NTU ADL 2023 Fall - HW1 Report

written Chih Han, Yang - B10902069

Q1 Data Processing

1. Tokenizer: The tokenization algorithm that I use is the [“Word Piece Tokenization”]. The procedure can be elaborated in a few steps:
    1. Define that the initial vocabulary for the tokenization is:

        ```
        Vocabulary: ["b", "h", "p", "##g", "##n", "#
        ```

    2. Then, for every words in the context, we split it repeatedly by the longest subwords starting from the beginning of the word until all the split words are in the vocabulary.
    3. For example, we want to split the word “bugs”. The longest subword satrting from the beginning would be “b”. Thus, we split “bugs” into

        ```
        ["b", "##ugs"]
        ```

        Next, for “##ugs”, the longest subword starting from the beginning would be “##u”. We consequently split “##ugs” into

        ```
        ["##u","##gs"]
        ```

        Since “##ugs” is in the vocabulary, our proced for “bugs” stops here. Eventually, we’ll have tokenized data:

        ```
        ["b", "##u", "##gs"]
        ```


2. Answer Span:
    - Here’s how I convert the answer span start/end position on characters to position on tokens after BERT tokenization. First, use the tokenizer to tokenize a given paragraph. Then, the tokenized paragraph would have a few padding tokens in the beginning and the end. We first exclude those paddings. T
      The features are labeled with the start position and the text of the answer. Also, for the tokenized

paragraph, each token is mapped with a position where the token should be in the original paragraph. Thus, we use all the tokens' mapped position to locate the first token of the answer, and the last token of the answer.
For example, we have a paragraph

```
It's a happy dog.
```

And the answer is

```
happy dog
```

Assume that the tokenized paragraph is

```
{"token":position_in_the_original_paragraph}
= {"it":0 , "##'s":0, "a":1, "happy":2, "d":
```

And the answer start and end position is

```
{"word":position}
= {"happy":2, "dog":3}
```

So we can locate the first token of the answer and the last token of the answer should be:

```
{"token":position_in_the_origianl_paragraph:
= {"happy":2:3, "##og":3:5}
```

Thus, we can have that the answer span start/end positions would be 3/5 in the tokenized paragraph.

- After the model predicts the probability of answer span start/end position, the code took a few steps to decide the final start/end positions:
  - Extract start and end logits from the model's prediction
  - Generate temporary predictions by considering different start and end position based on the logits
  - Filter out invalid answers and selects the top few predictions
  - Then calculate the probabilities for each predictions using softmax
  - Select the prediction with the highest probability
  - Finally, we'll have the selected prediction providing a start position and an end position.

Q2 Modeling with BERTs and their variants

1. Describe: (I used two different pre-trained LMs for the two different tasks)
    - Mulitple Choice (Paragraph Selection):
        - Model: `bert-base-chinese`
        - Performance of the model: The evalutation accuracy on the training dataset: `0.9501495513459621`
        - The loss function: [Huggingface] [Github Doc] [Github Code] [Cross-Entropy]. According to the hugginface documentation, the transformers models all have a default task-relevant loss function. And on the official github website, it uses `modeling_tf_utils.TFMultipleChoiceLoss` function, which is implemented by `tensorflow.keras.losses.SparseCategoricalCrossentropy` . It computes the crossentropy loss between the labels and predictions.
        - Learning rate: 3e-5, with linear learning rate scheduling.
        - Batch size: effective batch size = 8 = 4*2 = per_device_batch_size * gradient_accumulation_steps
    - Extractive (Extract the answer):
        - Model: `hfl/chinese-roberta-wwm-ext-large`
        - Performance of the model: Exact match percentage on the validation dataset: `83.05084745762711`
        - The loss function:[Huggingface] [Github Doc] [Github Code] [Cross-Entropy]: According the transformer's official github website, it uses `modeling_tf_utils.TFQuestionAnsweringLoss` function, which is implemented by `tensorflow.keras.losses.SparseCategoricalCrossentropy` . It returns the mean of the cross entropy for the start positions, and the cross entropy for the end positions.
        - The optimization algorithm: [AdamW optimizer]
        - Learning rate: 3e-5, with linear learning rate scheduling.

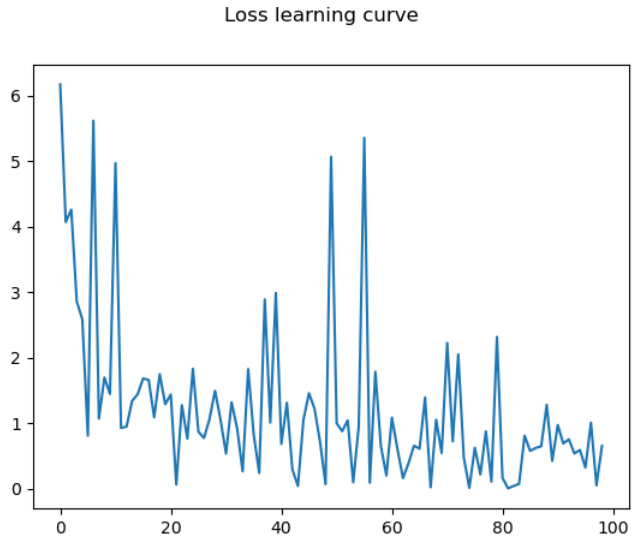
- Batch size: effective batch size = 2 = 2*1 = per_device_batch_size * gradient_accumulation_steps

2. For the extractive model, other than the `hfl/chinese-roberta-wwm-ext-large` model, I've tried fine-tuning `bert-base-chinese` as well. The performance wasn't the best, with exact match percentage on the validation dataset only about 75%. The differences between them is the masking strategy. In the original paper for `hfl/chinese-roberta-wwm-ext-large`, it introduced a masking strategy called Mac (MLM as correction). MLM stands for Masked language model, which should predict the masked token as it's output. This is what the original BERT does. However, the paper points out an issue that the artificial token in pretraining stage, such as `[MASK]`, would never appears in real fine-tuning tasks. What Mac does is that, given a sequence of input with a artificial wrong word token, the model should predict the correct token as output.
   Link to the Paper: (**https://arxiv.org/pdf/1906.08101.pdf** (https://arxiv.org/pdf/1906.08101.pdf))

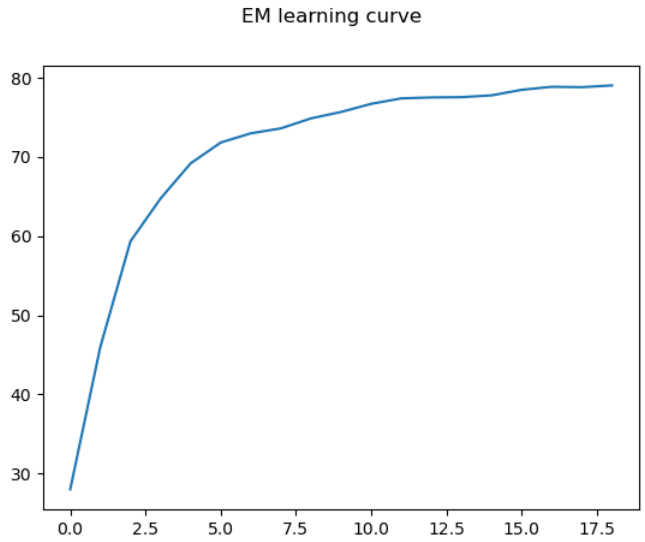
Q3 Curves

(On the validation dataset)

1. Learning curve of the loss, sampled every 20 batches



2. Learning curve of the EM, sampled every 100 batches



Q4 Pre-trained vs Not Pre-trained

- Here's the configuration of the **not pre-trained model**:


```
Model config BertConfig {
  "_name_or_path": "bert-base-chinese",
  "architectures": [
    "BertForMaskedLM"
  ],
  "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",
  "transformers_version": "4.33.3",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```

The architecture is BertForMaskedLM, which is the basic model architecture of BERT. However, for the weights are not pre-trained, the result could be significantly different from the pre-trained one.

- Here's the hyperparameters for training this **not pre-trained model**:

```
max_seq_length = 512
model_name_or_path = bert-base-chinese
learning_rate = 3e-5
num_train_epochs = 8
per_device_eval_batch_size = 4
per_device_train_batch_size = 4
gradient_accumulation_steps = 2
seed = 4125252
```

- Here's the performance of this model vs the pre-trained bert model:
  - pre-trained bert's accuracy: `0.9501495513459621`
  - not pre-trained bert's accuracy: `0.5370555001661682`

  Under the same configuration (e.g. model architecture, hidden layer, maximum sequence length, learning rate, training epochs, batch size, or even seed), a pre-trained model out-performs the not pre-trained model. On account of a pre-trained model, the actual training dataset is much bigger and extensive, for it has been pre-trained with those datas. On the other hand, the not pre-trained model only has the fine-tuning dataset for the pre-trained model as it's entire training dataset. From such difference on the size of the dataset, a pre-trained then fine-tuned model can definitely give a much more accurate, sophisticated, and general performance.

Q5 Bonus

1. Strategy: My strategy to merge the multiple choice model and the extractive model is actually pretty simple. Imagine that Alice is doing such task, which requires selecting the correct paragraph and highlighting the correct span of the answer. What Alice did is simply read all the four paragraphs and the question at once, then highlight the answer span.

Building on that analogy, **my strategy is to concatenate all the paragraphs into one long paragraph, and use the extractive model to extract the correct answer span.** Note that the label's answer start position should be modified as well.

2. Model: `hfl/chinese-roberta-wwm-ext-large`
3. Performance of the model: Exact match percentage on the validation dataset: `18.876703223662346`
4. Loss function: `tensorflow.keras.losses.SparseCategoricalCrossentropy`
5. The optimization algorithm: [AdamW optimizer]
6. Learning rate: 3e-5, with linear learning rate scheduling.
7. Batch size: effective batch size = 8 = 2*4 = per_device_batch_size * gradient_accumulation_steps
8. The performance was not so good as the two models. My guess is that, I only trained for one epoch and with batch size 8. Meanwhile, the original extractive model was trained for 3 epochs, and batch size 2. Which means this model was trained with 1/12 optimization steps. The lack of optimization steps maybe the cause of such performance. Lacking of training resources, unfortunately, I can't train it any longer. I believe it can obtain better result if it has been trained longer.