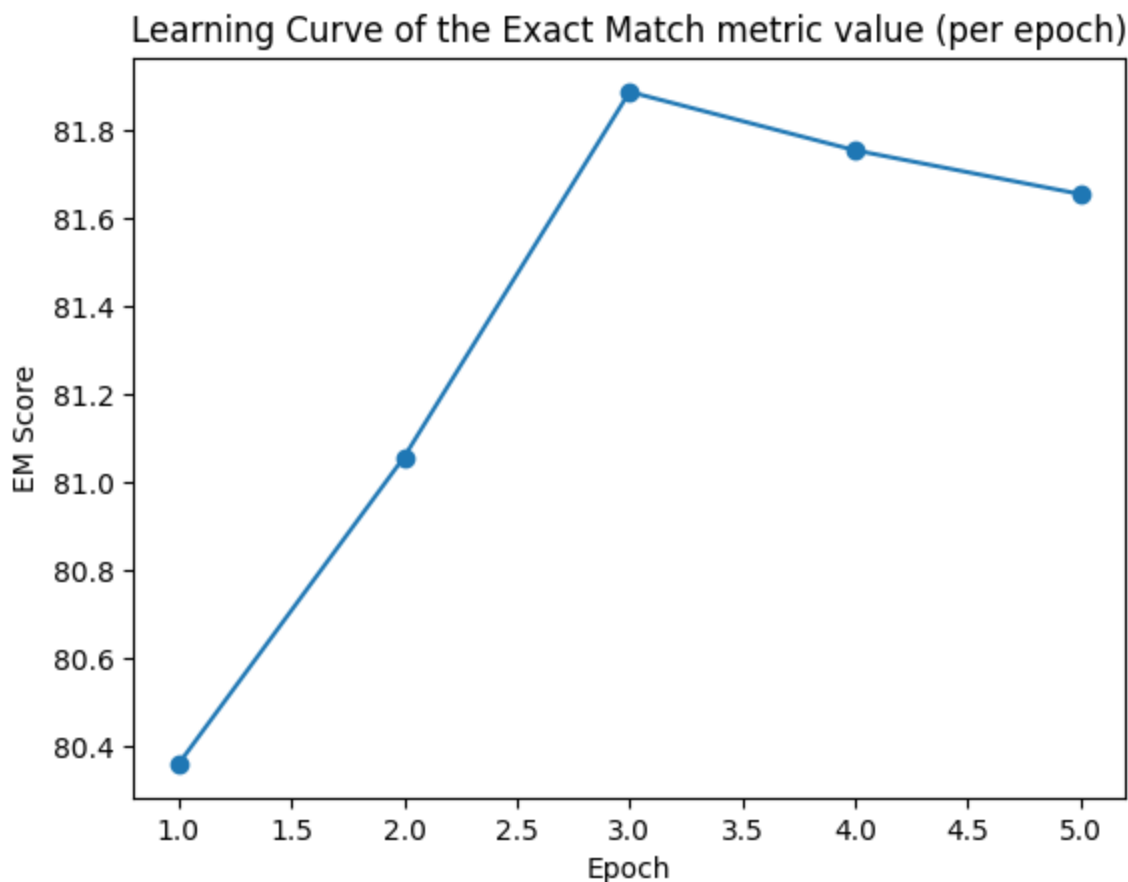
- Learning curve of the loss value (0.5%)

以下是使用每個 epoch 的 training loss 所繪製而成



- Learning curve of the Exact Match metric value (0.5%)

以下是使用每個 epoch 的 validation EM Score 所繪製而成



## Q4: Pre-trained vs Not Pre-trained (2%)

Train a transformer-based model (you can choose either paragraph selection or span selection) from scratch (i.e. without pre-trained weights).

### Describe

- The configuration of the model and how do you train this model (e.g., hyper-parameters)

我是使用 span selection model (QA model) 搭配 no pre-trained LM 與 pre-trained LM 進行 performance 比較。針對沒有使用的 pre-trained model，我是使用 default 的 `BertConfig` 如下：

```
{  
  "vocab_size": 30522,  
  "hidden_size": 768,  
  "num_hidden_layers": 12,  
  "num_attention_heads": 12,
```

```

    "intermediate_size": 3072,
    "hidden_act": "gelu",
    "hidden_dropout_prob": 0.1,
    "attention_probs_dropout_prob": 0.1,
    "max_position_embeddings": 512,
    "type_vocab_size": 2,
    "initializer_range": 0.02,
    "layer_norm_eps": 1e-12,
    "pad_token_id": 0,
    "position_embedding_type": "absolute",
    "use_cache": true,
    "classifier_dropout": null,
    "***kwargs": null
}


```

各 parameter 的解釋及說明可以參考連結如下：

[https://huggingface.co/docs/transformers/model\\_doc/bert#transformers.BertConfig](https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertConfig)

## BERT

We're on a journey to advance and democratize artificial intelligence through open source and open science.

 [https://huggingface.co/docs/transformers/model\\_doc/bert#transformers.BertConfig](https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertConfig)



而這個模型的訓練方式（optimization）是使用

- AdamW optimization algorithm: variant of the Adam optimizer that includes weight decay regularization
- 其餘 optimization 參數與原本的模型（pre-trained）相同：

```

--max_seq_length=512 \
--per_device_train_batch_size=8 \
--gradient_accumulation_steps=2 \
--num_train_epochs=3 \
--learning_rate=3e-5 \

```

- The performance of this model v.s. BERT.

當使用 no pre-trained QA model 進行訓練，得到的 local evaluation 結果（validation set）如下：



```
{
  "eval_exact_match": 3.5892323030907276,
  "eval_f1": 3.5892323030907276
}
```

而丟到 Kaggle 上的 public score 則為 **0.05515**，可以看出 performance 差距非常大。我認為原因是因為沒有 pre-trained 而單獨使用我們的資料集做訓練，training set size 實在是太小，沒有辦法得出每個 token 適合的 representation embeddings。

#### Hint

- You can use the same training code, just skip the part where you load the pre-trained weights.
- The model size configuration for BERT might be too large for this problem, if you find it hard to train a model of the same size, try to reduce model size (e.g. num\_layers, hidden\_dim, num\_heads).

## Q5: Bonus (2%)

- Instead of the paragraph selection + span selection pipeline approach, train a **end-to-end** transformer-based model and describe
  - Your model.
  - The performance of your model.
  - The loss function you used.
  - The optimization algorithm (e.g. Adam), learning rate and batch size.
- Hint: Try models that can handle long input (e.g., models that have a larger context windows)

## End to End 模型處理及訓練方式

我的作法是基於 Question Answering Model 任務做延伸，將原先透過 Multiple Choice 選出 relevant paragraph，並將其視為 QA context 的方式換成：在每次 Training，

context 其實就是四個可能 context 的 concatenation。並且在這麼長的 sequence 下作 answer span 的標記訓練。

而根據上述，為了要讓模型是 end-to-end，代表我們 concat 過後的 context span 是非常長的，必須要選擇有能力 handle long input context window 的模型會比較適合。因此在上網查找後，我發現 `hfl/chinese-xlnet-base` 符合我的需求，因為其主張他是眾多模型中少數支援 no sequence length limit 的模型。

但因為我的主機規格上有所限制，一次 batch 不能 Load 太大的 data，因此如果真的讓 tokenizer 切全部的 context，可能會發生 context concat 過後實在太長，資料量太巨大導致 CUDA error 的問題。因此我這邊選擇設定 `max_seq_length` 從 512 (`bert-base-chinese` 最長的限制) 到 2048。

而針對標記資料，原先提供的 answer start index 都是只看 relevant span 的 index start position。但我們現在要 concat 資料，因此必須重新設定 answer start position。

我是透過在 load datasets 後進行 mapping 的動作去計算每筆資料真實的 answer start，也就是 relevant paragraph 前面 concat 的長度都要加進去。演算法如下：

```
def count_true_answer_start(example):
    ans_start = example["answer"]["start"]
    start = 0
    for paragraph_id in example["paragraphs"]:
        if paragraph_id != example["relevant"]:
            start += len(context[paragraph_id])
        else:
            start += ans_start
            break
    example["answer"]["start"] = start
    return example
```

## Model Description

- Model: 我使用的是 `transformers` 提供的 `AutoModelForQuestionAnswering` 模型，而我所使用的 pre-trained LM 是 `hfl/chinese-xlnet-base`
- Loss function: Total span extraction loss is the sum of a Cross-Entropy for the start and end positions.

- Optimization Algorithm:
  - Optimizer: AdamW，並且將權重分為兩組，一組有 weight decay，另一組則沒有

以下是我在 training 時所使用的參數設定：

```
--max_seq_length=2048 \  
--per_device_train_batch_size=1 \  
--gradient_accumulation_steps=4 \  
--num_train_epochs=1 \  
--learning_rate=3e-5 \  

```

- Performance: **0.78842** (public score on Kaggle)