I am forced to submit this report in English due to my limited skills in Chinese. I apologize about this. I hope that it will not be a problem and I will be grateful if I can get feedback about the assignment in English, too.

Data processing

Tokenizer

As can be seen in the `tokenizer_config` files for the models I uploaded to Google Drive, in both cases the tokenizer class used - by default - was Bert Tokenizer. Both models loaded their respective pretrained tokenizers. In the technical report for the hfl/chinese-roberta-wwm-ext model, which I eventually used for question answering, the authors note that during pretraining, rather than splitting words with the WordPiece algorithm as in the original BERT, Chinese Word Segmentation is used to split the text. However, it is stated that WordPiece is still used for prediction. In general terms, it works as follows: during pretraining, it starts by splitting the whole text into as small units as possible (individual letters for English, and characters for Chinese). Then it iteratively merges some two subsequent units into a new unit. For each merge, the pair chosen to be merged is the one that occurs the most often in the text. Thus, the algorithm learns a number of merge rules. The desired number of merges is a hyperparameter. When tasked with tokenizing a new text, it again first splits it into individual symbols, and then applies the merge rules it learned during training.

Answer span

This part was done automatically, since I used the prediction scripts from the Transformers library with quite minor modifications. The postprocessing function `postprocess_qa_predictions` can be found in `examples/pytorch/question-answering/utils_qa.py` in the huggingface repository. We can observe (line 114) that the function uses an offset mapping, which I understand was computed earlier during tokenization, to convert start and end positions from after tokenization to positions from before tokenization. The function receives a number of predictions that the model made for start and end positions of the answer. At line 156, each of those predictions is assigned a score which is simply a sum of its start and end logits. Then, the predictions with the best scores for each sample are saved in the `predict_predictions.json` file.

Final models

All in all, I have tried two pretrained models for context selection, and three for question answering. I elaborate on this in the Other tested models section. The final models I decided on were: a fine-tuned hfl/chinese-roberta-wwm-ext model for context selection and a fine-tuned hfl/chinese-roberta-wwm-ext-large model for question answering.

Configuration and hyperparameters

I mostly used the simple baseline parameters provided in the task, as well as default parameters of the fine-tuned models. This resulted in the following parameters:

- Optimization algorithm: Adam for both.
- Learning rate: $3e-5$ for both.
- Batch size: 2 for both, I did not use gradient accumulation.
- Loss function: for QA, to quote the original BERT paper, the training objective was the 'sum of the log-likelihoods of the correct start and end positions'. I assume the negative of this was taken as loss function, so that the minimization, rather than maximization of the loss function would become the objective. For CS, as can be seen in the definition of the `BertForMultipleChoice` model, the default loss is Cross Entropy.
- Activation function: GELU for both.

- Training epochs: 1 epoch for both.
- Number of attention heads: 12 for CS, 16 for QA.
- Hidden layers: 12 for CS, 24 for QA.
- Hidden size: 768 for CS, 1024 for QA.

Given that a few parameters determining the model's size are bigger for the QA model, we can expect a larger training time. Indeed it was 12441 seconds versus 8740 seconds. It should be noted, however, that when the same bert-base-chinese model was applied for both tasks, the training time of the CS model was about two times higher despite the same setup - I included more details in the Other tested models section.

Architecture, data flow

The architecture of both the hfl/chinese-roberta-wwm-ext-large question answeing model and the hfl/chinese-roberta-wwm-ext context selection model is not different from the original BERT architecture, which in turn, in the words of its authors, is 'almost identical' to the original architecture proposed by Vaswani et al. This means that in both cases, a sequence of input symbols is processed into a sequence of output symbols. I will first focus on this processing, whose mechanism is the same for both models, and then explain the differences in the interpretation of the output symbols.

The processing is divided into two parts: the encoder part and the decoder part, followed by a final linear transformation. To quote the Attention is All You Need paper: 'the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given z, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time.'

The encoder is composed of N stacked identical layers, with the output of every layer being the input of the next one. The output for each layer is computed as

$$Layer(x) = LayerNorm(y + FF(y))$$

where

$$y = LayerNorm(x + MSA(x))$$

where:

- x is the output of the previous layer, or - in the first layer - the whole input sequence representation. The sequence representation is obtained by taking the tokenized input sequence, converting its tokens to embeddings, and adding positional encodings - vectors defined so that the i-th feature in the embedding of position p is $\sin(p/10000^{i/d})$ for even i and $\cos(p/10000^{(i-1)/d})$ for odd i. d denotes the embedding dimensionality.
- LayerNorm is the layer normalization function as described here,
- FF is a function implemented by a feed-forward network, performing two linear transformations with a ReLU acivation in between: $FF(x) = W_2 \cdot ReLU(W_1 x + b_1) + b_2$
- $MSA(x)$ is the function implemented by multi-headed self-attention over x.

The last one of those calls for a more detailed explanation. The simple attention function is defined as

$$A(k, q, v) = softmax(\frac{qk^T}{\sqrt{d}})v$$

where d is the dimensionality of k, and the variables k, q, v are by convention called keys, queries and values. For multi-headed self-attention over x, we first project it into key, query and value spaces using matrices learned for this task. This is done H times, and we call H the number of attention heads. Each head i has its own projection matrices K_i, Q_i, V_i. Then each head performs attention over the projections, and the final result is the concatenation of attention results for all the heads, with an additional linear transformation applied. In other words:

$$MSA(x) = W \cdot Concat(head_1, ...head_H)$$

where W is the matrix of the final transformation, and

$$head_i = A(K_i x, Q_i x, V_i x)$$

The decoder works similarly, but with an additional sublayer between the attention sublayer and the feed-forward sublayer, which, to quote the paper again, 'performs multi-head attention over the output of the encoder stack'. In addition, The attention mechanism is modified so that positions cannot attend to later positions in the sequence.

Finally, for the question answering task, additional start and end vectors S, E are introduced. The logit corresponding to the probability of the word i being the start of the answer span is computed as a dot product between S and the i-th output vector of the decoder, and analogously for the end. For context selection, a special vector is introduced, whose dot product with the representation of BERT's `[CLS]` token of a given context gives us the logit for the context being relevant.

Evaluation

The final Kaggle score achieved by the models was 0.7839, higher (but not considerably) than the strong baseline. The final evaluation accuracy for the CS model itself was 0.9608. Its final loss was 0.206, while the loss of the QA model was 1.02. The F1 metric was 82.5.

In addition to these statistics, learning curves for the QA model can be seen on the last page of the report.

Other tested models

Pretrained

Originally, for both context selection and question answering, I fine-tuned pretrained bert-base-chinese models, without changing the default parameters. Thus, in both cases the most relevant parameters were: 12 attention heads, 12 hidden layers, hidden size 768, learning rate $3e-5$. Train losses for CS and QA were 0.22 and 0.98, respectively. The F1 metric calculated for QA was 76.5. This tandem of models achieved a kaggle score of 0.72694 - below even the simple baseline. The train runtimes were 8629 and 3596 seconds, respectively - I am not entirely sure why the QA model required longer to train, although I intuitively understand that the task of selecting an answer span with multiple possibilities is more complicated than merely selecting one of four given choices. In addition, it may be related to the fact that a number of most likely answer predictions is generated for each question, rather than a single answer.

I decided to improve the QA model by substituting the hfl/chinese-roberta-wwm-ext model for bert-base-chinese, without changing any other configurations. This gave an F1 metric of 80.6, and increased Kaggle score to 0.78028. However, here I encountered a problem, mentioned in the README.md file too - this result was not reproducible if prediction was run on my laptop or a CSIE machine, rather than on Colab. In that case, the Kaggle score was lower and barely above the strong baseline. I tried to fix this by substituting hfl/chinese-roberta-wwm-ext for bert-base-chinese in the CS part, too, but this increased the Kaggle score only to 0.78209 (while predicting on Colab - I decided not to try this at home, because this would have taken a few hours, with a slim chance of obtaining a score safely above the baseline). Eventually, then, I trained a larger model for QA, based on hfl/chinese-roberta-wwm-ext-large - as explained in more detail in the Other tested models section.

Not pretrained

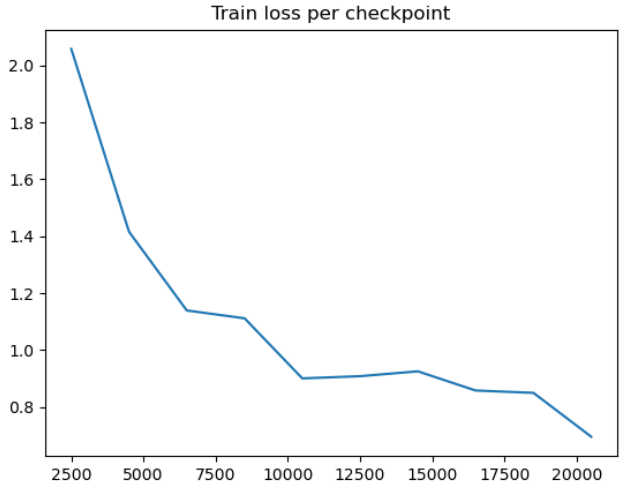
I trained a question answering model from scratch by running, on Google colab, a script that I copied into `misc_code/train_qa_from_scratch.ipynb`, which in turn calls the `train_qa_from_scratch.py` program. The program is based on the same training script I used for fine-tuning. However, beginning from the line 323, I create a model configuration with three paremeters (hidden size, number of

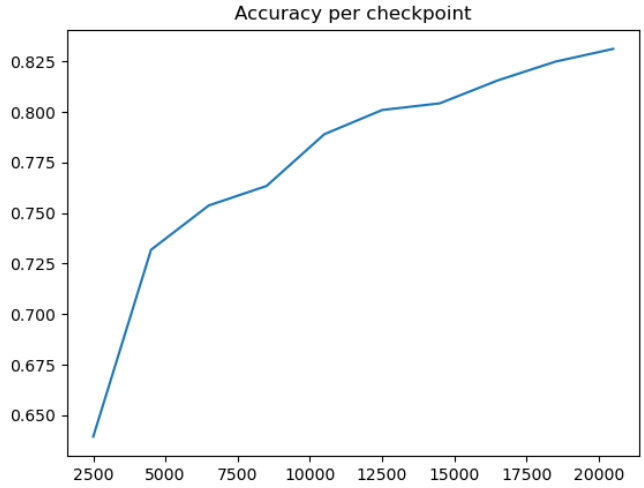
layers, number of attention heads) two times smaller than in the case of the bert-base-chinese model, so respectively - 384, 6 and 6. Then I create the model using the `from_config` function, rather than `from_pretrained`. Other than that, I used the same parameters as for the original fine-tuning of bert-base-chinese, in particular the learning rate of $3e-5$. I intended to train the model for 30 epochs, but I was cut off the GPU at Colab after about 22.3 epochs. Unfortunately, I think I lost the precise training time, but I recall it was about 6 hours. In the final iteration, loss was equal to 1.7176.

The Kaggle score obtained by this model, run on a test dataset generated with the help of the final context selection model, was very low - just 0.06419. I suppose this is both the fault of the reduced model size and of the short training time. That said, I had hoped the score would be considerably higher, especially since it was higher for the original fine-tuned bert-base-chinese model, whose training loss was, as mentioned, 0.98 - not drastically lower than 1.7176.

Learning curves

To plot the learning curves, I collected the loss and validation accuracy values for respective checkpoints manually. The checkpoint numbers (corresponding to the numbers of batches preprocessed at each checkpoint) were distributed every 2000 from 500 to 18500, making 10 checkpoints in total. Losses were obtained from the `trainer_state.json` files in the checkpoint directories generated during training. Accuracy values were calculated by manually running the `misc_code/eval.ipynb` file to create predictions for the evaluation dataset for each checkpoint, and then manually checking the number of correct answers. The program in `misc_code/plot.py` calculated accuracies and generated the plots, which can be seen below.


Train loss per checkpoint
2.0
1.8
1.6
1.4
1.2
1.0
0.8
2500
5000
7500
10000
12500
15000
17500
20000


Accuracy per checkpoint
0.825
0.800
0.775
0.750
0.725
0.700
0.675
0.650
2500
5000
7500
10000
12500
15000
17500
20000