Report - ADL HW2

網媒所碩一 R11944026 柯婷文

Q1. Data Processing

1. Tokenizer (1%):

實作上
使用這行code來呼叫hugging face上pretrained model的tokenizer。

```
tokenizer = AutoTokenizer.from_pretrained(model_args.model_name_or_path,use_fast=True)
```

由於我們的參數包括`use_fast`，所以得到的BERT tokenizer實際上是一個基於Rust的fast tokenizer，相較於BertTokenizer更快。
透過觀察transformers.PreTrainedTokenizer以及BertTokenizerFast這兩個libray的source code，可以發現實作上由`tokenize()`以及`convert_tokens_to_ids()`這兩個method實現tokenizer的主要功能：

> Converts a string in a sequence of tokens (string), using the tokenizer. Split in words for word-based vocabulary or sub-words for sub-word-based vocabularies (BPE/SentencePieces/WordPieces).

步驟是
- 將現有的special token(如`<unk>`, `<cls>`...)跟新加入的special token(可以以參數傳入)取聯集，存在`unique_added_tokens_encoder`
- 分別用每個special token `tok`來切分整段文字，並將結果更新在`text_list`；接著，用下一個`tok`切分更新過的`text_list`。整個過程迭代直到所有`tok`都輪完為止。
- 最後再從字典裡找token對應的id，就完成了。
經過上述流程後，每個token對應的ID並會存在`input_ids`；其他回傳值包括

- `token_type_id`，如果是兩句話中的第一句話則為0，第二句話則為1
- `attention_mask`非padding部分為1；padding的部分為0，代表在計算attention時不用去考慮

觀念上

bert tokenizer主要有這兩個重要觀念：

- **WordPiece Tokenization**。做法是先把字分成subword unit，而原本是同個字的token以##作為識別，比如Sleeping就可能被切成Sleep和##ing。之後再去字典裡找token對應的ID，如此一來便可以提升token出現在vocab的機率。
- **特殊token**：針對一個由多個句子組成的文章段落，開頭會用[CLS]標示，句與句之間則用[SEP]連接。[UNK]用來表示沒出現在字典的token；padding部分用[PAD]，預訓練時使用[MASK]做MLM。這些token一樣會被轉成字典裡的ID。
  而 `hfl/chinese-roberta-wwm-ext` 進一步使用了 **Whole Word Masking\(wwm\)** 則是指用MLM任務預訓練模型時，如果word中任何一個token被mask了，那整個word的token也都會被mask。
  範例如下，參考自 中文BERT-wwm系列模型
  備註：在中文裡一個word指的是一個詞，character指的是一個字。

说明	样例
原始文本	使用语言模型来预测下一个词的probability。
分词文本	使用 语言 模型 来 预测 下 一个 词 的 probability 。
原始Mask输入	使 用 语 言 [MASK] 型 来 [MASK] 测 下 一 个 词 的 pro [MASK] ##lity 。
全词Mask输入	使 用 语 言 [MASK] [MASK] 来 [MASK] [MASK] 下 一 个 词 的 [MASK] [MASK] [MASK] 。

2. Answer Span (1%):

a.

在處理QA問題時，預測結果會是一段長度不等的文字，也可說是一個區間內的文字。所以這次模型要輸出的不是token，而是token起始位置、結束位置。將資料傳給tokenizer時，設定 `return_offsets_mapping=True`，回傳值便會包括`offsets_mapping`，紀錄的是每個word的起始、結束位置。值得注意的是，這裡會按照token原本出現在text的順序排。

text	offset_mapping
\[CLS\] how many wins ...	\[(0, 0), (0, 3), (4, 8), (9, 13),...]

因為資料集提供的是answer_start跟text兩個欄位，所以我們先用text的長度計算answer_end的位置。如此一來便可得知答案的char-level start & end position，分別稱作`start_char`和`end_char`。接著，我們要找答案的token-level start&end position。

1. 先初始化`token_start_index`為0、`token_end_index`為最末端index。
2. 迭代offset_mapping，從第一個token的offset_mapping開始檢查，比較現在這個token的第一個char的位置(`offsets[token_start_index][0]`)是否小於等於`start_char`，若是，就將`token_start_index`+1，並往下一個token檢查，直到token的頭已經在`start_char`的右邊為止。
3. 反之，從尾部開始檢查這個token的最後一個char的位置(`offsets[token_end_index][1]`)是否大於等於`end_char`，一路往左算回來直到token尾已經在`end_char`的左邊為止。
4. 最終可以得到`token_start_index`、`token_end_index`，就是答案的token-level start&end position。

b.
首先我們可以從trainer的outputs直接得到兩個tensor分別為`start_logits` `end_logits`，分別代表了最有可能的起始位置跟結束位置。然而因為是分開去預測的，所以有可能發生起始位置大於結束位置這種不合理的情況。於是我們分別取前`n_best`個logit出來，排除掉前面所說不合理的index組合後，對於所有剩下的組合，將他們機率相加，作為分數，最終取分數最高的組合，作為答案輸出。

Q2. Modeling with BERTs and their variants

My model - chinese-roberta-wwm-ext-large

```json
{
  "_name_or_path": "hfl/chinese-roberta-wwm-ext",
  "architectures": [
    "BertForMultipleChoice"
  ],
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "classifier_dropout": null,
  "directionality": "bidi",
  "eos_token_id": 2,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "output_past": true,
  "pad_token_id": 0,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```

```json
{
  "_name_or_path": "hfl/chinese-roberta-wwm-ext-large",
  "architectures": [
    "BertForQuestionAnswering"
  ],
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "classifier_dropout": null,
  "directionality": "bidi",
  "eos_token_id": 2,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 1024,
  "initializer_range": 0.02,
  "intermediate_size": 4096,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "output_past": true,
  "pad_token_id": 0,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```


Performance

train_loss: 0.3849062052514726,

eval_exact_match: 84.31372549019608,

eval_f1: 84.31372549019608,

Kaggle public: 0.81029

Kaggle private: 0.81103

Loss function

CrossEntropyLoss()

Optimization Algorithm

AdamW (default)

learning rate

3e-5

batch size

8 = (per_device_train_batch_size) 2 * (gradient_accumulation_steps) 4

Another pretrained model: macbert

```json
{
  "_name_or_path": "hfl/chinese-macbert-base",
  "architectures": [
    "BertForMultipleChoice"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "directionality": "bidi",
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```

```json
{
  "_name_or_path": "hfl/chinese-macbert-large",
  "architectures": [
    "BertForQuestionAnswering"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "directionality": "bidi",
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 1024,
  "initializer_range": 0.02,
  "intermediate_size": 4096,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "pad_token_id": 0,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```


Performance

kaggle public: 0.8056

kaggle private: 0.80758

eval_exact_match: 83.5493519441675,

eval_f1: 83.5493519441675,

train_loss: 0.5261030852409805,

兩者差異

	Roberta-wwm	Macbert
Tokenization	Whole Word Masking (wwm) (更適合中文資料)	WordPiece

與Bert相比，Macbert在預訓練時引入MLM as correction的任務，讓預訓練的任務跟實際任務更為符合。macbert使用n-gram masking，並用相似詞來代替MASK，解決pretrain時有mask，而實際任務沒有的問題。

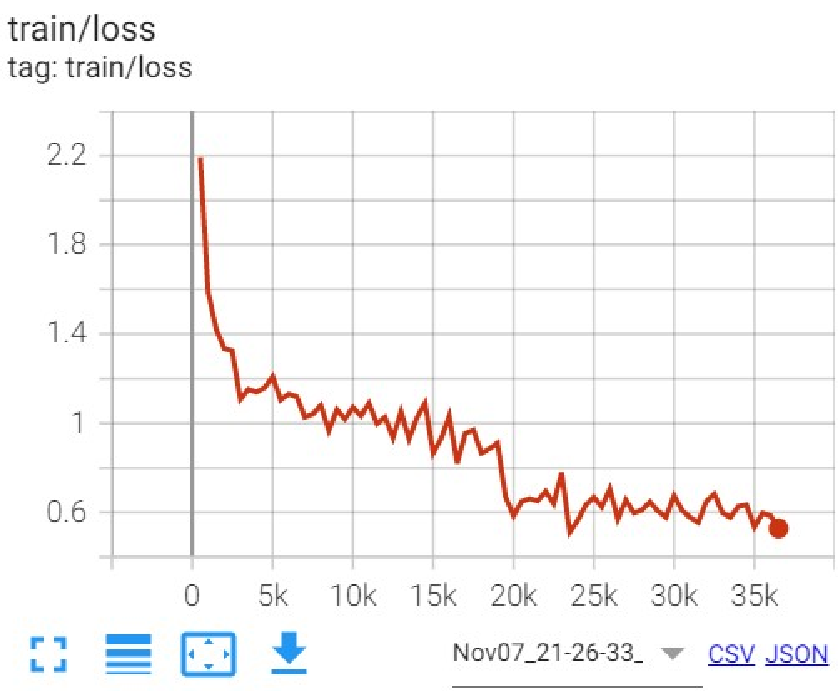
	例子
原始句子	we use a language model to predict the probability of the next word.
MLM	we use a language [M] to [M] ##di ##ct the pro [M] ##bility of the next word .
Whole word masking	we use a language [M] to [M] [M] [M] the [M] [M] [M] of the next word .
N-gram masking	we use a [M] [M] to [M] [M] [M] the [M] [M] [M] [M] [M] next word .
MLM as correction	we use a text system to ca ##lc ##ulate the po ##si ##bility of the next word .

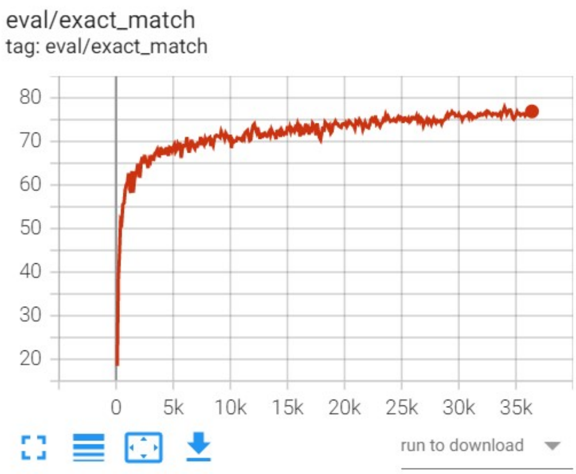
而Roberta跟Bert的差異在於

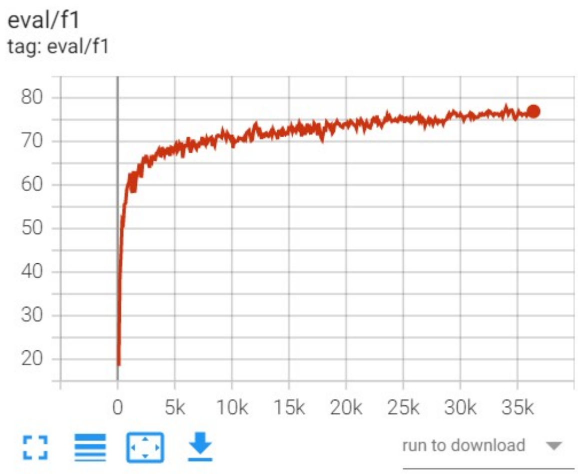
- bert的masking是在pretrain就做了，這樣一來每個epoch的masking都一樣，而macbert使用Dynamic Masking，只在sequence送入模型中的時候才去進行動態的masking
- 移除了Bert裡面的next sentence prediction
- 使用更長的句子、更大batch size、更多資料進行訓練

Q3. Curves

By setting `--with_tracker` option, hugging face trainer automatically exports the results to tensorboard etc. Therefore the charts below are captured from tensorboard.







Q4. Pretrained vs Not Pretrained


```json
{
  "_name_or_path": "bert-base-chinese",
  "architectures": [
    "BertForMultipleChoice"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "directionality": "bidi",
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```


我將context selection的部分做調整，加入一個新的command line argument `--`

`no_pretrain`，如果有設定這個選項則在load model時不使用原本的`from_pretrained()`，而是改用`from_config()`，如此一來便不會load pretrained weight。值得一提的是config跟tokenizer仍然需要使用`from_pretrained()`，因為我們還是需要模型架構跟tokenizer。

```
# old
model = AutoModelForMultipleChoice.from_pretrained(args.model_name_or_path)
# new
model = AutoModelForMultipleChoice.from_config(config)
```

the performance of this model v.s. BERT

eval_accuracy比較

No pretrain	Pretrain
0.5267530741110004	0.9594549684280492

Q5. Bonus: HW1 with BERTs


```json
{
  "_name_or_path": "bert-base-cased",
  "architectures": [
    "BertForSequenceClassification"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "problem_type": "single_label_classification",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 28996
}
```

```json
{
  "_name_or_path": "bert-base-uncased",
  "architectures": [
    "BertForTokenClassification"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "finetuning_task": "ner",
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "torch_dtype": "float32",
  "transformers_version": "4.22.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

(這張圖刪除了label2id, id2label的部分，不然貼不下)

Performance

Intent classification

```
"eval_accuracy": 0.8756666779518127,
"eval_loss": 1.9993304014205933,
"train_loss": 3.157837913926171,
```

Slot tagging

```
"eval_loss": 0.09548604488372803,
"eval_token_accuracy": 0.9679340937896072,
"train_loss": 0.1432706798825945,
```

Loss function

都是CrossEntropyLoss

Optimization

都是AdamW

lr

5e-5 (default)

batch_size

64 = s8 (per_device_train_batch_size) * 8 (gradient_accumulation_steps)