R14922141 張宜安

Q1

Tokenizer

For tokenization, I used the tokenizer corresponding to each pre-trained model (e.g., WordPiece for BERT). The algorithm is not just a simple word split but a multi-step procedure to map natural language into a sequence of token IDs:

1. Preprocessing: The text is first normalized, such as lowercasing (for uncased models) and removing extra whitespace. For Chinese models, lowercasing is usually not applied since Chinese characters do not have case distinctions.
2. Initial segmentation: In Chinese, each character is directly treated as a basic unit.
3. Subword decomposition: Each word (or character in Chinese) is decomposed into subwords using the greedy longest-match-first algorithm against the model's vocabulary. This ensures that any word, even if unseen, can be represented in terms of known subwords.
4. Special tokens: Special markers like [CLS] at the beginning, [SEP] between sentences, and [PAD] for padding are added to form proper input sequences.
5. Numerical mapping: Each token is then mapped to its integer ID (*input_ids*). Additional attributes are also generated, such as *token_type_ids* (indicating sentence segments) and *attention_mask* (masking out padding positions).

Additional note on Chinese-specific models:
Different models also optimize their tokenizers. For example, Chinese-BERT-wwm improves upon *bert-base-chinese* by adopting a word-level segmentation strategy instead of character-level segmentation. Moreover, it applies the Whole Word Masking (WWM) technique during pre-training: when a word is masked, all

its constituent characters/subwords are masked together. This enhances the model's ability to learn coherent semantic representations compared to masking characters independently.

Answer Span

To handle answer spans, I needed to convert character-level annotations into token-level indices after tokenization, and then convert model predictions back into readable text:

1. Character-to-token mapping: During tokenization, the tokenizer provides an offset_mapping that stores the start and end character positions of each token in the original text. By aligning the gold answer span with this mapping, I can determine the start and end token indices. If the answer spans multiple tokens, the first token is marked as the start, and the last token as the end.
2. Handling missing spans: If the answer cannot be found in the current sliding window of the context, I assign the answer index to the [CLS] token, which denotes "no answer."
3. Post-prediction rules: After the model predicts start and end logits for each token:
    - I filter out invalid spans (e.g., end < start, or spans that exceed a maximum length).
    - I rank the candidate spans by their confidence score (start logit + end logit).
    - The best span is chosen, and its token indices are mapped back to the original character positions using offset_mapping, yielding the final answer text.

Q2

Model Description

I used the BERT-base Chinese model as the backbone for extractive question answering. I fine-tuned it on my QA dataset. The maximum input length was set to 512 tokens, which allows the model to handle relatively long contexts.

Performance (0.5%)

MC: accuracy : 95.4%

QA : 75.9%

Loss Function

The model was trained using cross-entropy loss, applied separately to the start position and end position of the answer span. This is the standard loss function for extractive QA tasks.

Optimization Algorithm and Hyperparameters

- Optimizer: AdamW (Adam with decoupled weight decay).
- Learning Rate: 3e-5 with a linear decay scheduler and warmup steps = 1500.
- Batch Size: Since my GPU memory was limited, I used per_device_train_batch_size = 1 with gradient_accumulation_steps = 8, which effectively makes the total batch size = 8.
- Epochs: MC : 1epo and same parameter as in HW1 ppt. QA: Trained for 5 epochs.

Q2-2

The new model

I used lert-base for MC.

I used hfl/chinese-pert-large as the backbone for extractive QA. This model follows the same Transformer encoder-only architecture as BERT but in a Large configuration (24 layers, 1024 hidden size, 16 attention heads). The tokenizer is still WordPiece without lowercasing, treating each Chinese character as a basic unit and splitting unseen words into subwords.

Training was conducted with the same settings as in my BERT baseline: maximum sequence length = 512, effective batch size = 8 (using per_device_train_batch_size=1 with gradient_accumulation_steps=8), learning rate = 3e-5, and 5 epochs.

The performance

For MC ,lert-base got 96.4% accuracy on 4 epochs.

On the validation set, pert-large achieved the following Exact Match:

Epoch	Exact Match (EM)
0	80.36
1	81.32
2	82.39
3	82.95
4	**83.15**

The difference between pre-trained LMs

- Architecture:
    - bert-base-chinese: 12 layers, hidden size 768, 12 heads.
    - chinese-pert-large: 24 layers, hidden size 1024, 16 heads (much larger parameter count, stronger representational power).
- PERT-Large requires more GPU memory and longer training time but delivers better accuracy.

The best result I submitted used PERT (chinese-lert-base / chinese-pert-large). Accuracy:96.4% on multiple choice and 83% exact match on QA.

79% on Kaggle public.

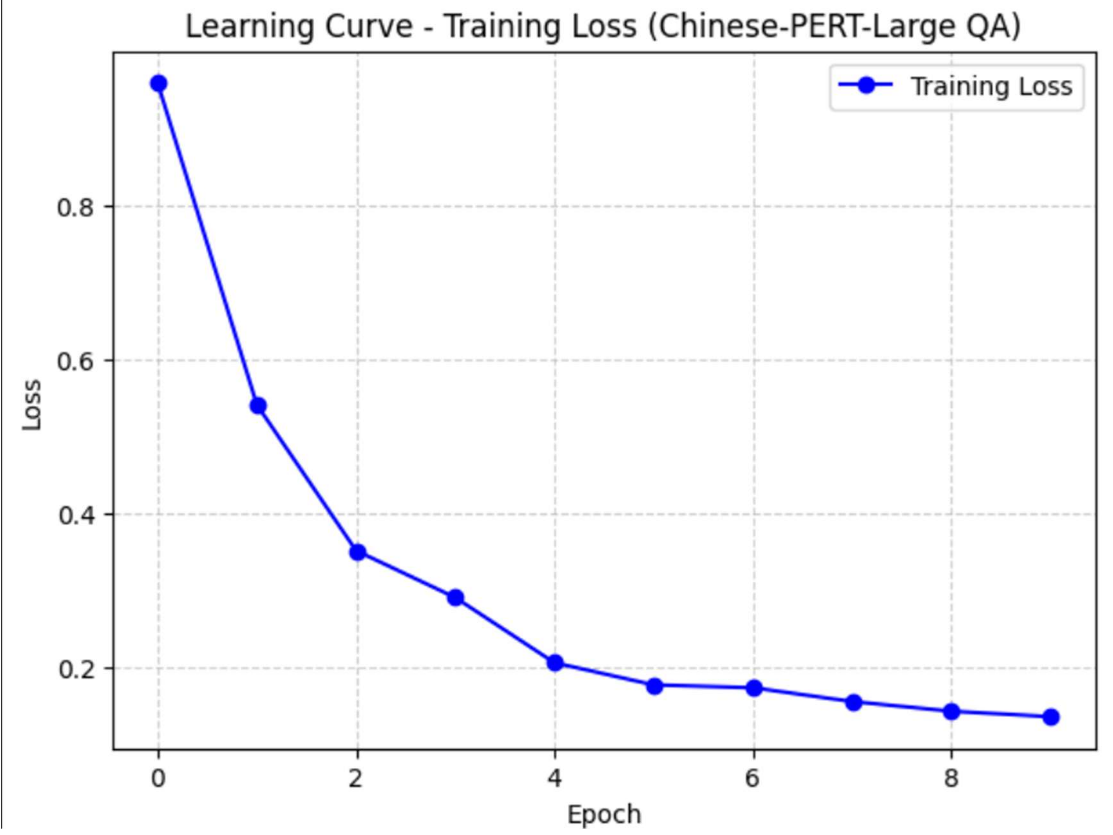
Other models for training QA :

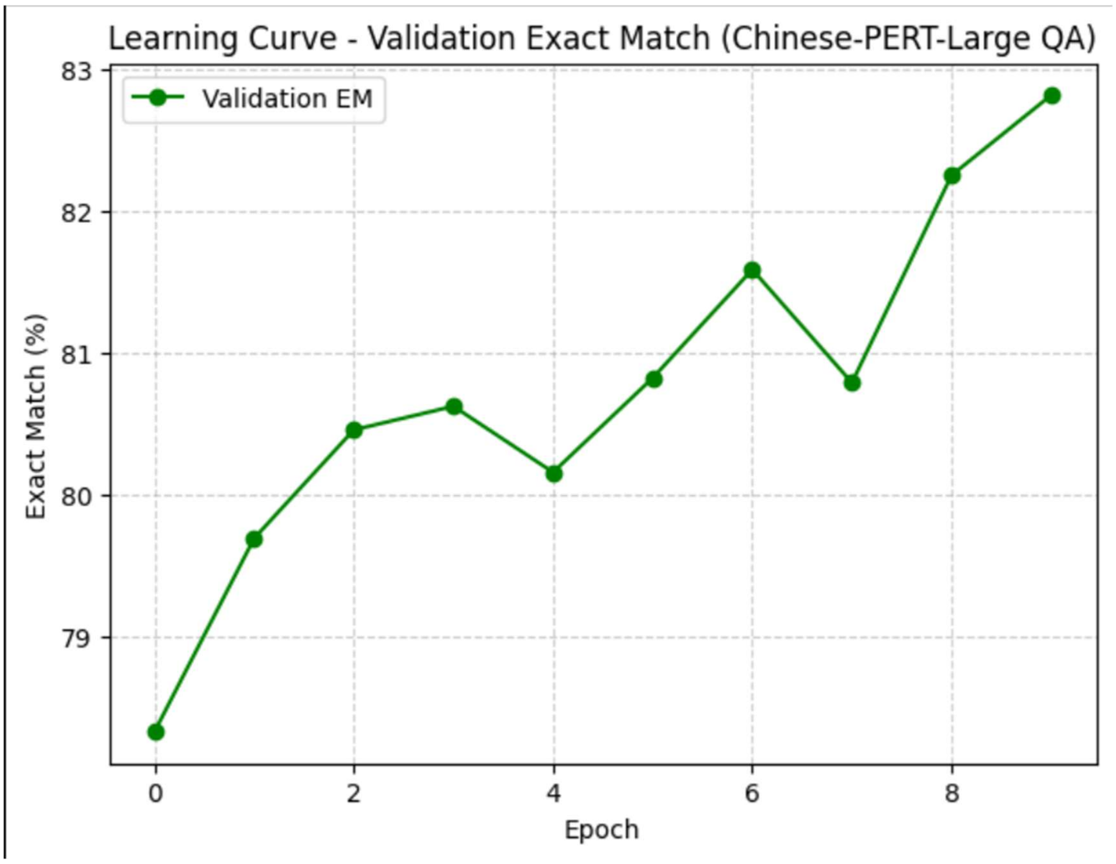
Parameters are the same as Chinese-pert-large

model	Exact Match (%)	Public Score
hfl/chinese-pert-large	83.15	0.79
bert-base-chinese	80.72%	0.76
hfl/chinese-lert-base	79.99	–
hfl/chinese-lert-large	82.02	0.77
roberta-wwm-ext	81.85	0.77
macbert-large	83.02	0.77

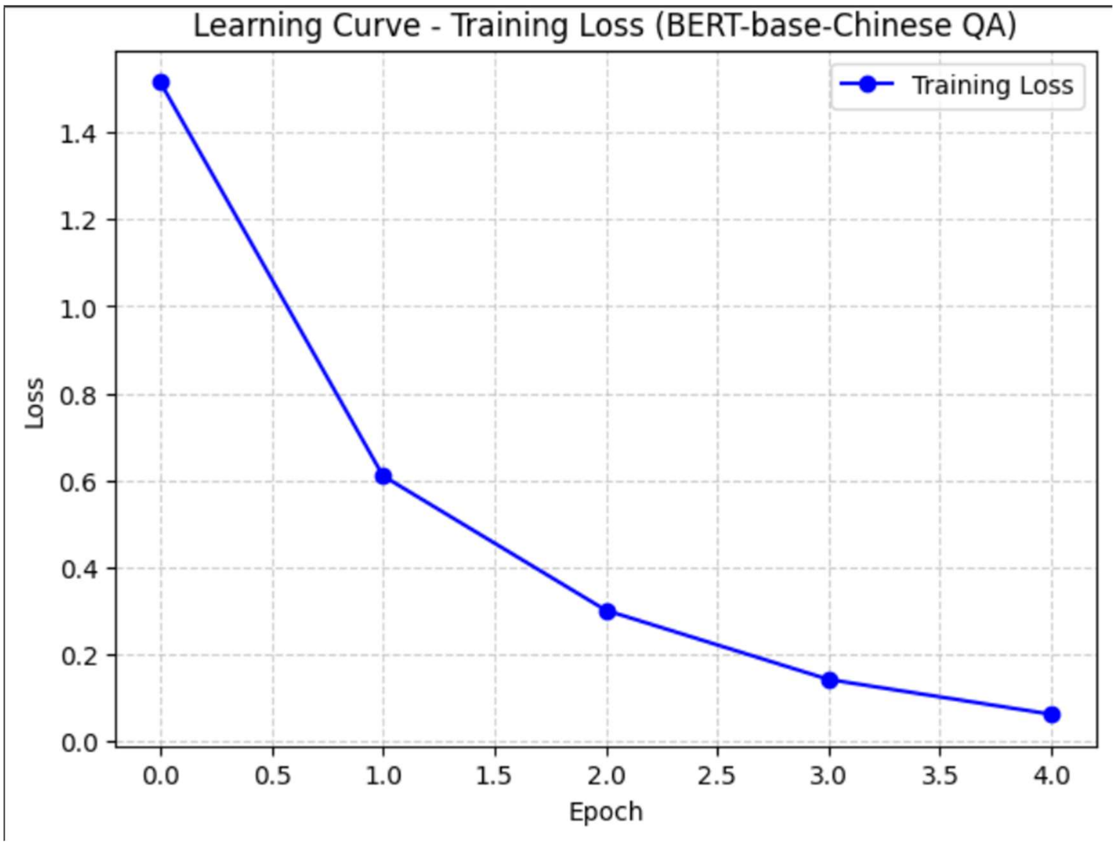
Q3 PLOT

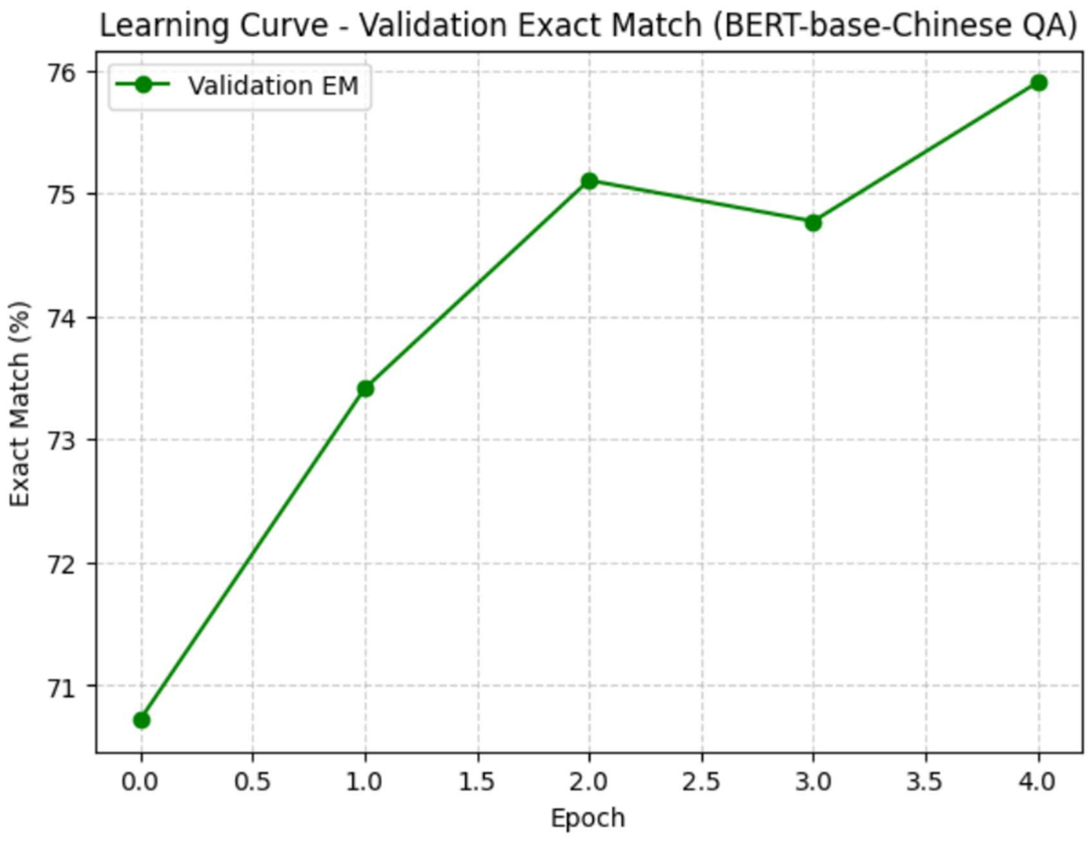
PERT(10 epo)


Learning Curve - Training Loss (Chinese-PERT-Large QA)
Training Loss
Loss
0.2
0.4
0.6
0.8
0
2
4
6
8
Epoch


Learning Curve - Validation Exact Match (Chinese-PERT-Large QA)
Validation EM
Exact Match (%)
79
80
81
82
83
0
2
4
6
8
Epoch

BERT(5 epo)


Learning Curve - Training Loss (BERT-base-Chinese QA)
Training Loss
Loss
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Learning Curve - Validation Exact Match (BERT-base-Chinese QA)
Validation EM
Exact Match (%)
Epoch
71
72
73
74
75
76
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Q4. Training a Transformer-based Model from Scratch (Paragraph Selection)

Model configuration and training procedure

- Architecture:
    - Hidden size = 384
    - Number of hidden layers = 6
    - Number of attention heads = 6
    - Max position embeddings = 512
- Tokenizer: bert-base-chinese WordPiece tokenizer.
- Training setup:
    - Batch size = 2
    - Learning rate = 5e-5 with cosine decay scheduler
    - Warmup steps = 200
    - Weight decay = 0.01
    - Number of training epochs = 1

The model was randomly initialized and trained using the same dataset and codebase as my pretrained experiments.

Performance

- From scratch (Mini-BERT, 6×384×6):
    - Training finished in ~918 seconds.
    - train_loss = 1.0111, val_loss = 0.9382, val_acc = 0.5194.
    - Best validation accuracy: 0.52.
- Pretrained (bert-base-chinese) under the same settings:

- Validation accuracy after 1 epoch: 0.954.

Comparison and analysis

1. Pretrained knowledge: The pretrained BERT already encodes rich Chinese linguistic features learned from massive corpora, while the from-scratch model must learn both syntax and semantics from limited task data.
2. Model capacity vs. data size: Even with reduced depth and hidden size, the scratch model is data-inefficient and requires many more epochs and larger training sets to approach competitive performance.

Q5

End-to-End Longformer QA Results

Model

I trained an end-to-end span selection model using LongformerForQuestionAnswering. (https://huggingface.co/ValkyriaLenneth/longformer_zh/blame/main/README.md) This architecture extends BERT with sparse attention, allowing it to process much longer input sequences than standard transformers. I set the maximum sequence length to 4096 with a document stride of 256, enabling the model to read long passages in overlapping windows. Global attention was applied to the question and the [CLS] token to ensure that essential tokens could attend across the entire context.

Performance

After 1 epoch, the model achieved the following metrics on the validation set:

- Exact Match (EM): 55.1%
- F1 score: 71.0%
- Validation loss: 1.62

- Evaluation runtime: ~12,161 seconds (~3.4 hours)
- Throughput: ~0.082 samples per second

These results demonstrate that the model can successfully learn to identify answer spans in long contexts, though training is computationally expensive due to the extended sequence length.

Loss function

The model was optimized using the cross-entropy loss on both start and end positions of the answer span. The final training objective is the sum of these two losses, which is the standard approach for extractive QA models.

Optimization algorithm

- Optimizer: AdamW (with weight decay)
- Learning rate: 5e-5 with linear decay and warmup ratio of 0.1
- Batch size: 2 (with no gradient accumulation)
- Training epochs: 2 were planned, results here are from epoch 1

Comparison :

Signal-to-Noise Ratio:
Feeding very long sequences injects a large amount of irrelevant tokens. The start/end span softmax is forced to compete against many more negatives, which dilutes supervision and makes learning harder. In the two-stage pipeline, the MC step first filters out unrelated paragraphs, so the QA model operates on a short, high-signal context, yielding more stable and accurate span predictions.

Sparse Attention:
Longformer's sparse attention expands the visible context but restricts full pairwise interactions. Merely assigning global attention to [CLS] and the question often isn't enough to reliably bridge all crucial cues to the answer span. By contrast, the two-stage approach typically uses a dense self-attention encoder over a single paragraph, making fine-grained token interactions and exact span localization easier.