HW2

Q1:Data processing

Tokenizer

The tokenizer I used is bert-base-chinese.

It is an implementation for WordPiece.

Let me describe this algorithm briefly:

1. We should split all sentences into many minimal tokens
2. Learning the rules for merge token
    a. use the formula below to choose token pair

    `score=(freq_of_pair)/(freq_of_first_element×freq_of_second_element)`

    b. we can merge 2 tokens to get new token
    c. loop to get more merge rule until reach the desired vocabulary size
3. Use the rules to tokenize
    a. input sentence
    b. split to smallest tokens
    c. use best rule to merge tokens (Attempt to match tokens merged by the longest tokens)
4. There should be some special markup meaning, like split, start, end, …

Answer Span

a. The dataset will give the target string and start position, so we can easily get the end position of the target string, after that, we should check all the tokens and get the result that belongs to the string. Finally, we get the tokenized location

b. We violently try all possible consecutive tokens, use the offset between the previously saved token and the actual content to get the corresponding substring, and finally select the substring with the highest score

Q2: Modeling with BERTs and their variants

Describe

a. my model configure

   their are two NN model in this model

   first one is MC


```
{
  "_name_or_path": "ckiplab/albert-tiny-chinese",
  "architectures": [
    "AlbertForMultipleChoice"
  ],
  "attention_probs_dropout_prob": 0.0,
  "bos_token_id": 101,
  "classifier_dropout_prob": 0.1,
  "down_scale_factor": 1,
  "embedding_size": 128,
  "eos_token_id": 102,
  "gap_size": 0,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.0,
  "hidden_size": 312,
  "initializer_range": 0.02,
  "inner_group_num": 1,
  "intermediate_size": 1248,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "albert",
  "net_structure_type": 0,
  "num_attention_heads": 12,
  "num_hidden_groups": 1,
  "num_hidden_layers": 4,
  "num_memory_blocks": 0,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "tokenizer_class": "BertTokenizerFast",
  "torch_dtype": "float32",
  "transformers_version": "4.23.1",
  "type_vocab_size": 2,
  "vocab_size": 21128
}
```


   second one is QA

```
{
  "_name_or_path": "hfl/chinese-roberta-wwm-ext",
  "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": 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": "4.23.1",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 21128
}
```


b. performance of your model: **0.77667**

c. the loss function you used:

   I didn't overwrite the loss function of the model, and I do not find the source code. But I traced the source code of similar architecture and I think each architecture uses CrossEntropyLoss as the loss function

d. my optimizer: AdamW

   learning rate: 5e-5

   batch size: 16

Try another type of pretrained model and describe

a. my model configure

   their are two NN model in this model

   because “ckiplab/albert-tiny-chinese” have small memory space and faster speed,

   so I just change QA NN to “ckiplab/albert-tiny-chinese”

   below is config


```
{
  "_name_or_path": "ckiplab/albert-tiny-chinese",
  "architectures": [
    "AlbertForQuestionAnswering"
  ],
  "attention_probs_dropout_prob": 0.0,
  "bos_token_id": 101,
  "classifier_dropout_prob": 0.1,
  "down_scale_factor": 1,
  "embedding_size": 128,
  "eos_token_id": 102,
  "gap_size": 0,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.0,
  "hidden_size": 312,
  "initializer_range": 0.02,
  "inner_group_num": 1,
  "intermediate_size": 1248,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "albert",
  "net_structure_type": 0,
  "num_attention_heads": 12,
  "num_hidden_groups": 1,
  "num_hidden_layers": 4,
  "num_memory_blocks": 0,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "tokenizer_class": "BertTokenizerFast",
  "torch_dtype": "float32",
  "transformers_version": "4.23.1",
  "type_vocab_size": 2,
  "vocab_size": 21128
}
```


b. performance of your model: **0.53**

   I guess it's because the QA task is more complex, so the small model can't do it

c. the difference between pretrained model (architecture, pretraining loss, etc.)

   I will explain the difference between each model by talking about their pros and cons

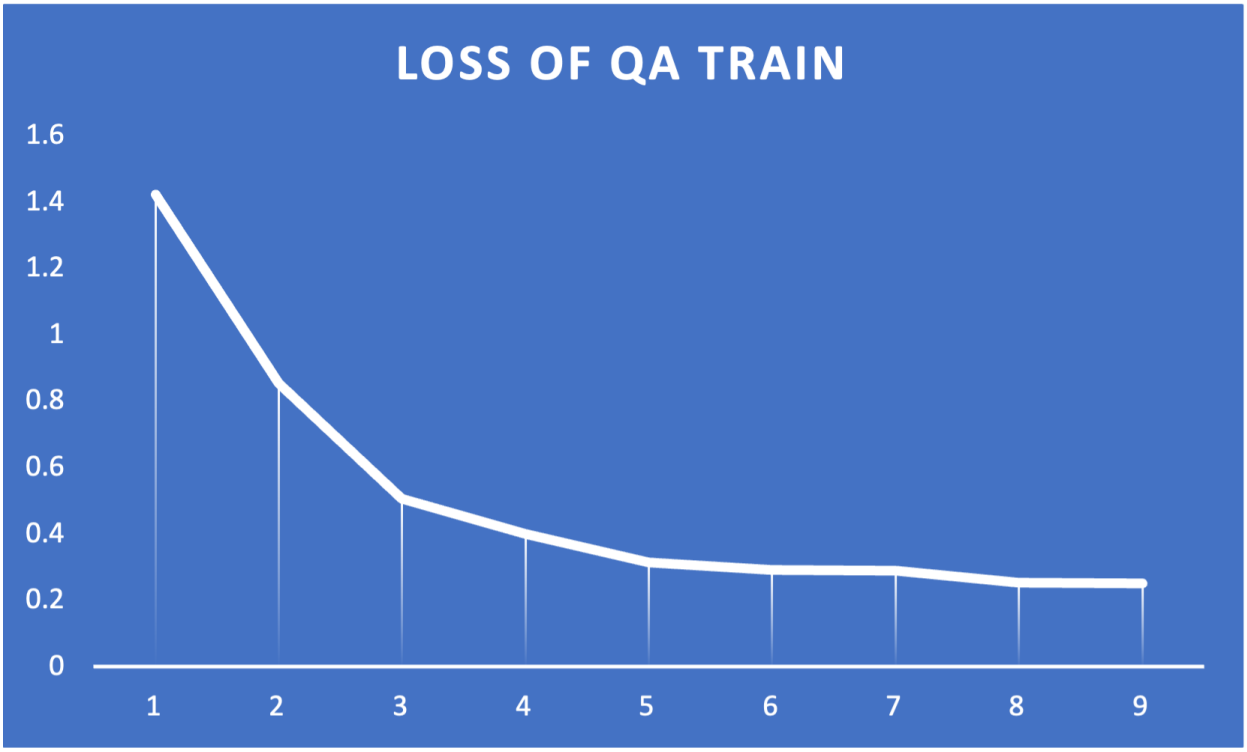
   "ckiplab/albert-tiny-chinese": small memory space, fast training/prediction, but not very good for complex tasks

   "hfl/chinese-roberta-wwm-ext": large memory space, slow training/prediction, but good for complex tasks

d. I also tried using "hfl/chinese-roberta-wwm-ext" in the model per NN, but I found it was bigger and slower than my model, but didn't get a higher score than my model

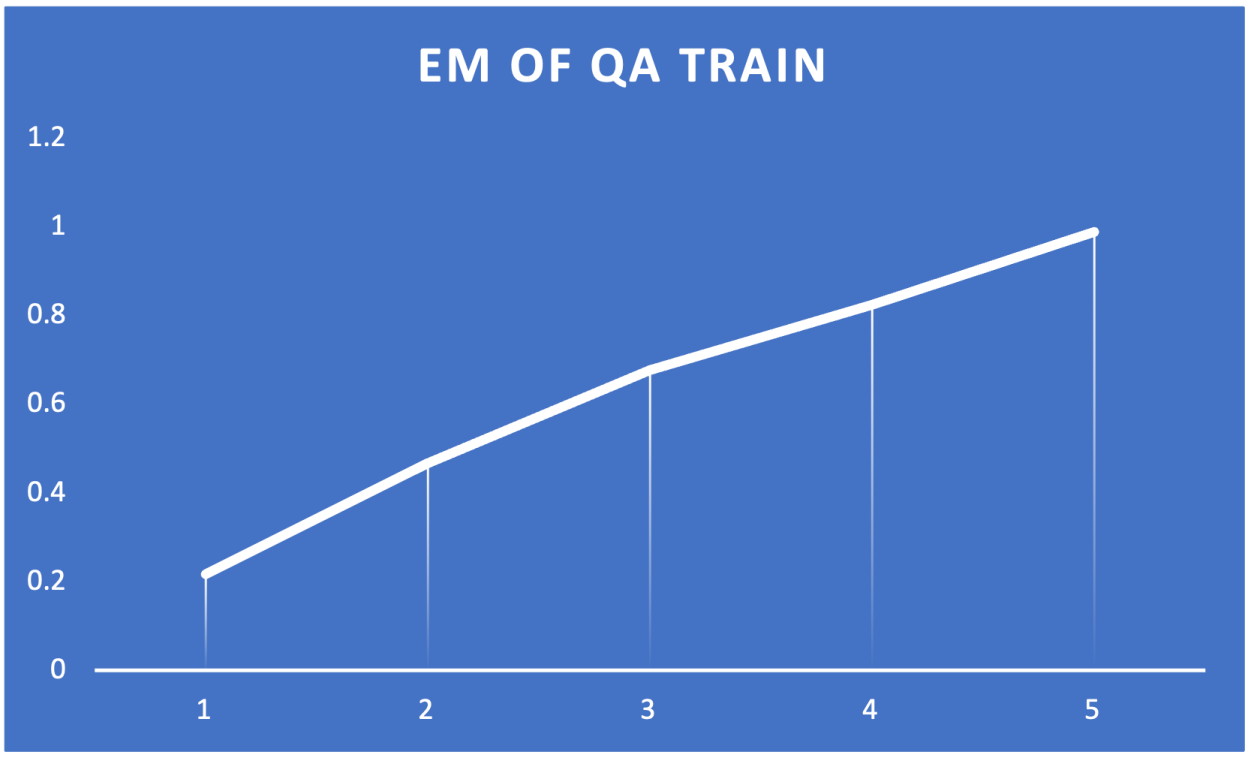
Q3: Curves

loss function of QA



x-axis mean 500 batch (batch size = 16)

EM of QA



x-axis 1000 batch (batch size = 16)

Q4: Pretrained vs Not Pretrained

I trained an unpretrained MC model

the config of it is below

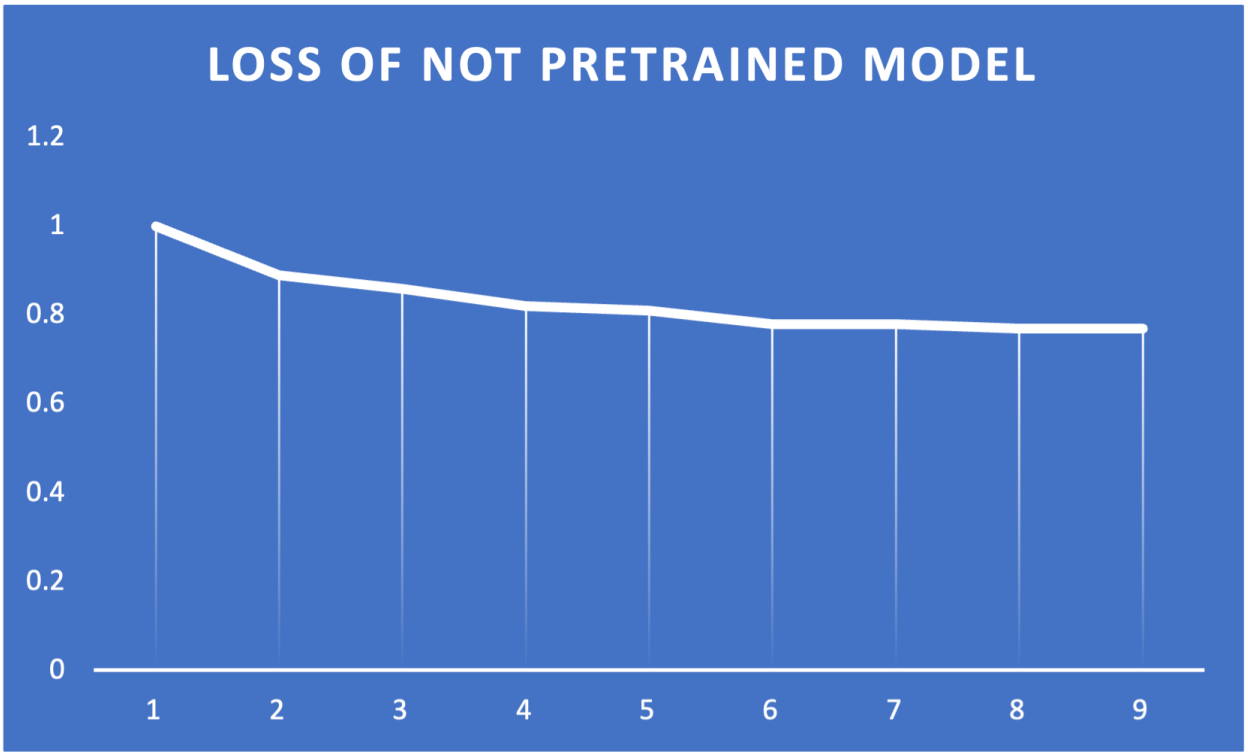

```
{
  "_name_or_path": "ckiplab/albert-tiny-chinese",
  "architectures": [
    "AlbertForMultipleChoice"
  ],
  "attention_probs_dropout_prob": 0.0,
  "bos_token_id": 101,
  "classifier_dropout_prob": 0.1,
  "down_scale_factor": 1,
  "embedding_size": 128,
  "eos_token_id": 102,
  "gap_size": 0,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.0,
  "hidden_size": 312,
  "initializer_range": 0.02,
  "inner_group_num": 1,
  "intermediate_size": 1248,
```

```
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "albert",
  "net_structure_type": 0,
  "num_attention_heads": 12,
  "num_hidden_groups": 1,
  "num_hidden_layers": 4,
  "num_memory_blocks": 0,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "tokenizer_class": "BertTokenizerFast",
  "torch_dtype": "float32",
  "transformers_version": "4.23.1",
  "type_vocab_size": 2,
  "vocab_size": 21128
}
```


I just cancel the pre-weight in origin model MC

the performance of this model is really bad

I draw the curve for it below

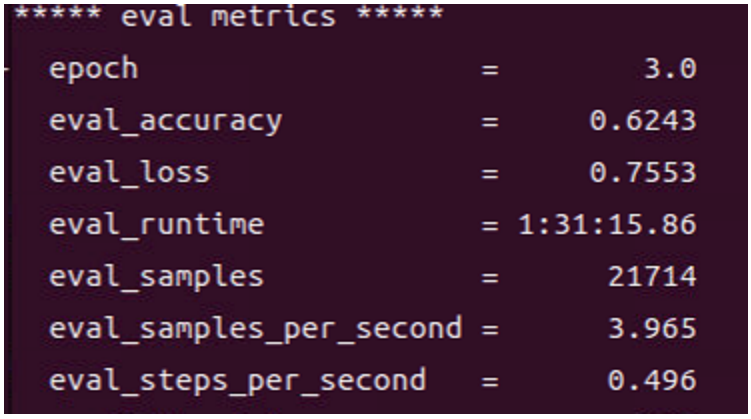


x-axis mean 500 batch (batch = 8)

We found that there is insufficient data to train such a model, and the loss is stable at 0.77 (trained model can be 0.2).

Therefore, the final accuracy is also small.

Here is the eval_metrics



Also, I have tried some other models to solve this problem, but because this model has the fewest parameters, it has the best performance

Q5: Bonus: HW1 with BERTs

you can check `/bonus` for more detail

a. your model
    a. intent: bert tiny
    b. slot: bert-base-uncased
b. performance of your model
    a. intent: 0.771

```
***** Running training *****
  Num examples = 15000
  Num Epochs = 8
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distribute
  Gradient Accumulation steps = 1
  Total optimization steps = 7504
/usr/local/lib/python3.7/dist-packages/ipykernel_
  if __name__ == '__main__':
```


Step	Training Loss	Validation Loss	Accuracy
50	No log	5.022862	0.006000
100	No log	5.021906	0.007333
150	No log	5.020112	0.008000
200	No log	5.017566	0.011333
250	No log	5.014102	0.011667
300	No log	5.009676	0.013667
350	No log	5.003967	0.013000
400	No log	4.996514	0.018667
450	No log	4.986050	0.018000
500	5.014500	4.976303	0.020000
550	5.014500	4.959329	0.027000
600	5.014500	4.938393	0.031667
650	5.014500	4.926051	0.024333

b. slot: 0.976

```
***** eval metrics *****
  epoch                   =        3.0
  eval_accuracy           =      0.976
  eval_f1                 =     0.8485
  eval_loss               =     0.0836
  eval_precision          =     0.8455
  eval_recall             =     0.8515
  eval_runtime            = 0:00:08.91
  eval_samples            =       1000
  eval_samples_per_second =    112.147
  eval_steps_per_second   =     14.018
```

c. the loss function you used

   a. intent

      by source code

      ⇒ CrossEntrophyLoss

   b. slot

      I didn't overwrite the loss function of the model, and I do not find the source code. But I traced the source code of similar architecture and I think this architecture uses CrossEntropyLoss

d. The optimization algorithm (e.g. Adam), learning rate and batch size.

   a. intent

- optimization algorithm: AdamW
- learning rate: 3e-5
- batch size: 16

b. slot

- optimization algorithm: AdamW
- learning rate: 3e-5
- batch size: 16