# 案例十五 基于BERT模型的机器阅读理解

## 案例概述

机器阅读理解是自然语言处理中的一个重要的任务，最常见的有单篇章的抽取式阅读理解。机器阅读理解的应用范围很广，比如客服机器人，通过文字或者语音与用户进行沟通交流，然后获取相关的信息并提供准确可靠的回答。搜索引擎中精确返回用户所给定问题的答案。在医疗领域中自动阅读病人的资料来找到相应的病因。在教育领域中，利用阅读理解模型自动为学生的作文给出改进意见等等。

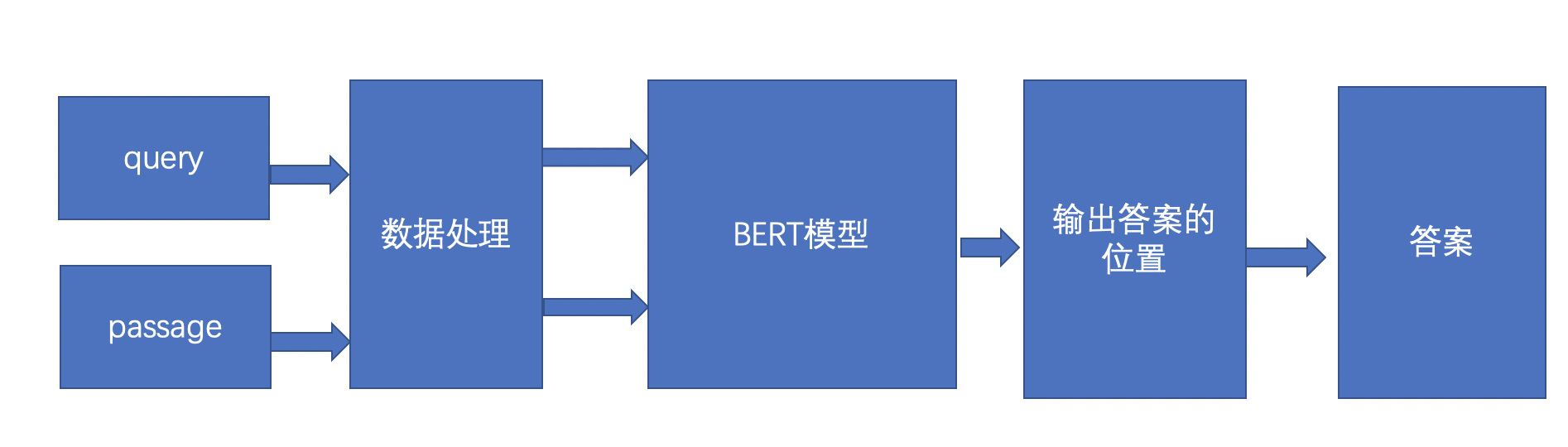


图1.1 阅读理解任务的设计方案

阅读理解的方案如上图，首先是query表示的是问句，一般是用户的提问，passage表示的是文章，表示的是query的答案要从passage里面抽取出来，query和passage经过数据预处理，得到id形式的输入，然后把query，passage的id形式输入到BERT模型里面，BERT模型经过处理会输出答案的位置，输出位置以后就可以得到相应的答案了。

具体的任务定义为：对于一个给定的问题q和一个篇章p，根据篇章内容，给出该问题的答案a。数据集中的每个样本，是一个三元组<q, p, a>，例如：

**问题 q**: 乔丹打了多少个赛季

**篇章 p**: 迈克尔.乔丹在NBA打了15个赛季。他在84年进入nba，期间在1993年10月6日第一次退役改打棒球，95年3月18日重新回归，在99年1月13日第二次退役，后于2001年10月31日复出，在03年最终退役…

**参考答案 a**: [‘15个’,‘15个赛季’]

阅读理解模型的鲁棒性是衡量该技术能否在实际应用中大规模落地的重要指标之一。随着当前技术的进步，模型虽然能够在一些阅读理解测试集上取得较好的性能，但在实际应用中，这些模型所表现出的鲁棒性仍然难以令人满意。本示例使用的DuReader-robust数据集作为首个关注阅读理解模型鲁棒性的中文数据集，旨在考察模型在真实应用场景中的过敏感性、过稳定性以及泛化能力等问题。

## 数据处理

首先导入实验所需要用到的库包。（utils、squad中涉及到的函数请见附录）

1. **import** os
2. **import** torch
3. **import** torch.nn as nn
4. **import** torch.nn.functional as F
5. **import** torch.nn.init as I
6. **from** torch.nn.utils.rnn **import** pad\_sequence
7. **import** torch.optim as optim
8. **import** torch.utils.data
10. **from** torch.utils.data **import** RandomSampler,DataLoader,SequentialSampler

13. **import** matplotlib.pyplot as plt
14. **import** transformers
15. **from** transformers **import** BertTokenizerFast,BertForQuestionAnswering,get\_linear\_schedule\_with\_warmup
17. **from** utils **import** prepare\_train\_features, prepare\_validation\_features
18. **from** squad **import** compute\_prediction,squad\_evaluate
19. **from** functools **import** partial
21. **import** collections
22. **import** time
    1. 数据集加载

对于PaddleNLP，其中已经内置SQuAD，CMRC等中英文阅读理解数据集，可通过调用函数完成加载。在本案例中我们使用DuReaderRobust中文阅读理解数据集，可通过公开链接完成数据集下载[[1]](#footnote-1)。

由于DuReaderRobust数据集采用SQuAD数据格式，InputFeature使用滑动窗口的方法生成，即一个example可能对应多个InputFeature。

答案抽取任务即根据输入的问题和文章，预测答案在文章中的起始位置和结束位置。由于文章加问题的文本长度可能大于max\_seq\_length，答案出现的位置有可能出现在文章最后，所以不能简单的对文章进行截断。所以对于过长的文章，则采用滑动窗口将文章分成多段，分别与问题组合。再用对应的tokenizer转化为模型可接受的feature。doc\_stride参数就是每次滑动的距离。滑动窗口生成InputFeature的过程如下图：

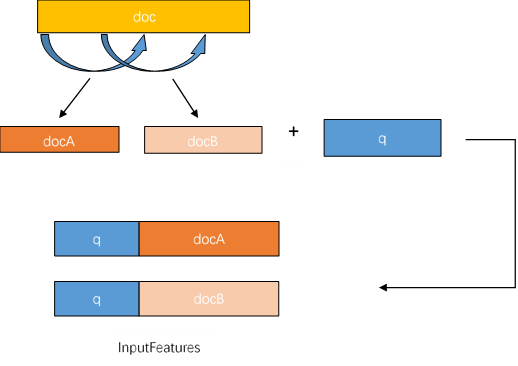


图2.1滑动窗口生成InputFeature示意图

本次实验使用的数据集为json格式的数据集，数据文件均存储在data文件夹下，读取数据集的代码示例如下：

1. **def** load\_data(filepath):
3. '''''
4. json 数据的处理
5. '''
7. train\_path = os.path.join(filepath, 'train.json')
8. dev\_path = os.path.join(filepath, 'dev.json')
10. train\_file = open(train\_path, 'r',encoding='utf-8')
11. tr\_content = train\_file.read()
12. tr\_ds = json.loads(tr\_content)
13. tr\_ds = tr\_ds['data'] # a list
15. dev\_path = open(dev\_path, 'r',encoding='utf-8')
16. dev\_content = dev\_path.read()
17. dev\_ds = json.loads(dev\_content)
18. dev\_ds = dev\_ds['data'] # a list

21. **return** tr\_ds,dev\_ds
    1. 数据格式处理

源数据集的结构比较复杂为方便后续的处理，首先将读取到的数据转化为列表，列表中的每个元素为一个字典格式数据，代表着一个实例，对数据进行处理的代码示例如下：

1. **def** extract\_data(data):
2. #源数据结构复杂，重新构建一个结构简单的字典列表，方便后续处理 train和dev数据结构基本一致，可以直接调用相同函数
4. #需要注意的是dev数据中存在一个question多个answer的情况
5. results = []
6. **for** article **in** data:
7. paragraphs = article['paragraphs']
8. **for** paragraph **in** paragraphs:
9. context = paragraph['context']
10. qas = paragraph['qas']
11. **for** q **in** qas:
12. question = q['question']
13. id = q['id']
14. answers = q['answers']
15. text\_temp = []
16. answer\_start\_temp = []
17. **for** answer **in** answers:
18. text = answer['text']
19. text\_temp.append(text)
20. answer\_start = answer['answer\_start']
21. answer\_start\_temp.append(answer\_start)
22. results.append({'question':question, 'id':id,'context':context,'answers':text\_temp,'answer\_starts':answer\_start\_temp})
24. **return** results
    1. 将数据转换为模型可以接收的形式

首先调用BertTokenizerFast进行数据处理。预训练模型Bert对中文数据的处理是以字为单位。在本案例中，我们选择调用transformers库中已经内置的tokenizer，指定想要使用的模型名字即可加载对应的tokenizer，将原始输入文本转化成模型可以接受的输入数据形式。具体的调用方法如下（本案例中选择使用的是bert-base-chinese）：

1. #调用BertTokenizer进行数据处理
2. #tokenizer的作用是将原始输入文本转化成模型可以接受的输入数据形式。
3. **def** getTokenizer(model\_name):
4. tokenizer = BertTokenizerFast.from\_pretrained(model\_name)# 使用'bert-base-chinese'
5. #这里使用BertTokennizerFast的原因是为了使用return\_offsets\_mapping参数 使用BertTokennizer会报错
6. **return** tokenizer

利用已经获取的tokenizer对数据进行进一步处理：

1. **def** trans\_features(data,tokenizer,tr\_or\_dev):
2. '''''
3. 数据集中的example将会被转换成了模型可以接收的feature，包括input\_ids、token\_type\_ids、答案的起始位置等信息。
4. 其中：
5. input\_ids: 表示输入文本的token ID。
6. token\_type\_ids: 表示对应的token属于输入的问题还是答案。（Transformer类预训练模型支持单句以及句对输入）。
7. overflow\_to\_sample: feature对应的example的编号。
8. offset\_mapping: 每个token的起始字符和结束字符在原文中对应的index（用于生成答案文本）。
9. start\_positions: 答案在这个feature中的开始位置。
10. end\_positions: 答案在这个feature中的结束位置。
11. '''
12. max\_seq\_length = 512
13. doc\_stride = 128
15. train\_trans\_func = partial(prepare\_train\_features,
16. max\_seq\_length=max\_seq\_length,
17. doc\_stride=doc\_stride,
18. tokenizer=tokenizer)
20. dev\_trans\_func = partial(prepare\_validation\_features,
21. max\_seq\_length=max\_seq\_length,
22. doc\_stride=doc\_stride,
23. tokenizer=tokenizer)
25. **if** tr\_or\_dev == 'train':
26. data = map(data,train\_trans\_func)
27. **elif** tr\_or\_dev == 'dev':
28. data = map(data,dev\_trans\_func)
30. **return** data
32. **def** map(data,trans\_func):
33. data = trans\_func(data)
34. **return** data

通过上述操作，数据集中的example已经被转换成了模型可以接收的feature，包括input\_ids、token\_type\_ids、答案的起始位置等信息。 其中：

* input\_ids: 表示输入文本的token ID。
* token\_type\_ids: 表示对应的token属于输入的问题还是答案。（Transformer类预训练模型支持单句以及句对输入）。
* overflow\_to\_sample\_mapping: feature对应的example的编号。
* offset\_mapping: 每个token的起始字符和结束字符在原文中对应的index（用于生成答案文本）。
* start\_positions: 答案在这个feature中的开始位置。
* end\_positions: 答案在这个feature中的结束位置。
  1. 构造Dataloader

使用torch.utils.data中的RandomSampler和SequentialSampler以及Dataloader，将feature组成batch。

因为训练数据和验证数据的用途不同，在格式处理上有差别，因此针对两个数据集编写不同的构造dataloader的函数：

1. **def** get\_tr\_data\_loader(tr\_ds,berttokenizer):
2. '''''
3. 构造训练数据集的dataloader
4. '''
5. batch\_size = 8
6. train\_sampler = RandomSampler(tr\_ds)
8. train\_batchify\_fn = **lambda** batch:{
9. 'input\_ids' : pad\_sequence([torch.tensor(example['input\_ids'][0]) **for** example **in** batch],batch\_first= True,padding\_value = berttokenizer.pad\_token\_id),
11. 'token\_type\_ids': pad\_sequence([torch.tensor(example['token\_type\_ids'][0]) **for** example **in** batch],batch\_first= True,padding\_value = berttokenizer.pad\_token\_type\_id),
13. 'start\_positions' : torch.tensor([example['start\_positions'] **for** example **in** batch],dtype=torch.int64),
15. 'end\_positions' : torch.tensor([example['end\_positions'] **for** example **in** batch], dtype=torch.int64)
16. }
18. train\_data\_loader = DataLoader(tr\_ds,sampler=train\_sampler,batch\_size=batch\_size,collate\_fn=train\_batchify\_fn,pin\_memory=True)
20. **return** train\_data\_loader
21. **def** get\_dev\_data\_loader(dev\_ds,berttokenizer):
23. batch\_size = 8
24. dev\_sampler = SequentialSampler(dev\_ds)
26. dev\_batchify\_fn = **lambda** batch:{
27. 'input\_ids': pad\_sequence([torch.tensor(example['input\_ids'][0]) **for** example **in** batch],batch\_first= True,padding\_value = berttokenizer.pad\_token\_id),
29. 'token\_type\_ids': pad\_sequence([torch.tensor(example['token\_type\_ids'][0]) **for** example **in** batch],batch\_first= True,padding\_value = berttokenizer.pad\_token\_type\_id)
30. }
32. dev\_data\_loader = DataLoader(dev\_ds,sampler=dev\_sampler,batch\_size = batch\_size,collate\_fn=dev\_batchify\_fn,pin\_memory=True)

35. **return** dev\_data\_loader

## 模型构建

阅读理解本质是一个答案抽取任务，答案抽取任务的本质就是根据输入的问题和文章，预测答案在文章中的起始位置和结束位置。基于BERT的答案抽取原理如下图所示：

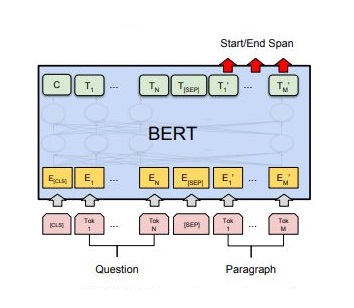


图3.1 基于BERT的答案抽取原理示意图

transformers已经内置了对于下游任务-答案抽取的Fine-tune网络。

此次使用的同样是bert-base-chinese

1. **def** get\_model(model\_name):
3. model = BertForQuestionAnswering.from\_pretrained(model\_name)
4. **return** model

## 模型配置与模型训练

* 1. 设置Fine-Tune优化策略

在本案例中使用学习率为warmup的动态学习率，动态学习率的示意图如下图所示：

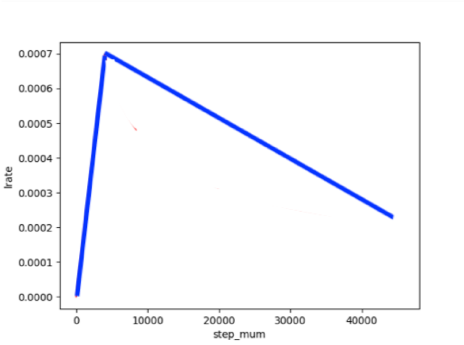


图4.1动态学习率示意图

代码示例如下：

1. # 训练过程中的最大学习率
2. learning\_rate = 3e-5
3. # 训练轮次
4. epochs = 1
5. # 学习率预热比例
6. warmup\_proportion = 0.1
7. # 权重衰减系数，类似模型正则项策略，避免模型过拟合
8. weight\_decay = 0.01
10. num\_training\_steps = len(tr\_dataloader)\*epochs
12. num\_warmup\_steps = warmup\_proportion\*num\_training\_steps
14. #学习率预热
16. # Generate parameter names needed to perform weight decay.
17. # All bias and LayerNorm parameters are excluded.
18. decay\_param = [
19. p **for** n,p **in** model.named\_parameters()
20. **if** **not** any(nd **in** n **for** nd **in** ["bias", "norm"])
21. ] #注意这里不要使用p.name
22. optimizer = optim.AdamW(decay\_param,lr = learning\_rate,weight\_decay=weight\_decay)
24. lr\_schedule = get\_linear\_schedule\_with\_warmup(optimizer=optimizer,num\_training\_steps=num\_training\_steps,num\_warmup\_steps=num\_warmup\_steps)
    1. 设计loss function

由于BertForQuestionAnswering模型对将BertModel的sequence\_output拆开成start\_logits和end\_logits进行输出，所以阅读理解任务的loss也由start\_loss和end\_loss组成，我们需要自己定义loss function。对于答案其实位置和结束位置的预测可以分别成两个分类任务。所以设计的loss function如下：

1. **class** CrossEntropyLossForSQuAD(nn.Module):
2. **def** \_\_init\_\_(self):
3. super(CrossEntropyLossForSQuAD, self).\_\_init\_\_()
5. **def** forward(self,y,label):
6. start\_logits, end\_logits = y   # both shape are [batch\_size, seq\_len]
7. start\_position, end\_position = label
8. # start\_position = start\_position.unsqueeze(-1)
9. # end\_position = end\_position.unsqueeze(-1)
11. start\_loss = F.cross\_entropy(start\_logits,start\_position,reduction ='mean')
13. end\_loss = F.cross\_entropy(end\_logits,end\_position,reduction = 'mean')
15. total\_loss =(start\_loss + end\_loss)/2
17. **return** total\_loss
    1. 训练过程

模型训练的过程通常有以下步骤：

1. 从dataloader中取出一个batch data
2. 将batch data喂给model，做前向计算
3. 将前向计算结果传给损失函数，计算loss。
4. loss反向回传，更新梯度。重复以上步骤。

每训练一个epoch时，程序通过evaluate()调用squad.py中的squad\_evaluate(), compute\_predictions()评估当前模型训练的效果，其中：

* compute\_predictions()用于生成可提交的答案；
* squad\_evaluate()用于返回评价指标。
* **def** evaluate(model, raw\_data,data\_loader,device):
* model.eval()
* all\_start\_logits = []
* all\_end\_logits = []
* tic\_eval = time.time()
* **for** batch **in** data\_loader:
* input\_ids = batch['input\_ids'].to(device)
* token\_type\_ids = batch['token\_type\_ids'].to(device)
* logits = model(input\_ids,token\_type\_ids)
* start\_logits\_tensor = logits['start\_logits'].cpu()
* end\_logits\_tensor = logits['end\_logits'].cpu()
* **for** idx **in** range(start\_logits\_tensor.shape[0]):
* **if** len(all\_start\_logits) % 1000 == 0 **and** len(all\_start\_logits):
* **print**("Processing example: %d" % len(all\_start\_logits))
* **print**('time per 1000:', time.time() - tic\_eval)
* tic\_eval = time.time()
* all\_start\_logits.append(start\_logits\_tensor.numpy()[idx])
* all\_end\_logits.append(end\_logits\_tensor.numpy()[idx])
* all\_predictions, \_ ,\_ = compute\_prediction(raw\_data,data\_loader.dataset,(all\_start\_logits,all\_end\_logits),False,20,30)
* out\_eval=squad\_evaluate(examples=raw\_data,preds= all\_predictions,is\_whitespace\_splited=False)
* **print**(out\_eval)
* model.train()

上述内容完成后，即可开始模型的训练：

1. **print**("开始训练")
2. **for** epoch **in** range(epochs+1):
3. **for** step,batch **in** enumerate(tr\_dataloader,start=1):
4. global\_step += 1
5. # input\_ids,segment\_ids,start\_positions,end\_positions = batch
6. input\_ids = batch['input\_ids'].to(device)
7. segment\_ids = batch['token\_type\_ids'].to(device)
8. start\_positions = batch['start\_positions'].to(device)
9. end\_positions = batch['end\_positions'].to(device)
10. logits = model(input\_ids = input\_ids,token\_type\_ids = segment\_ids)
12. start\_logits = logits['start\_logits'].to(device)
13. end\_logits = logits['end\_logits'].to(device)
14. logits = (start\_logits,end\_logits)
16. loss = criterion(logits,(start\_positions,end\_positions)) #y, label
18. **if** global\_step % 100 == 0 :
19. **print**("global step %d, epoch: %d, batch: %d, loss: %.5f" % (global\_step, epoch, step, loss))
21. loss.backward()
22. optimizer.step()
23. lr\_schedule.step()
24. optimizer.zero\_grad()
26. evaluate(model = model ,raw\_data = dev\_ds\_raw ,data\_loader = dev\_dataloader,device = device)

29. **print**("正在保存模型")
30. model.save\_pretrained('./checkpoint')
31. tokenizer.save\_pretrained('./checkpoint')
32. **print**('-------------------------------------------------')

在示例中，将模型训练相关的部分均放置在train()函数部分，请参阅示例代码以知晓函数所需参数与返回内容。

## 模型预测

通过train()函数进行模型训练后得到的模型，可以用于下一步预测工作：

1. @torch.no\_grad()
2. **def** do\_predict(model,raw\_data,data\_loader,device):
4. **print**('------------------------------------------------')
5. **print**("开始模型预测")
6. model.eval()
7. all\_start\_logits = []
8. all\_end\_logits = []
9. tic\_eval = time.time()
10. **for** batch **in** data\_loader:
11. # input\_ids,token\_type\_ids = batch
12. input\_ids = batch['input\_ids'].to(device)
13. token\_type\_ids = batch['token\_type\_ids'].to(device)
14. # start\_logits\_tensor,end\_logits\_tensor = model(input\_ids,token\_type\_ids)
15. logits = model(input\_ids,token\_type\_ids)
16. start\_logits\_tensor = logits['start\_logits'].cpu()
17. end\_logits\_tensor = logits['end\_logits'].cpu()
18. **for** idx **in** range(start\_logits\_tensor.shape[0]):
19. **if** len(all\_start\_logits) % 1000 == 0 **and** len(all\_start\_logits):
20. **print**("Processing example: %d" % len(all\_start\_logits))
21. **print**('time per 1000:', time.time() - tic\_eval)
22. tic\_eval = time.time()
24. all\_start\_logits.append(start\_logits\_tensor.numpy()[idx])
25. all\_end\_logits.append(end\_logits\_tensor.numpy()[idx])


29. all\_predictions, \_, \_ = compute\_prediction(
30. raw\_data, data\_loader.dataset,
31. (all\_start\_logits, all\_end\_logits), False, 20, 30)
32. **print**("模型预测结束")
33. **print**('------------------------------------------------')
35. count = 0
36. **for** example **in** raw\_data:
38. count += 1
39. **print**()
40. **print**('问题：',example['question'])
41. **print**('原文：',''.join(example['context']))
42. **print**('答案：',all\_predictions[example['id']])
43. **if** count >=5:
44. **break**
46. model.train()

main()函数：

1. **if** \_\_name\_\_ == '\_\_main\_\_':
3. filepath = './data'
5. tr\_ds\_raw,dev\_ds\_raw = load\_data(filepath)
6. tr\_ds\_raw = extract\_data(tr\_ds\_raw)
7. dev\_ds\_raw = extract\_data(dev\_ds\_raw)

10. device = torch.device("cuda" **if** torch.cuda.is\_available() **else** "cpu")
12. model\_name = 'bert-base-chinese'
14. berttokenizer = getTokenizer(model\_name)
15. model = get\_model(model\_name)
16. model = model.to(device)
18. tr\_ds = trans\_features(tr\_ds\_raw,tokenizer = berttokenizer,tr\_or\_dev='train')
19. dev\_ds = trans\_features(dev\_ds\_raw,tokenizer = berttokenizer,tr\_or\_dev='dev')

22. tr\_dataloader = get\_tr\_data\_loader(tr\_ds, berttokenizer)
23. dev\_dataloader = get\_dev\_data\_loader(dev\_ds,berttokenizer)
25. model = train(dev\_ds\_raw=dev\_ds\_raw,model = model ,tokenizer=berttokenizer, tr\_dataloader=tr\_dataloader ,dev\_dataloader=dev\_dataloader,device=device)
27. do\_predict(model=model , raw\_data= dev\_ds\_raw ,data\_loader = dev\_dataloader , device=device)

1. [dureader\_robust - 飞桨AI Studio (baidu.com)](https://aistudio.baidu.com/aistudio/datasetdetail/75194) [↑](#footnote-ref-1)