ADL HW 2

B08902134, 曾揚哲

1 Data Processing

1.1 Tokenizer

I use the 🤗 `BertTokenizer` pretrained with the model used for training. The tokenizer converts the word into tokens which could be recognized by `BERT` model. The tokenizer does the following:

1. Convert each word into token represented by their respective unique ID, while division into subwords is used in western languages, Chinese tokens cannot be cut into subwords.
2. Add specific tokens such as `[CLS]`,`[SEP]` and `[UNK]` to represent the classification token, separation token and the unknown words during pretrain phase of the tokenizer, respectively.
3. Pad, truncate or split the context into lists in a task-oriented manner, which is done by the 🤗 tokenizer itself.

1.1.1 Context Selection

Each data consist of a question and four relevant context, and they are tokenized in the following manner:
What I intended to do originally:

`[CLS]` tokenized question... `[SEP]` tokenized context 1... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[SEP]` tokenized context 2... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[SEP]` tokenized context 3... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[SEP]` tokenized context 4... `[SEP]` `[PAD]`...

But I write the code wrong so it is actually processed as:

`[CLS]` tokenized question... `[space]` tokenized context 1... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[space]` tokenized context 2... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[space]` tokenized context 3... `[SEP]` `[PAD]`...
`[CLS]` tokenized question... `[space]` tokenized context 4... `[SEP]` `[PAD]`...

But in the end it turns out to work (the performance is not bad?)!

1.1.2 Question Answering

Each data consist of a question and the selected context, and they are tokenized and truncated in the following manner:

`[CLS]` tokenized question... `[SEP]` tokenized context part 1... `[SEP]`
`[CLS]` tokenized question... `[SEP]` tokenized context part 2... `[SEP]`
⋮
`[CLS]` tokenized question... `[SEP]` tokenized context part n... `[SEP]` `[PAD]`...

I use the tokenizer with a `doc_stride` of 128 tokens so that it could cover the longest answer, and only the context would be truncated, which could be done by passing `truncation="only_second"` to the 🤗 tokenizer.

1.2 Answer Span

1.2.1 Preprocessing

While the 🤗 tokenizer does the answer span processing for us, we could still inspect what it does for us. As mentioned in **1.1.2**, only the context would be truncated while a stride of it would be preserved in each vector. To elaborate on the detail, we here denote the maximum length of question and context pair, the length of the question, the length of the context part in each vector, and the padding length by L, L_q, L_c, L_p, respectively. Here, we assume that $L_q <$ `doc_stride` and the whole question context pair is divided into n vectors. Therefore, we have

$$L = 1 + L_q + 1 + L_{c,i} + 1, 1 \leq i < n$$
$$L = 1 + L_q + 1 + L_{c,n} + 1 + L_p$$

For `doc_stride` t, $L_{c,i-1}[t:] = L_{c,i}[:t], \forall 1 < i \leq n$.
To contain the largest question answer pair (ideally according to the training set), I set `doc_stride` $= 128$. Then we could find out that the each vector contains context of length $L_c' = L - 3 - L_q$, and hence the i-th vector will contain context$[i * (L_c' - \texttt{doc_stride}) : i * (L_c' - \texttt{doc_stride}) + L_c']$. Given these information, it is easy to find the original start/end position in these vectors. Notice that this is easy because we are not requried to cut the context into subwords during tokenization because we are working on a Chinese dataset.

1.2.2 Postprocessing

After predicting the start/end positions in each vector, I take the `n_best`$= 20$ probabilties from the logits and exclude those positions that give

- answer not in the context
- ansewr with negative length, i.e., `end_position` < `start_position`
- answer that is too long, i.e. `L_a` > 30

After finding the valid (`start_position`, `end_position`) pairs, I would return the answer who has the highest probabilties at the start and the end, theoretically multiplied, but I used summed assuming log are taken for logits score.
The predicted answer is then slightly modified as some brackets may not be closed after the prediction. For Chinese, I added (「,」, 《,》) to complete the bracket if the complement present. Moreover, the half-width comma (i.e. ’,’) was removed from the answer to prevent kaggle misintepreting it as the csv separator. With the bracket completion, the exact match on test dataset improved from 0.79746 to 0.80018 (pretrained model with best performance).

2 Modeling with BERTs and their variants

Note: the reported metrics are recorded from the last epoch, while the result on kaggle was not necessarily inference with checkpoint from the last epoch.

2.1 Baseline Model

1. My model: 🤗 `bert-base-chinese`
   Configuration:
   - Hidden size: 768
   - Hidden Layers: 12
   - Attention Heads: 12
   - Intermediate Size: 3072
   - `max_len`: 512
   - `doc_stride`: 128
2. Performance

	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9988	0.0040	0.9629	0.1503
Question Answering	0.9769	0.0412	0.7988	0.9172

   Result on kaggle public test dataset: 0.75406 (Strong Baseline: 0.75497)
3. Loss Function: CrossEntropy Loss
4. Optimization:
   - Algorithm: AdamW with $\texttt{lr} = 2 \times 10^{-5}$ and weight decay $= 1 \times 10^{-6}$
   - Learning Rate Scheduler: Cosine Annealing Scheduler with $1 : 9$ steps for warmup and training

2.2 Model with best performance

1. My model: 🤗 `hfl/chinese-macbert-large`
   Configuration:
   - Hidden size: 1024
   - Hidden Layers: 24
   - Attention Heads: 16
   - Intermediate Size: 4096
   - `max_len`: 512
   - `doc_stride`: 128

2. Performance

	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9914	0.006363	0.9668	0.1368
Question Answering	0.9925	0.003711	0.8404	0.7389

   Result on kaggle public test dataset: 0.79746 → 0.80018 as mentioned in **1.2.2**

3. Loss Function: same as baseline model
4. Optimization: same as baseline model

2.3 Other models

In this section, the hyperparameters are mentioned only if they are different from previous setting. The loss function and optimization algorithms are all the same. Moreover, most of these are not tested on test dataset, i.e. only metrics for each task on training and validation set is reported.

1. 🤗 `hfl/chinese-macbert-base`
   Performance

	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9979	0.0030	0.9625	0.1877
Question Answering	0.6753	0.0600	0.8173	0.8664

   Result on kaggle public test dataset: 0.78028 (only this one is inferenced)

2. 🤗 `hfl/chinese-roberta-wwm-ext-large`
   Performance:

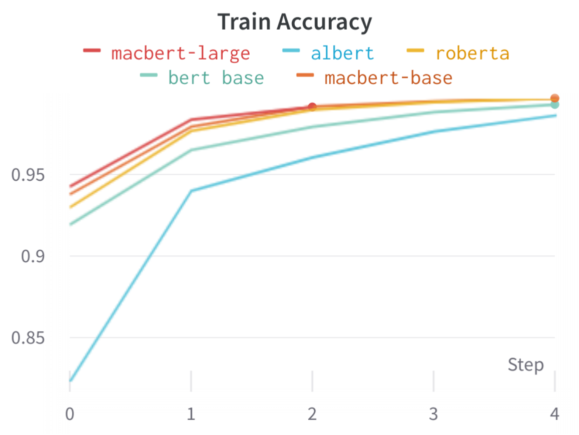
	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9967	0.0029	0.9576	0.1651
Question Answering	0.7467	0.4127	0.8003	0.8414

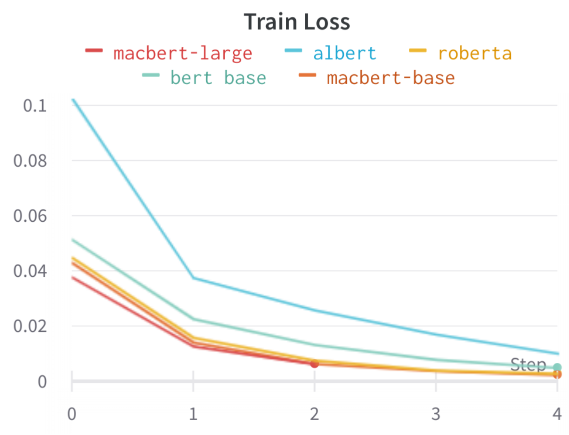
3. `ckiplab/albert-tiny-chinese`
   Performance:

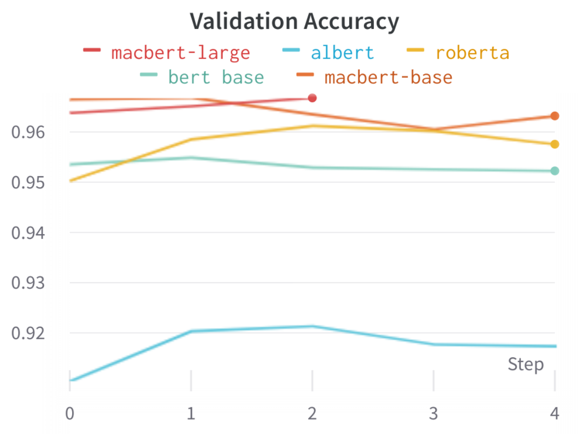
	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9996	0.0003	0.9105	0.7017
Question Answering	0.9268	0.1493	0.5717	2.0880

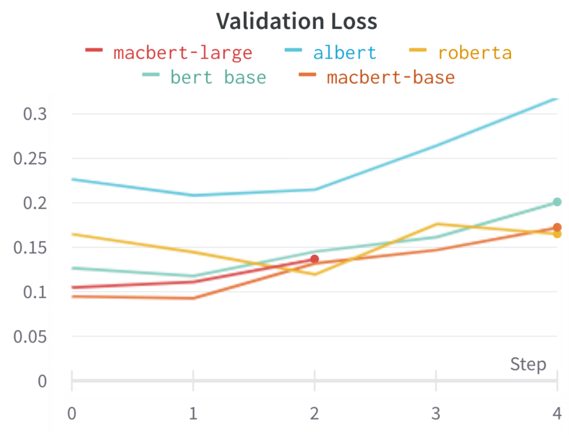
3 Curves

3.1 Context Selection

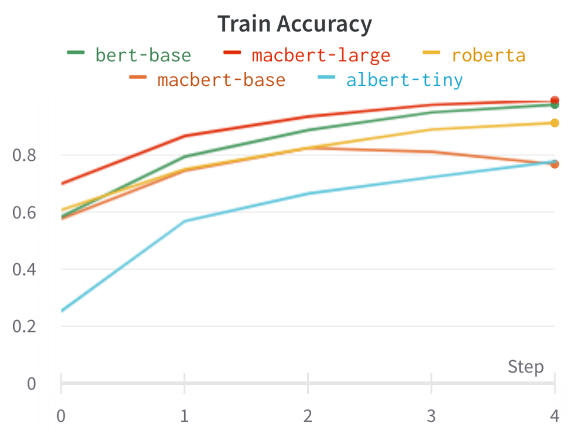


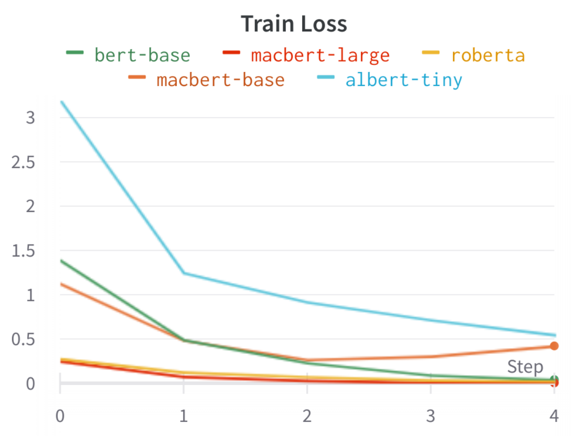


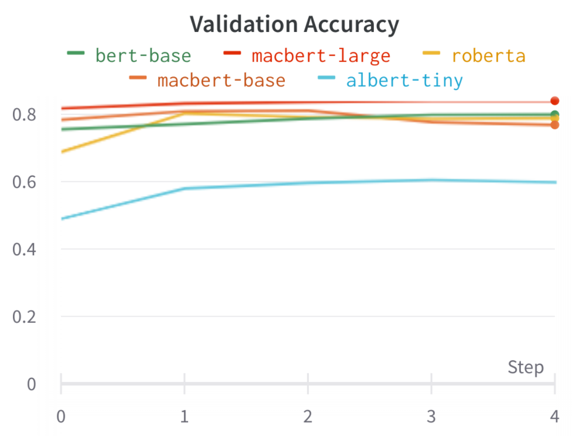


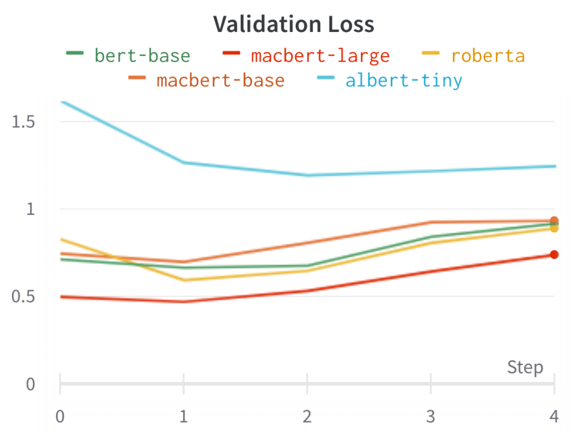


3.2 Question Answering









4 Pretrained vs From Scratch

4.1 Configuration

The configuration on both tasks are the same. Configuration:

- Hidden size: 384
- Hidden Layers: 6
- Attention Heads: 6
- Intermediate Size: 1024
- `max_len`: 384
- `doc_stride`: 128

4.2 Performance

The best accuracy and the lowest loss are reported.

	Train Acc. / EM	Train Loss	Val. Acc. / EM	Val. Loss
Context Selection	0.9681	0.0534	0.35	3.61
Question Answering	0.9961	0.0170	0.54	0.9092

4.3 Comparison

The model was trained from scrtach by simply removing the process of loading pretrained weight, who fits well on the training set but appears to be overfitting after around the tenth epoch on context selection task, and the weird thing is that the validation loss has been increasing since the first epoch from the first epoch to the last one in question answering. I thought this BERT was small enough but it still overfits, so I tried a smaller one with the following configuration, but the result is quite the same.

4.3.1 Even smaller BERT

- Hidden size: 256
- Hidden Layers: 4
- Attention Heads: 2
- Intermediate Size: 512
- `max_len`: 256
- `doc_stride`: 64

5 HW1 with BERTs

5.1 Model

I used the widely-used RoBERTa (`roberta-base`) pretrained on pretrained on wikipedia and bookcorpus with the following configuration.

5.1.1 Configuration

- Hidden size: 768
- Hidden Layers: 12
- Attention Heads: 12
- Intermediate Size: 3072
- `max_len`: 512

5.2 Performance

The metrics reported are captured from the last epoch which is also the best. I didn't save the fine-tuned model so its performance on test dataset is not reported.

5.2.1 Intent Classification

	Train Acc.	Train Loss	Val. Acc.	Val. Loss
HW1	0.9942	0.01793	0.9332	0.7642
Transformer	0.9887	0.1387	0.9701	0.1743

5.2.2 Slog Tagging

	Train Acc.	Train Loss	Val. Acc.	Val. Loss
HW1	0.955	0.001362	0.838	0.01562
Transformer	0.8096	0.0071	0.8290	0.0073

5.3 Loss Function

I used cross entropy loss for these two tasks.

5.4 Optimization

Exactly same setting as previously mentioned, AdamW optimizer with learning rate of 3×10^{-5} and weight decay of 1×10^{-6} and the same cosine scheduler with warmup.