7. bert预训练模型

全称： BERT (Bidirectional Encoder Representations from Transformers)

网络上一些资料：

- tokenizer的用法
- 分类：能使用bert预训练模型训练下游任务的 种类
- 由浅入深从源码解释bert-pretrianed
- BERT的[CLS]有什么用
- 简书 最通俗易懂的BERT原理与代码实现

西二第五轮考核资料：

- level:0 BERT实战（上） - 简书 (jianshu.com)
- https://huggingface.co/docs/transformers/tasks/sequence_classification
- (BERT实现简单的文本分类 - 知乎 (zhihu.com))
- 考核文档
- 巨佬文章

7.1 参数量的计算

引入：

> bert-base-chinese: 编码器具有12个隐层, 输出768维张量, 12个自注意力头, 共110M参数量, 在简体和繁体中文文本上进行训练而得到.
>
> **重要性： 求解Bert模型的参数量是面试常考的问题，也是作为算法工程师必须会的一个点。**
>
> 目前，预训练模型在NLP领域占据核心地位。预训练模型的参数量是庞大的，例如BERT(base)的参数量是110M，BERT(large)的参数量是340M

主流bert模型参数：

> - BERT-Base, Uncased 12层，768个隐单元，12个Attention head，110M参数
> - BERT-Large, Uncased 24层，1024个隐单元，16个head，340M参数
> - BERT-Base, Cased 12层，768个隐单元，12个Attention head，110M参数
> - BERT-Large, Uncased 24层，1024个隐单元，16个head，340M参数。

以BERT(base)为例进行计算：

Bert 的模型由多层双向的Transformer编码器组成，由12层组成，768隐藏单元，12个head，总参数量110M，约1.15亿参数量。

> 1. 各种参数准备：
>
> | Parameters in BERT（base) | Number |
> | --- | --- |
> | vocab_size | 30522 |
> | layer | 12 |
> | hidden size | 768 |

Parameters in BERT（base)	Number
max length	512
multi head attention	12
inner size	3072

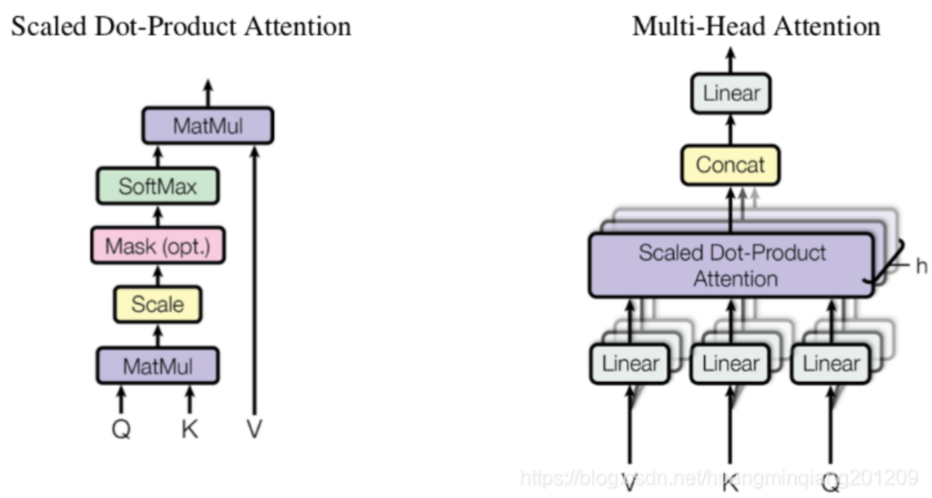
2. 分为四大部分：
   - **词向量参数：**

     词向量包括三个部分的编码：词向量参数，位置向量参数，句子类型参数。

```
词汇量的大小 vocab_size=30522

隐藏层 hidden_size=768 （即词向量维度d_model=768）

文本输入最长大小 max_position_embeddings=512

词向量参数 token embedding=30522*768

位置向量参数 position embedding=512*768

句子类型参数 Segment embedding=2*768（2个句子，0和1区分上下句子）

故：embedding总参数 = （30522+512+2）*768 = 23,835,648 = 23 MB 约，无偏置
```

   - **multi-heads参数（Multi-Heads Attention）：**

     多头注意力结构如下：



```
multi-head 因为先分成12份然后再 concat 在一起

单个head的参数是 768 * 768/12 * 3  (*3 是 有q,k,v三个矩阵)

12个head就是 768 * 768/12 * 3 * 12

紧接着将多个 head 进行 concat 再进行变换，此时W的大小是 768 * 768

所以这个部分是 768 * 768/12 * 3 * 12 + 768 * 768 = 2359296

12层multi-head 2359296 * 12 =  28,311,552 = 27MB 约，无偏置
```

- **全连接层（FeedForward）参数:**

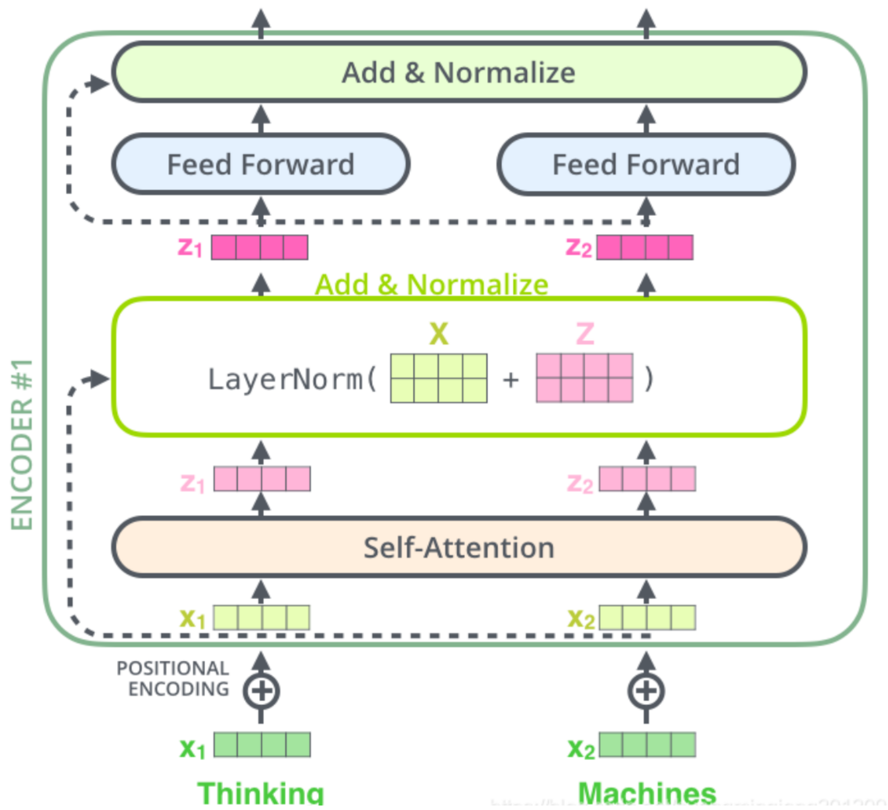
  前馈网络feed forword的参数主要由2个全连接层组成，论文中全连接层的公式为:

  FFN(x) = max(0, xW1 + b1)W2 + b2

  其中用到了两个参数W1和W2，Bert沿用了惯用的全连接层大小设置，即4 * dmodle = 3072，因此，W1，W2分别为（768, 3072），（3072, 768）

```
12层的全连接层参数为：12 *（ 2 * 768 * 3072）= 56,623,104 = 54MB 约，无偏置
```

- **LayerNorm层:**




```
  LN层有gamma和beta等2个参数。在三个地方用到了layernorm层：embedding层后、multi-head attention后、feed forward后。
  故，12层LN层参数为：768*2 + (768*2)*12 + (768*2)*12 = 38,400 = 37.5KB
```

上述四部分，加上偏置bias和基于encoder的两个任务next sentence prediction 和 MLM涉及的参数（分别是768 * 2，768 * 768，总共约0.5M） 共计约110M参数。

7.2 BertTokenizer 和 BertModel 的基础语法

输入：

- **导包** （可以下载，也可以直接联网导入，有时候可能需要翻墙网速会高一点）

```
from transformers import BertTokenizer, BertModel
import torch
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")
```

- **编码可以同时编码多个句子，但是解码一次只能解一个句子**
- **tokenizer.tokenize()** 先分词，转化出来每一个 token

```
sentence = 'I love China'
print('句子: {}'.format(sentence))
# 句子: I love China

tokens = tokenizer.tokenize(sentence)
print('分词: {}'.format(tokens))
# 分词: ['i', 'love', 'china']
```

- **tokenizer.convert_tokens_to_ids** 将 token 转化为 id

```
tokens = ['[CLS]', 'i', 'love', 'china', '[SEP]', '[PAD]', '[PAD]']
input_ids = tokenizer.convert_tokens_to_ids(tokens)
print('将标记转化为标记id: {}'.format(input_ids))
# 将标记转化为标记id: [101, 1045, 2293, 2859, 102, 0, 0]
```

- **tokenizer.encode()** 将句子编码，会补在句首补[CLS],句尾补[SEP]
    - 约等于 tokenizer(sentence).input_ids = tokenizer(sentence)['input_ids']

```
sentence = '吾儿莫慌'
print(tokenizer.encode(sentence))
# [101, 1434, 1036, 5811, 2707, 102]
```

- **tokenizer.encode_plus (与tokenizer效果类似)**

可以给两个句子同时编码，中间直接加上[SEP]分隔符，分隔符算前一个句子。与tokenizer类似

```
sentence1 = '我爱中国，我是中国人'
sentence = 'I love China'
print(tokenizer.encode_plus(sentence, sentence1))
{'input_ids': [101, 100, 8451, 100, 102, 2769, 4263, 704, 1744, 8024, 2769, 3221,
704, 1744, 782, 102], 'token_type_ids': [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

- **tokenizer()** 比encode_plus()更好用：
    - **参数：**
    - max_length = 128 输入句子最大长度是128 不足则会补充pad
    - truncation=True 超过128的句子允许截断

- padding = 'max_length' 允许补充pad以保证长度一致
- **输出：**
- input_ids：是单词在词典中的编码
- token_type_ids：区分两个句子的编码（上句全为0，下句全为1）
- attention_mask：指定 对哪些词 进行self-Attention操作

```
sentence1 = '这个故事没有终点'
sentence2 = '正如星空没有彼岸'
seq_code = tokenizer(sentence1,sentence2,max_length=128,truncation=True,padding = 'max_length')

print(seq_code)
'''
{'input_ids': [101, 6821, 702, 3125, 752, 3766, 3300, 5303, 4157, 102, 3633,
1963, 3215, 4958, 3766, 3300, 2516, 2279, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'token_type_ids': [0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}
'''
# 如果是tokenizer() 没有后面的参数：    则是与tokenizer.encode_plus()一致
seq_code = tokenizer(sentence1,sentence2)
seq_code['input_ids'] = [101, 6821, 702, 3125, 752, 3766, 3300, 5303, 4157, 102,
3633, 1963, 3215, 4958, 3766, 3300, 2516, 2279, 102]
seq_code['token_type_ids'] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1]
seq_code['attention_mask'] = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]
```

- **tokenizer.decode()**   对序号 id 进行解码

```
# 将标记转化为标记id: [101, 1045, 2293, 2859, 102, 0, 0]
decode_ids = tokenizer.decode(input_ids)
print('标记id解码成标记: {}'.format(decode_ids))
# 标记id解码成标记: [CLS] i love china [SEP] [PAD] [PAD]
```

经过bert传递后的输出数据类型：

- outputs = bert(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)

如果不传入，后面三个参数的话，**bert也会自动进行处理**，但是效果可能不好。

- 如上边调用Bert模型时，**输出结果out**中包含last_hidden_state、pooler_output、hidden_states、past_key_values、attentions、cross_attentions**几个参数属性如下。**

> 1. `last_hidden_state`: 这是BERT模型最后一个隐藏层的输出。它是一个形状为 `[batch_size, sequence_length, hidden_size]` 的张量，表示每个输入令牌的上下文相关表示。这个张量包含了输入序列中每个位置的隐藏状态信息。
> 2. `pooler_output`: 这是BERT模型经过池化操作得到的输出。它是一个形状为 `[batch_size, hidden_size]` 的张量，表示整个输入序列的池化表示。它通常被用作**句子级别**的表示，用于下游任务的分类或句子级别的特征提取。
> 3. hidden_states: 这是BERT模型中所有隐藏层的输出。它是一个包含每个隐藏层输出的列表，其中每个元素的形状为 [batch_size, sequence_length, hidden_size]。hidden_states[0] 表示第一个隐藏层的输出，hidden_states[1] 表示第二个隐藏层的输出，以此类推，hidden_states[-1] 表示最后一个隐藏层的输出（即 last_hidden_state）。这些隐藏层输出可以理解为**字级输出**，可以用于更详细的分析或进行一些特殊 任务如mask预测
> 4. past_key_values: 这是用于生成下一个预测令牌的先前键值对。它是一个元组，其中包含了前几次调用BERT模型时生成的先前键值对。它通常在生成任务（如文本生成）中使用，以便在多步预测中保留先前的状态信息。
> 5. attentions: 这是**自注意力机制产生的注意力权重。**它是一个列表，包含每个注意力头的注意力权重矩阵。注意力权重矩阵的形状为 [batch_size, num_heads, sequence_length, sequence_length]，表示模型在**每个位置上关注其他位置的程度。**
> 6. cross_attentions: 这是BERT模型中的**交叉注意力机制产生的注意力权重。**它是一个列表，包含每个交叉注意力头的注意力权重矩阵。注意力权重矩阵的形状为 [batch_size, num_heads, sequence_length, sequence_length]，**表示模型在每个位置上关注另一个输入序列（如句子级别的任务中的两个句子）的程度。**
> 7. 这些属性提供了BERT模型在不同层级和注意力机制上的输出信息，**可以根据任务的需求选择合适的属性来使用。**

```
sentence_a = '这是一个短句子。'
sentence_b = '这是一个更长的句子。 他比第一个句子更长一点。'
inputs = tokenizer(sentence_a, sentence_b, max_length=128,padding='max_length')
# .unsqueeze(0) 为了使得测试的输入维度与训练的输入维度相同，即batch_size = 1
input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
attention_mask = torch.tensor(inputs['attention_mask']).unsqueeze(0)
token_type_ids = torch.tensor(inputs['token_type_ids']).unsqueeze(0)
outputs = bert(input_ids, attention_mask=attention_mask,
token_type_ids=token_type_ids)
pooler_output = outputs['pooler_output']
# [batch_size, hidden_size]
print('pooler_output shape: {}'.format(pooler_output.shape))
# pooler_output shape: torch.Size([1, 768])
last_hidden_state = outputs['last_hidden_state']
print('last_hidden_state shape: {}'.format(last_hidden_state.shape))
w# [batch_size, sequence_length, hidden_size]
# last_hidden_state shape: torch.Size([1, 26, 768])
```

具体解释如下：

> `hidden_states = outputs['hidden_states']`：获得13 * [batch_size, sequence_length, hidden_size] 的张量
>
> `hidden_states[0]`：输入嵌入层的输出 shape：[batch_size, sequence_length, hidden_size]

> `hidden_states[-1]`：最后一个编码器层的输出 shape: [batch_size, sequence_length, hidden_size]
>
> `pooler_output`：句子级特征，用于分类或者情感分析哦 shape: [batch_size, hidden_size]

采用其他方式表达输出：

> outputs = bert_model(tokens_tensor, token_type_ids = segments_tensors)
>
> outputs 按顺序依次 是：
>
> outputs = `sequence_output, pooled_output, (hidden_states), (attentions)`

```
# 对编码进行转换，以便输入Tensor
tokens_tensor = torch.tensor([sen_code['input_ids']])  # 添加batch维度并,转换为合适大小tensor,torch.Size([1, 19])
segments_tensors = torch.tensor(sen_code['token_type_ids']) # torch.Size([19])
bert_model.eval()
# 进行编码
with torch.no_grad():
    outputs = bert_model(tokens_tensor, token_type_ids = segments_tensors)
    encoded_layers = outputs   # outputs类型为tuple

    print(encoded_layers[0].shape, encoded_layers[1].shape,
          encoded_layers[2][0].shape, encoded_layers[3][0].shape)
    # torch.Size([1, 19, 768]) torch.Size([1, 768])
    # torch.Size([1, 19, 768]) torch.Size([1, 12, 19, 19])
# Bert最终输出的结果维度为：sequence_output, pooled_output, (hidden_states), (attentions)
```

7.3 冻结参数训练 与 超参数选择

- **超参数选择**：

对于微调，除了批量大小、学习率和训练次数外，大多数模型超参数与预训练期间相同。Dropout 概率总是使用 0.1。最优超参数值是特定于任务的，但我们发现以下可能值的范围可以很好地在所有任务中工作:

- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 3, 4

- **冻结参数训练：**

可以先输出模型model，确定每一层的名字，选择适当的层进行训练。

```
class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,200)
        self.dropout = nn.Dropout()
        self.classifier = nn.Linear(200,label_nums)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
```

```
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)
    def forward(self,x,attention_masks,token_type_ids):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['pooler_output']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        out = self.classifier(x)
        return out
```

7.4 微调模型 fine-tuning [一] 分类

中文语言理解测评基准我们采用CLUE benchmark，它分为多个测试数据集，最终的测试成绩为所有数据集上成绩的算数平均，本轮各位需要运行的数据集有：AFQMC' 蚂蚁语义相似度、IFLYTEK' 长文本分类、TNEWS' 今日头条中文新闻（短文本）分类三个数据集，其他数据集如果有余力可以自行测试并一起提交，其他数据集的下载以及微调时的参数可以参照CLUE benchmark的GitHub地址。

- **IFLYTEK' 长文本分类** label_nums = 119

```
'''
iflytek 数据集  119分类问题
2023.07.17 西二第五轮考核 nlp_bert_clue测试
只训练最后一个pooler模块 epoch 80左右 acc 55% 还在增长但是很缓慢
全部训练参数加一个全连接层200 在分类层119 dropout  epoch 5 acc 61%
'''
from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")
# 导入工具包
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import Dataset,DataLoader

path1 = 'D:\S\iflytek_public\\train.json'
path2 = 'D:\S\iflytek_public\\dev.json'

# 读取数据
def make_data(filename):
    data = []
    label = []
    with open(filename, 'r', encoding='utf-8') as fp:
        for line in fp.readlines():
        # read()方法将fp(一个支持.read()的文件类对象，包含一个JSON文档)转换成字符串
            obj = json.loads(line)
            data.append(obj['sentence'])
            label.append(obj['label'])
    return data,label
train_data,train_label = make_data(path1)
```

```python
test_data,test_label = make_data(path2)
# print(f'训练集大小：{len(train_data)}   测试集大小：{len(test_data)}')
# print(train_label[0],train_data[0])

# 构建自定义的dataset
class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=200, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target = torch.tensor(int(self.y[index]))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

batch_size = 1
train_dataset = Mydataset(train_data,train_label)
train_loader =
DataLoader(train_dataset,batch_size=batch_size,shuffle=True,drop_last=True)
test_dataset = Mydataset(test_data,test_label)
test_loader =
DataLoader(test_dataset,batch_size=batch_size,shuffle=True,drop_last=True)
print(test_loader.dataset[15])
# # len(label) = 119

# 构建微调的模型
# len(label) = 119
label_nums = 119
class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,200)
        self.dropout = nn.Dropout()
        self.classifier = nn.Linear(200,label_nums)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)
    def forward(self,x,attention_masks,token_type_ids):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['pooler_output']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        out = self.classifier(x)
        return out

def accuracy(predictions,labels):
```

```
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights

model =  MypretrainModel()
criterion = F.cross_entropy
optimizer = optim.Adam(model.parameters(),lr=1e-5)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)
train_losses = []
test_losses = []
epochs = 5
savename = 'iflytek_bert'
print(f'./{savename}_model.pth')
max_acc = 0
def train():
    for epoch in range(1, epochs + 1):
        model.train()
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in
loop:
            total += 1
            input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx +
len(input_ids)))

        train_losses.append(running_loss / total)

        # 开始测试
        model.eval()
        test_loss = 0
        correct = 0
        with torch.no_grad():
            for input_ids, attention_masks, token_type_ids, target in
test_loader:
                input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                    device), token_type_ids.to(device), target.to(device)
                output = model(input_ids, attention_masks, token_type_ids)
                test_loss += F.cross_entropy(output, target,
size_average=False).item()
```

```
                correct += accuracy(output, target)

        test_loss /= len(test_loader.dataset)
        test_losses.append(test_loss)
        print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.3f}%)\n'.format(
            test_loss, correct, len(test_loader.dataset),

            100. * correct / len(test_loader.dataset)))
        # 测试结束
        if correct / len(test_loader.dataset) > max_acc:
            max_acc = correct / len(test_loader.dataset)
            torch.save(model.state_dict(), f'./{savename}_model.pth')
            best_model_wts = copy.deepcopy(model.state_dict())
train()
```

- AFQMC' 蚂蚁语义相似度　近似二分类 label_nums = 2 好坏之分
- TNEWS' 今日头条中文新闻（短文本） 分类 label_nums = 15 且分类标签不连续

7.5 微调模型 fine-tuning [二] 小说续写

整体思路： Bert+Lstm

LSTM网络 回顾:

针对RNN多加了一部分 选择遗忘部分，加了输入门，遗忘门，输出门三态门。能够处理较长序列的问题

LSTM 参数:

> **官方定义:**
>
> - **input_size** – The number of expected features in the input *x*
> - **hidden_size** – The number of features in the hidden state *h*
> - **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
> - **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
> - **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
> - **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
> - **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
> - **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

具体理解主要参数:

> LSTM一共有7个输入参数，只挑选重要的理解
>
> 1. input_size：输入的特征维度，一般来说就是字向量的维度，比如如果用bert(base)的话，那么输入的维度input_size=768。如果在时间序列预测中，比如需要预测负荷，每一个负荷都是一个单独的值，都可以直接参与运算，因此并不需要将每一个负荷表示成一个向量，此时input_size=1。 但如果我们使用多变量进行预测，比如我们利用前24小时每一时刻的[负荷、

风速、温度、压强、湿度、天气、节假日信息]来预测下一时刻的负荷，那么此时input_size=7。

2. hidden_size：隐藏层的维度,这里我的理解是输出的特征维度。比如将bert的output[0]的768维的转变为512维度，这512就是hidden_size。也是隐藏层输出节点的个数
3. num_layers：很好理解，就是LSTM 堆叠的层数,默认值为1，设置为2的时候，第一层的输出是第二层的输入。
4. batch_first：默认为False，在制作数据集和数据集载入的时候，有个参数叫batch_size，也就是一次输入几个数据，lstm的输入默认将batch_size放在第二维，当为True的时候，则将batch_size放在第一维。
5. dropout：神经网络防止过拟合的参数。

模型的 Inputs:

> **Inputs**: input, (h_0, c_0)
>
> **input (seq_len, batch, input_size) 分别是:**
>
> seq_len ： 时间步数或序列长度 其实也是隐藏神经元的个数
>
> batch ： batch_size数
>
> input_size ： 输入特征维度。
>
> 如果设置了batch_first，则batch为第一维。
>
> **(h_0, c_0) 隐层状态f分别是:**
>
> h0 shape：(num_layers * num_directions, batch, hidden_size)
>
> c0 shape：(num_layers * num_directions, batch, hidden_size)

> **Outputs:** output, (h_n, c_n)
>
> output (seq_len, batch, hidden_size * num_directions)
>
> 包含每一个时刻的输出特征，如果设置了batch_first，则batch为第一维
> (h_n, c_n) 隐层状态

- **通用代码块:**

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size,
batch_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.num_directions = 1 # 单向LSTM
        self.batch_size = batch_size
        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input_seq):
        batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
        h_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
```

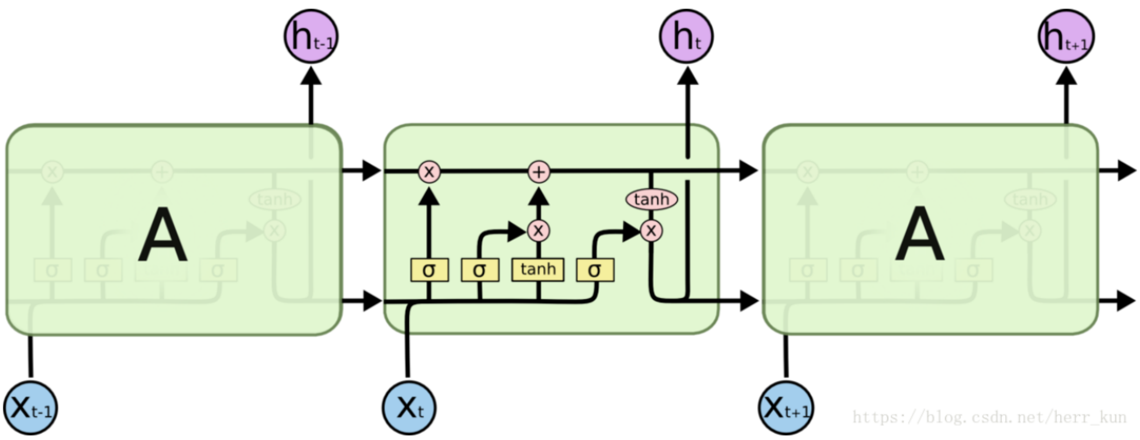
```
        c_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size, self.hidden_size).to(device)
        # output(batch_size, seq_len, num_directions * hidden_size)
        output, _ = self.lstm(input_seq, (h_0, c_0)) # output(5, 30, 64)
        pred = self.linear(output)  # (5, 30, 1)
        pred = pred[:, -1, :]  # 单个预测的分类 模型(5, 1)
        return pred

# 定义模型
self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers, batch_first=True)
self.linear = nn.Linear(self.hidden_size, self.output_size)
# 加上参数
self.lstm = nn.LSTM(self.input_size=1, self.hidden_size=64, self.num_layers=5, batch_first=True)
self.linear = nn.Linear(self.hidden_size=64, self.output_size=1)
# 输入内容 和 输出内容
output, _ = self.lstm(input_seq, (h_0, c_0))  # output(5, 30, 64)
```

LSTM 层数理解：

Seq_len： 其实就是一层LSTM的神经元的个数

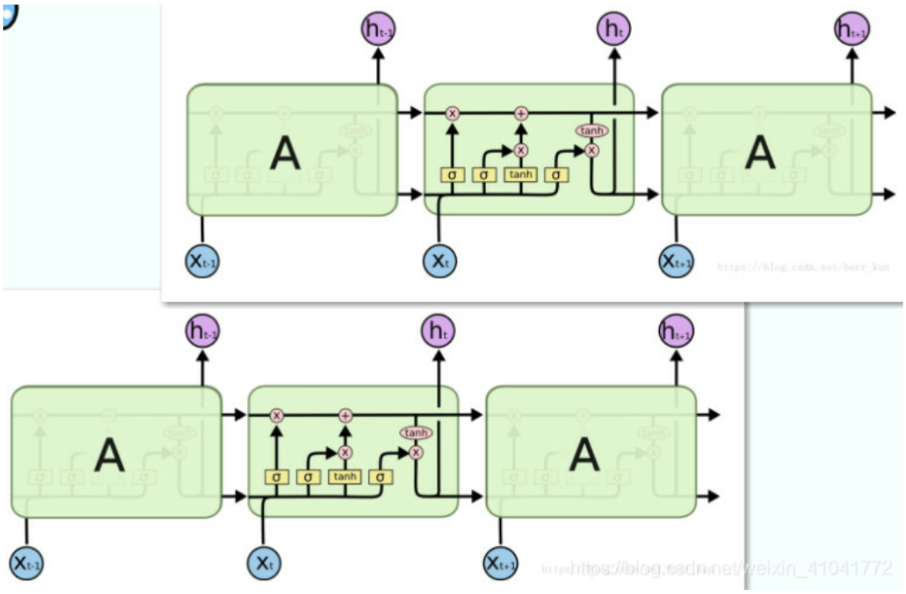
单层LSTM：

> 此结构包含3个LSTM单元，seq_len=3



两层LSTM:

> **第一层的3个时间步的多维隐藏输出作为第二层的3个时间步的输入.**
>
> **并且初始h0((2 * num_directions, batch, hidden_size))默认为0初始化。**



小说可优化方案:

> **样本数据构成优化:**
>
> - 采用不同的固定seq_len： 32 64 128进行预测后面一个词语，但是太短学习的**语义不足**，太长的训练**开销太大**。
>   - 连续样本 应该是连续更好
>   - 间隔样本
> - 采用不固定长度的但有最大长度的seq_len，在末尾加一个[MASK]，预测MASK的值，再小于最大长度时候直接预测，大于等于最大长度的时候循环迭代预测。
> - 小说样本要多，种类要丰富，否则会过于特定，或者会记住一些名字。
> - 出现的不合适的地方： 先用几篇小说，过拟合了，导致记住了局部信息
>
> **网络结构优化可以尝试:**
>
> - 直接bert后面 + 两个全连接层 进行分类。
> - 可以再使用bert + lstm + 自注意机制生成。
> - 采用apex,混合精度训练，用半精度训练，用单精度测试。
> - 试试在bert用34 在LSTM用32 不传入cls 和 seq直接舍弃开头和结尾的编码的句子。
> - 调整超参数优化，开始学习率高，后来学习率低。
>
> **文本生成优化可以尝试:**
>
> - TopK 选择预测概率最高的K个词，如果只选一个，生成的文本单一；
> - 忽略特殊字符[UNK] [PAD]，这些词语会对后续的文本生成造成干扰；
> - 重复词惩罚，如果一直重复会使得文本生成陷入循环，没有意义；
> - 集束搜索Beam-search

- ## model.generare()文本生成 参数

> [hugging face github的官方参数]
>
> Hugging Face 中的生成工具主要用于实现[文本生成]任务，包括机器翻译、文本摘要、对话生成等。这些工具基于 Transformer 模型，其中最为常用的是 GPT-2、GPT-3 和 T5 等。是自回归文本生成预训练模型相关参数的集大成者。主要是 **Greedy Search**、**Beam Search**、**Sampling**（**Temperature**、**Top-k**、**Top-p**）等各个算法的原理。

```
    @torch.no_grad()
    def generate(
        self,
        inputs: Optional[torch.Tensor] = None,
        max_length: Optional[int] = None,
        min_length: Optional[int] = None,
        do_sample: Optional[bool] = None,
        early_stopping: Optional[bool] = None,
        num_beams: Optional[int] = None,
        temperature: Optional[float] = None,
        top_k: Optional[int] = None,
        top_p: Optional[float] = None,
        typical_p: Optional[float] = None,
        repetition_penalty: Optional[float] = None,
        bad_words_ids: Optional[Iterable[int]] = None,
        force_words_ids: Optional[Union[Iterable[int], Iterable[Iterable[int]]]] = None,
        bos_token_id: Optional[int] = None,
        pad_token_id: Optional[int] = None,
        eos_token_id: Optional[int] = None,
        length_penalty: Optional[float] = None,
        no_repeat_ngram_size: Optional[int] = None,
        encoder_no_repeat_ngram_size: Optional[int] = None,
        num_return_sequences: Optional[int] = None,
        max_time: Optional[float] = None,
        max_new_tokens: Optional[int] = None,
        decoder_start_token_id: Optional[int] = None,
        use_cache: Optional[bool] = None,
        num_beam_groups: Optional[int] = None,
        diversity_penalty: Optional[float] = None,
        prefix_allowed_tokens_fn: Optional[Callable[[int, torch.Tensor], List[int]]] = None,
        logits_processor: Optional[LogitsProcessorList] = LogitsProcessorList(),
        renormalize_logits: Optional[bool] = None,
        stopping_criteria: Optional[StoppingCriteriaList] = StoppingCriteriaList(),
        constraints: Optional[List[Constraint]] = None,
        output_attentions: Optional[bool] = None,
```

7.6 训练过程代码

1. bert + lstm + bert全训练参数 最初版本

```
import os,sys
import numpy as np
from torch.cuda.amp import GradScaler, autocast
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import Dataset,DataLoader
# todo 每次训练要改slide model保存路径
# 预处理  输出是 32连续字符 label是后面接着的字
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
```

```
    y = []
    for sub_folder in dirs:
        for path in os.listdir('./data/'+ sub_folder):
            path1 = r'./data/'+ sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1 :
                continue
            for i in range(len(data) - slide):
                x.append(data[i:i + slide])
                y.append(data[i + slide])
            print(sub_folder + ' ' + path.replace('.txt',' ') +'已经读取完毕')
    return x,y
# 读txt文件 输出无空格和换行的纯文本
def read_txt(path):
    with open(path, "r", encoding="utf-8") as f:
        content = f.readlines()
        if len(content) >= 2 :
            content = content[1].replace(u'\xa0', u'').replace(u'\u3000\u3000',
u'').replace('\n','').replace(' ','')
        else:
            content = ''
            return content
    return content

from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")

class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=slide+2, truncation=True
,padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target =
torch.tensor(int(tokenizer.convert_tokens_to_ids(self.y[index])))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

# # len(label) = 119

class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size,
batch_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.num_directions = 1 # 单向LSTM
```

```
        self.batch_size = batch_size
        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
        # 分类的全链接层
        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input_seq):
        batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
        #  前向传播过程新生成的变量，需要传递到device中去
        # 如果是测试  batch_size = 1
        if input_seq.size(0) == 1:
            self.batch_size = 1
        h_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        c_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        # output(batch_size, seq_len, num_directions * hidden_size)
        output, _ = self.lstm(input_seq, (h_0, c_0)) # output(bs, seq_len,
hidden_size)
        pred = self.linear(output)  # (bs, seq_len, output_size)
        pred = pred[:, -1, :]  # (bs, output_size)
        return pred

class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,512)
        self.dropout = nn.Dropout()
        self.lstm = LSTM(input_size=512, hidden_size=256, num_layers=1,
output_size=len(tokenizer), batch_size=batch_size)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)
    def forward(self,x,attention_masks=None,token_type_ids=None):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['last_hidden_state']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        x = self.lstm(x)
        return x

# 训练文件
def train(epochs):
    for epoch in range(1, epochs + 1):
        model.train()
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in
loop:
```

```
            total += 1
            input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx +
len(input_ids)))

        train_losses.append(running_loss / total)

        # 开始测试
#         model.eval()
#         test_loss = 0
#         correct = 0
#         with torch.no_grad():
#             for input_ids, attention_masks, token_type_ids, target in
test_loader:
#                 input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
#                     device), token_type_ids.to(device), target.to(device)
#                 output = model(input_ids, attention_masks, token_type_ids)
#                 test_loss += F.cross_entropy(output, target,
size_average=False).item()
#                 correct += accuracy(output, target)

#         test_loss /= len(test_loader.dataset)
#         test_losses.append(test_loss)
#         print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.3f}%)\n'.format(
#             test_loss, correct, len(test_loader.dataset),

#             100. * correct / len(test_loader.dataset)))
        # 测试结束
        # if correct / len(test_loader.dataset) > max_acc:
        #     max_acc = correct / len(test_loader.dataset)
        torch.save(model.state_dict(), f'./{savename}_model.pth')
        torch.save(optimizer.state_dict(), f'./{savename}_optimizer.pth')
        best_model_wts = copy.deepcopy(model.state_dict())
def accuracy(predictions,labels):
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights

if __name__ == "__main__":
    #todo 设置相关变量
    folder = '/root/data'
    slide = 32
```

```
    savename = 'bert_lstm'
    path = f'/root/2{savename}_model.pth'
    data, label = make_dataset(folder, slide)
    print('输出处理完成 一共', len(data))
    #todo  定义数据集
    batch_size = 64
    train_dataset = Mydataset(data, label)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    print(train_loader.dataset[2])
    #todo 设置模型相关信息
    model = MypretrainModel(is_lock=False)
    criterion = F.cross_entropy
    optimizer = optim.Adam(model.parameters(), lr=1e-5)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
    train_losses = []
    test_losses = []
    epochs = 1
    max_acc = 0
    scaler = GradScaler()

    print(path)
    if not os.path.exists(path):
        print('不存在路径')
        train(1)
    else:
        # 加载已有模型
        print('路径已有,加载模型ing')
        model = MypretrainModel()
        model.to(device)
        model.load_state_dict(torch.load(path))
        train(1)
    # todo     测试函数      句子长度也要改
    # input_ids = train_loader.dataset[15804][0].unsqueeze(0)
    # print(input_ids)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
```

```
"""
导入已训练好的masked语言模型并对有[MASK]的句子做预测
"""
from transformers import BertForMaskedLM

# 除了tokens 以外我们还需要辨别句子的segment ids
tokens_tensor = torch.tensor([ids])  # (1, seq_len)
segments_tensors = torch.zeros_like(tokens_tensor)  # (1, seq_len)
maskedLM_model = BertForMaskedLM.from_pretrained(PRETRAINED_MODEL_NAME)
clear_output()

# 使用masked LM 估计[MASK]位置所代表的实际标识符(token)
maskedLM_model.eval()
with torch.no_grad():
```

```
    outputs = maskedLM_model(tokens_tensor, segments_tensors)
    predictions = outputs[0]
    # (1, seq_len, num_hidden_units)
del maskedLM_model

# 将[MASK]位置的概率分布取前k个最有可能的标识符出来
masked_index = 5
k = 3
probs, indices = torch.topk(torch.softmax(predictions[0, masked_index], -1), k)
predicted_tokens = tokenizer.convert_ids_to_tokens(indices.tolist())

# 显示前k个最可能的字。一般取第一个作为预测值
print("輸入 tokens ：", tokens[:10], '...')
print('-' * 50)
for i, (t, p) in enumerate(zip(predicted_tokens, probs), 1):
    tokens[masked_index] = t
    print("Top {} ({:2}%)：{}".format(i, int(p.item() * 100), tokens[:10]),
'...')
```

2. wps 序号7

```
import os,sys
from torch.cuda.amp import GradScaler, autocast
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import Dataset,DataLoader
# todo 每次训练要改slide model保存路径
# 预处理  输出是 32连续字符 label是后面接着的字
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir('./data/'+ sub_folder):
            path1 = r'./data/'+ sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1:
                continue
            for i in range(len(data) // batch_size-1):
                x.append(data[i * batch_size:i* batch_size + slide])
                y.append(data[i * batch_size + slide])
                print(data[i * batch_size:i* batch_size + slide], data[i *
batch_size + slide])
            print(sub_folder + ' ' + path.replace('.txt', ' ') + '已经读取完毕')
    return x,y
# 读txt文件 输出无空格和换行的纯文本
def read_txt(path):
    with open(path, "r", encoding="utf-8") as f:
```

```
        content = f.readlines()
        if len(content) >= 2 :
            content = content[1].replace(u'\xa0', u'').replace(u'\u3000\u3000',
u'').replace('\n','').replace(' ','')
        else:
            content = ''
            return content
    return content

from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")

class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=slide+2, truncation=True
,padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target =
torch.tensor(int(tokenizer.convert_tokens_to_ids(self.y[index])))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size,
batch_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.num_directions = 1 # 单向LSTM
        self.batch_size = batch_size
        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
        # 分类的全链接层
        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input_seq):
        batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
        #  前向传播过程新生成的变量，需要传递到device中去
        # 如果是测试  batch_size = 1
        if input_seq.size(0) == 1:
            self.batch_size = 1
        h_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        c_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        # output(batch_size, seq_len, num_directions * hidden_size)
```

```
        output, _ = self.lstm(input_seq, (h_0, c_0)) # output(bs, seq_len, 
hidden_size)
        pred = self.linear(output)  # (bs, seq_len, output_size)
        pred = pred[:, -1, :]  # (bs, output_size)
        return pred

class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,512)
        self.fc1 = nn.Linear(512, len(tokenizer))
        self.dropout = nn.Dropout()
        self.lstm = LSTM(input_size=512, hidden_size=512, num_layers=1, 
output_size=len(tokenizer), batch_size=batch_size)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)

    def forward(self,x,attention_masks=None,token_type_ids=None):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['last_hidden_state']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        x = self.lstm(x)
        return x
# 训练文件
def train(epochs):
    for epoch in range(1, epochs + 1):
        model.train()
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in 
loop:
            total += 1
            input_ids, attention_masks, token_type_ids, target = 
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx + 
len(input_ids)))
```

```
        train_losses.append(running_loss / total)

        # 开始测试
#         model.eval()
#         test_loss = 0
#         correct = 0
#         with torch.no_grad():
#             for input_ids, attention_masks, token_type_ids, target in
test_loader:
#                 input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
#                     device), token_type_ids.to(device), target.to(device)
#                 output = model(input_ids, attention_masks, token_type_ids)
#                 test_loss += F.cross_entropy(output, target,
size_average=False).item()
#                 correct += accuracy(output, target)

#         test_loss /= len(test_loader.dataset)
#         test_losses.append(test_loss)
#         print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.3f}%)\n'.format(
#             test_loss, correct, len(test_loader.dataset),

#             100. * correct / len(test_loader.dataset)))
        # 测试结束
        # if correct / len(test_loader.dataset) > max_acc:
        #     max_acc = correct / len(test_loader.dataset)
        torch.save(model.state_dict(), path)
        best_model_wts = copy.deepcopy(model.state_dict())
def accuracy(predictions,labels):
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights

def test(sentence,pred_len):
    pred_sentence = ''
    for i in range(pred_len):
        inputs = tokenizer(sentence[-32:], max_length=slide + 2, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
        pred = model(input_ids.to(device))
        result = torch.max(pred, -1, keepdim=True)[1].item()
        print(tokenizer.decode(result))
        sentence = sentence + tokenizer.decode(result)
    print('续写完成后的内容是：', sentence)
if __name__ == "__main__":
    #todo 设置相关变量
    folder = '/root/data'
    slide = 32
    batch_size = 32
    savename = 'bert_lstm'
    path = f'/root/7{savename}_model.pth'
    data, label = make_dataset(folder, slide)
    print('输出处理完成 一共', len(data))
    #todo  定义数据集
```

```
    train_dataset = Mydataset(data, label)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    print(train_loader.dataset[2])
    #todo 设置模型相关信息
    model = MypretrainModel(is_lock=False)
    criterion = F.cross_entropy
    optimizer = optim.Adam(model.parameters(), lr=1e-5)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
    train_losses = []
    test_losses = []
    epochs = 1
    max_acc = 0
    print(path)
    if not os.path.exists(path):
        print('不存在路径')
        train(1)
    else:
        # 加载已有模型
        print('路径已有,加载模型ing')
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        # model.load_state_dict(torch.load(path))
        train(1)
    # todo     测试函数      句子长度也要改
    is_test = False
    pred_len = 100
    sentence = '"你个废物，不会是还对我余情未了，想要来这里找我，想跟我复合吧？我告诉你，想都
别想！我的心里现在只有李少，你就死了这条心吧！"林小曼冷冷的盯着楚尘，眼神中充满了鄙夷。'
    if is_test:
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        test(sentence,pred_len)
    # input_ids = train_loader.dataset[15804][0].unsqueeze(0)
    # print(input_ids)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
```

3. 序号8 5的进阶，再训练3个epoch

```
import os,sys
from torch.cuda.amp import GradScaler, autocast
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
```

```
from torch.utils.data import Dataset,DataLoader
# todo 每次训练要改slide model保存路径
# 预处理  输出是 32连续字符 label是后面接着的字
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir(folder+ '/' + sub_folder):
            path1 = folder+ '/' + sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1 :
                continue
            for i in range(len(data) // batch_size-1):
                x.append(data[i * batch_size:i* batch_size + slide])
                y.append(data[i * batch_size + slide])
                print(data[i * batch_size:i* batch_size + slide], data[i *
batch_size + slide])
    return x,y
# 读txt文件 输出无空格和换行的纯文本
def read_txt(path):
    with open(path, "r", encoding="utf-8") as f:
        content = f.readlines()
        if len(content) >= 2 :
            content = content[1].replace(u'\xa0', u'').replace(u'\u3000\u3000',
u'').replace('\n','').replace(' ','')
        else:
            content = ''
            return content
    return content

from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")

class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=slide+2, truncation=True
,padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target =
torch.tensor(int(tokenizer.convert_tokens_to_ids(self.y[index])))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

# # len(label) = 119

# class LSTM(nn.Module):
```

```
#     def __init__(self, input_size, hidden_size, num_layers, output_size, 
batch_size):
#         super().__init__()
#         self.input_size = input_size
#         self.hidden_size = hidden_size
#         self.num_layers = num_layers
#         self.output_size = output_size
#         self.num_directions = 1 # 单向LSTM
#         self.batch_size = batch_size
#         self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers, 
batch_first=True)
#         # 分类的全链接层
#         self.linear = nn.Linear(self.hidden_size, self.output_size)
#
#     def forward(self, input_seq):
#         batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
#         #  前向传播过程新生成的变量，需要传递到device中去
#         # 如果是测试  batch_size = 1
#         if input_seq.size(0) == 1:
#             self.batch_size = 1
#         h_0 = torch.randn(self.num_directions * self.num_layers, 
self.batch_size, self.hidden_size).to(device)
#         c_0 = torch.randn(self.num_directions * self.num_layers, 
self.batch_size, self.hidden_size).to(device)
#         # output(batch_size, seq_len, num_directions * hidden_size)
#         output, _ = self.lstm(input_seq, (h_0, c_0)) # output(bs, seq_len, 
hidden_size)
#         pred = self.linear(output)  # (bs, seq_len, output_size)
#         pred = pred[:, -1, :]  # (bs, output_size)
#         return pred

class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,512)
        self.fc1 = nn.Linear(512, len(tokenizer))
        self.dropout = nn.Dropout()
        # self.lstm = LSTM(input_size=512, hidden_size=256, num_layers=1, 
output_size=len(tokenizer), batch_size=batch_size)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)


    def forward(self,x,attention_masks=None,token_type_ids=None):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['last_hidden_state']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        x = self.fc1(x)
        pred = x[:, -1, :]
        return pred
```

```
# 训练文件
def train(epochs):
    for epoch in range(1, epochs + 1):
        model.train()
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in
loop:
            total += 1
            input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx +
len(input_ids)))

        train_losses.append(running_loss / total)

        # 开始测试
#         model.eval()
#         test_loss = 0
#         correct = 0
#         with torch.no_grad():
#             for input_ids, attention_masks, token_type_ids, target in
test_loader:
#                 input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
#                     device), token_type_ids.to(device), target.to(device)
#                 output = model(input_ids, attention_masks, token_type_ids)
#                 test_loss += F.cross_entropy(output, target,
size_average=False).item()
#                 correct += accuracy(output, target)

#         test_loss /= len(test_loader.dataset)
#         test_losses.append(test_loss)
#         print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.3f}%)\n'.format(
#             test_loss, correct, len(test_loader.dataset),

#             100. * correct / len(test_loader.dataset)))
        # 测试结束
        # if correct / len(test_loader.dataset) > max_acc:
        #     max_acc = correct / len(test_loader.dataset)
        torch.save(model.state_dict(), path)
        best_model_wts = copy.deepcopy(model.state_dict())
```

```
def accuracy(predictions,labels):
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights

def test(sentence,pred_len):
    present = []
    for i in range(pred_len):
        inputs = tokenizer(sentence[-32:], max_length=slide + 2, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
        pred = model(input_ids.to(device))
        print('topk顺序如下')
        k = 10
        topk = torch.topk(torch.softmax(pred,dim=-1),k=k,dim=-1,largest=True)
[1].cpu().numpy()
        # result 预测出来的编号
        result = 0
        print(topk[0,:])
        # 找到不重复的
        if len(present) == 30:  # 30个字内不能重复
            present.remove(present[0])
        for i in topk[0,:]:
            if i == 100 or i == 0:    # 跳过特殊字符
                continue
            elif present.count(i) > 0:   # 重复惩罚: 30个词内不能重复
                continue
            else:              #todo 束搜集
                result = i
                present.append(i)
                break
            # 非[UNK]
        print(result,tokenizer.decode(result))
        print(present)
        sentence = sentence + tokenizer.decode(result)
    print('续写完成后的内容是：', sentence)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
    # sentence = sentence + tokenizer.decode(result)
    # print('续写完成后的内容是：', sentence)

if __name__ == "__main__":
    #todo 设置相关变量
    folder = r'/root/data'
    slide = 32
    batch_size = 32
    savename = 'bert_lstm'
    path = f'/root/8{savename}_model.pth'
    data, label = make_dataset(folder, slide)
    print('输出处理完成 一共', len(data))
    #todo  定义数据集
    train_dataset = Mydataset(data, label)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    print(train_loader.dataset[2])
```

```
    #todo 设置模型相关信息
    model = MypretrainModel(is_lock=False)
    criterion = F.cross_entropy
    optimizer = optim.Adam(model.parameters(), lr=1e-5)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
    train_losses = []
    test_losses = []
    epochs = 1
    max_acc = 0
    print(path)

    pred_len = 100
    sentence = '今天天气晴朗，我和我的朋友们一起来到公园游玩，但是突然发现了一个人'
    # 选择进行  测试or训练
    is_test = False
    if is_test:
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        test(sentence, pred_len)
    else:
        if not os.path.exists(path):
            print('不存在路径')
            train(1)
        else:
            # 加载已有模型
            print('路径已有,加载模型ing')
            model = MypretrainModel()
            model.load_state_dict(torch.load(path))
            model.to(device)
            # model.load_state_dict(torch.load(path))
            train(1)
    # todo     测试函数     句子长度也要改


    # input_ids = train_loader.dataset[15804][0].unsqueeze(0)
    # print(input_ids)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
```

4. 序号9 7的进阶，再训练3个epoch

```
import os,sys
from torch.cuda.amp import GradScaler, autocast
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import Dataset,DataLoader
```

```python
# todo 每次训练要改slide model保存路径
# 预处理  输出是 32连续字符 label是后面接着的字
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir(folder + '/' + sub_folder):
            path1 = folder + '/' + sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1:
                continue
            for i in range(len(data) // batch_size-1):
                x.append(data[i * batch_size:i* batch_size + slide])
                y.append(data[i * batch_size + slide])
                print(data[i * batch_size:i* batch_size + slide], data[i *
batch_size + slide])
            print(sub_folder + ' ' + path.replace('.txt', ' ') + '已经读取完毕')
    return x,y
# 读txt文件 输出无空格和换行的纯文本
def read_txt(path):
    with open(path, "r", encoding="utf-8") as f:
        content = f.readlines()
        if len(content) >= 2 :
            content = content[1].replace(u'\xa0', u'').replace(u'\u3000\u3000',
u'').replace('\n','').replace(' ','')
        else:
            content = ''
            return content
    return content

from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")

class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=slide+2, truncation=True
,padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target =
torch.tensor(int(tokenizer.convert_tokens_to_ids(self.y[index])))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size,
batch_size):
        super().__init__()
```

```
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.num_directions = 1 # 单向LSTM
        self.batch_size = batch_size
        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
        # 分类的全链接层
        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input_seq):
        batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
        #  前向传播过程新生成的变量，需要传递到device中去
        # 如果是测试  batch_size = 1
        if input_seq.size(0) == 1:
            self.batch_size = 1
        h_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        c_0 = torch.randn(self.num_directions * self.num_layers, self.batch_size,
self.hidden_size).to(device)
        # output(batch_size, seq_len, num_directions * hidden_size)
        output, _ = self.lstm(input_seq, (h_0, c_0)) # output(bs, seq_len,
hidden_size)
        pred = self.linear(output)  # (bs, seq_len, output_size)
        pred = pred[:, -1, :]  # (bs, output_size)
        return pred

class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,512)
        self.fc1 = nn.Linear(512, len(tokenizer))
        self.dropout = nn.Dropout()
        self.lstm = LSTM(input_size=512, hidden_size=512, num_layers=1,
output_size=len(tokenizer), batch_size=batch_size)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)

    def forward(self,x,attention_masks=None,token_type_ids=None):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['last_hidden_state']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        x = self.lstm(x)
        return x
# 训练文件
def train(epochs):
    for epoch in range(1, epochs + 1):
        model.train()
```

```
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in
loop:
            total += 1
            input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx +
len(input_ids)))

        train_losses.append(running_loss / total)

        # 开始测试
#         model.eval()
#         test_loss = 0
#         correct = 0
#         with torch.no_grad():
#             for input_ids, attention_masks, token_type_ids, target in
test_loader:
#                 input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
#                     device), token_type_ids.to(device), target.to(device)
#                 output = model(input_ids, attention_masks, token_type_ids)
#                 test_loss += F.cross_entropy(output, target,
size_average=False).item()
#                 correct += accuracy(output, target)

#         test_loss /= len(test_loader.dataset)
#         test_losses.append(test_loss)
#         print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.3f}%)\n'.format(
#             test_loss, correct, len(test_loader.dataset),

#             100. * correct / len(test_loader.dataset)))
        # 测试结束
        # if correct / len(test_loader.dataset) > max_acc:
        #     max_acc = correct / len(test_loader.dataset)
        torch.save(model.state_dict(), path)
        best_model_wts = copy.deepcopy(model.state_dict())
def accuracy(predictions,labels):
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights
```

```
def test(sentence,pred_len):
    pred_sentence = ''
    for i in range(pred_len):
        inputs = tokenizer(sentence[-32:], max_length=slide + 2, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
        pred = model(input_ids.to(device))
        result = torch.max(pred, -1, keepdim=True)[1].item()
        print(tokenizer.decode(result))
        sentence = sentence + tokenizer.decode(result)
    print('续写完成后的内容是：', sentence)
if __name__ == "__main__":
    #todo 设置相关变量
    folder = '/root/data'
    slide = 32
    batch_size = 32
    savename = 'bert_lstm'
    path = f'/root/9{savename}_model.pth'
    data, label = make_dataset(folder, slide)
    print('输出处理完成 一共', len(data))
    #todo  定义数据集

    train_dataset = Mydataset(data, label)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    print(train_loader.dataset[2])
    #todo 设置模型相关信息
    model = MypretrainModel(is_lock=False)
    criterion = F.cross_entropy
    optimizer = optim.Adam(model.parameters(), lr=1e-5)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
    train_losses = []
    test_losses = []
    epochs = 1
    max_acc = 0
    print(path)
    if not os.path.exists(path):
        print('不存在路径')
        model = MypretrainModel()
        model.load_state_dict(torch.load(f'/root/7{savename}_model.pth'))
        model.to(device)
        train(3)
    else:
        # 加载已有模型
        print('路径已有,加载模型ing')
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        # model.load_state_dict(torch.load(path))
        train(1)
    # todo    测试函数    句子长度也要改
    is_test = False
    pred_len = 100
```

```
    sentence = '“你个废物，不会是还对我余情未了，想要来这里找我，想跟我复合吧？我告诉你，想都别想！我的心里现在只有李少，你就死了这条心吧！”林小曼冷冷的盯着楚尘，眼神中充满了鄙夷。'
    if is_test:
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        test(sentence,pred_len)
    # input_ids = train_loader.dataset[15804][0].unsqueeze(0)
    # print(input_ids)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
```

5. 序号11 从头训练 bert + fc

```
import os,sys
from torch.cuda.amp import GradScaler, autocast
import torch.nn as nn
import torch
import copy
import json
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import Dataset,DataLoader
# todo 每次训练要改slide model保存路径
# 预处理  输出是 32连续字符 label是后面接着的字
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir(folder + '/' + sub_folder):
            path1 = folder + '/' + sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1:
                continue
            for i in range(len(data) // slide-1):
                x.append(data[i * slide:i* slide + slide])
                y.append(data[i * slide + slide])
                print(data[i * slide:i* slide + slide], data[i * slide + slide])
            print(sub_folder + ' ' + path.replace('.txt', ' ') + '已经读取完毕')
    return x,y
# 读txt文件 输出无空格和换行的纯文本
def read_txt(path):
    with open(path, "r", encoding="utf-8") as f:
        content = f.readlines()
        if len(content) >= 2 :
            content = content[1].replace(u'\xa0', u'').replace(u'\u3000\u3000', u'').replace('\n','').replace(' ','')
        else:
            content = ''
            return content
```

```python
    return content

from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained("bert-base-chinese")
bert = BertModel.from_pretrained("bert-base-chinese")

class Mydataset(Dataset):
    def __init__(self,data,label):
        self.x = data
        self.y = label
    def __getitem__(self, index):
        inputs = tokenizer(self.x[index], max_length=slide+2, truncation=True
,padding='max_length')
        input_ids = torch.tensor(inputs['input_ids'])
        attention_masks = torch.tensor(inputs['attention_mask'])
        token_type_ids = torch.tensor(inputs['token_type_ids'])
        target =
torch.tensor(int(tokenizer.convert_tokens_to_ids(self.y[index])))
        return input_ids,attention_masks,token_type_ids,target
    def __len__(self):
        return  len(self.x)

# # len(label) = 119

# class LSTM(nn.Module):
#     def __init__(self, input_size, hidden_size, num_layers, output_size,
batch_size):
#         super().__init__()
#         self.input_size = input_size
#         self.hidden_size = hidden_size
#         self.num_layers = num_layers
#         self.output_size = output_size
#         self.num_directions = 1 # 单向LSTM
#         self.batch_size = batch_size
#         self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
#         # 分类的全链接层
#         self.linear = nn.Linear(self.hidden_size, self.output_size)
#
#     def forward(self, input_seq):
#         batch_size, seq_len = input_seq.shape[0], input_seq.shape[1]
#         #  前向传播过程新生成的变量，需要传递到device中去
#         # 如果是测试  batch_size = 1
#         if input_seq.size(0) == 1:
#             self.batch_size = 1
#         h_0 = torch.randn(self.num_directions * self.num_layers,
self.batch_size, self.hidden_size).to(device)
#         c_0 = torch.randn(self.num_directions * self.num_layers,
self.batch_size, self.hidden_size).to(device)
#         # output(batch_size, seq_len, num_directions * hidden_size)
#         output, _ = self.lstm(input_seq, (h_0, c_0)) # output(bs, seq_len,
hidden_size)
#         pred = self.linear(output)  # (bs, seq_len, output_size)
#         pred = pred[:, -1, :]  # (bs, output_size)
#         return pred
```

```python
class MypretrainModel(nn.Module):
    def __init__(self,is_lock=False):
        super(MypretrainModel,self).__init__()
        self.bert_pretrained = bert
        self.fc = nn.Linear(768,512)
        self.fc1 = nn.Linear(512, len(tokenizer))
        self.dropout = nn.Dropout()
        # self.lstm = LSTM(input_size=512, hidden_size=256, num_layers=1,
output_size=len(tokenizer), batch_size=batch_size)
        if is_lock:
            # 加载并冻结bert模型参数
            for name, param in self.bert_pretrained.named_parameters():
                if name.startswith('pooler'):
                    continue
                else:
                    param.requires_grad_(False)

    def forward(self,x,attention_masks=None,token_type_ids=None):
        x = self.bert_pretrained(x,attention_masks,token_type_ids)
['last_hidden_state']
        x = F.relu(self.fc(x))
        x = self.dropout(x)
        x = self.fc1(x)
        pred = x[:, -1, :]
        return pred
# 训练文件
def train(epochs):
    for epoch in range(1, epochs + 1):
        model.train()
        loop = tqdm(enumerate(train_loader), total=len(train_loader))
        running_loss = 0.0
        total = 0
        right = 0
        for batch_idx, (input_ids, attention_masks, token_type_ids, target) in
loop:
            total += 1
            input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
                device), token_type_ids.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input_ids, attention_masks, token_type_ids)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            right += accuracy(output, target)

            loop.set_description(f'Epoch [{epoch}/{epochs}]')
            loop.set_postfix(loss=running_loss / (batch_idx + 1),
                             acc=float(right) / float(batch_size * batch_idx +
len(input_ids)))

        train_losses.append(running_loss / total)

        # 开始测试
#         model.eval()
```

```
#         test_loss = 0
#         correct = 0
#         with torch.no_grad():
#             for input_ids, attention_masks, token_type_ids, target in
test_loader:
#                 input_ids, attention_masks, token_type_ids, target =
input_ids.to(device), attention_masks.to(
#                     device), token_type_ids.to(device), target.to(device)
#                 output = model(input_ids, attention_masks, token_type_ids)
#                 test_loss += F.cross_entropy(output, target,
size_average=False).item()
#                 correct += accuracy(output, target)

#         test_loss /= len(test_loader.dataset)
#         test_losses.append(test_loss)
#         print('Test set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.3f}%)\n'.format(
#             test_loss, correct, len(test_loader.dataset),

#             100. * correct / len(test_loader.dataset)))
        # 测试结束
        # if correct / len(test_loader.dataset) > max_acc:
        #     max_acc = correct / len(test_loader.dataset)
        torch.save(model.state_dict(), path)
        best_model_wts = copy.deepcopy(model.state_dict())
def accuracy(predictions,labels):
    pred = torch.max(predictions.data,1,keepdim=True)[1]
    rights = pred.eq(labels.data.view_as(pred)).sum()
    return rights

def test(sentence,pred_len):
    present = []
    print('输入句子: ',sentence)
    for i in range(pred_len):
        inputs = tokenizer(sentence[-32:], max_length=slide + 2, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
        pred = model(input_ids.to(device))
        # print('topk顺序如下')
        k = 10
        topk = torch.topk(torch.softmax(pred,dim=-1),k=k,dim=-1,largest=True)
[1].cpu().numpy()
        # result 预测出来的编号
        result = 0
        # print(topk[0,:])
        # 找到不重复的
        if len(present) == 30:  # 30个字内不能重复
            present.remove(present[0])
        for i in topk[0,:]:
            if i == 100 or i == 0:     # 跳过特殊字符
                continue
            elif present.count(i) > 0:    # 重复惩罚: 30个词内不能重复
                continue
            else:              #todo 束搜集
                result = i
                present.append(i)
```

```
                break
            # 非[UNK]
        # print(result,tokenizer.decode(result))
        # print(present)
        sentence = sentence + tokenizer.decode(result)
    print('续写完成后的内容是：', sentence)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
    # sentence = sentence + tokenizer.decode(result)
    # print('续写完成后的内容是：', sentence)

if __name__ == "__main__":
    #todo 设置相关变量
    folder = r'/root/data'
    slide = 32
    batch_size = 64
    savename = 'bert_lstm'
    path = f'/root/11{savename}_model.pth'
    data, label = make_dataset(folder, slide)
    print('输出处理完成 一共', len(data))
    #todo  定义数据集
    train_dataset = Mydataset(data, label)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    print(train_loader.dataset[2])
    #todo 设置模型相关信息
    model = MypretrainModel(is_lock=False)
    criterion = F.cross_entropy
    optimizer = optim.Adam(model.parameters(), lr=2e-5)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model.to(device)
    train_losses = []
    test_losses = []
    epochs = 5
    max_acc = 0
    print(path)

    pred_len = 500
    sentence = '“你个废物，不会是还对我余情未了，想要来这里找我，想跟我复合吧？我告诉'
    # 选择进行  测试or训练
    is_test = False
    if is_test:
        model = MypretrainModel()
        model.load_state_dict(torch.load(path))
        model.to(device)
        test(sentence, pred_len)
    else:
        if not os.path.exists(path):
            print('不存在路径')
            train(epochs)
        else:
            # 加载已有模型
            print('路径已有,加载模型ing')
            model = MypretrainModel()
            model.load_state_dict(torch.load(path))
```

```
            model.to(device)
            # model.load_state_dict(torch.load(path))
            train(epochs)
    # todo      测试函数      句子长度也要改


    # input_ids = train_loader.dataset[15804][0].unsqueeze(0)
    # print(input_ids)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
```

7.8 两个处理数据集的方法

间隔16 lenth = int(batch_size / 2)

```
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir(folder + '/' + sub_folder):
            path1 = folder + '/' + sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1:
                continue
            lenth = int(batch_size / 2)
            for i in range(len(data) // lenth - 32):
                x.append(data[i * lenth:i* lenth + slide])
                y.append(data[i * lenth + slide])
                print(data[i * lenth:i* lenth + slide], data[i * lenth + slide])
            print(sub_folder + ' ' + path.replace('.txt', ' ') + '已经读取完毕')
    return x,y
```

间隔32

```
def make_dataset(folder, slide):
    dirs = os.listdir(folder)
    x = []
    y = []
    for sub_folder in dirs:
        for path in os.listdir(folder + '/' + sub_folder):
            path1 = folder + '/' + sub_folder + '/' + path
            data = read_txt(path1)
            #print(data)
            if len(data) < slide + 1:
                continue
            for i in range(len(data) // slide-1):
                x.append(data[i * slide:i* slide + slide])
                y.append(data[i * slide + slide])
                print(data[i * slide:i* slide + slide], data[i * slide + slide])
            print(sub_folder + ' ' + path.replace('.txt', ' ') + '已经读取完毕')
```

```
    return x,y
```

7.9 test函数

```
def test(sentence,pred_len):
    present = []
    print('输入句子: ',sentence)
    for i in range(pred_len):
        inputs = tokenizer(sentence[-32:], max_length=slide + 2, truncation=True,
padding='max_length')
        input_ids = torch.tensor(inputs['input_ids']).unsqueeze(0)
        pred = model(input_ids.to(device))
        # print('topk顺序如下')
        k = 10
        topk = torch.topk(torch.softmax(pred,dim=-1),k=k,dim=-1,largest=True)
[1].cpu().numpy()
        # result 预测出来的编号
        result = 0
        # print(topk[0,:])
        # 找到不重复的
        if len(present) == 30:  # 30个字内不能重复
            present.remove(present[0])
        for i in topk[0,:]:
            if i == 100 or i == 0:    # 跳过特殊字符
                continue
            elif present.count(i) > 0:   # 重复惩罚: 30个词内不能重复
                continue
            else:                 #todo 束搜集
                result = i
                present.append(i)
                break
            # 非[UNK]
        # print(result,tokenizer.decode(result))
        # print(present)
        sentence = sentence + tokenizer.decode(result)
    print('续写完成后的内容是: ', sentence)
    # pred = model(input_ids.to(device))
    # result = torch.max(pred, -1, keepdim=True)[1].item()
    # print(tokenizer.decode(result))
    # sentence = sentence + tokenizer.decode(result)
    # print('续写完成后的内容是: ', sentence)
```

7.10 测试的效果

wps: 11   /root/11bert_lstm_model.pth

输入句子: 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛

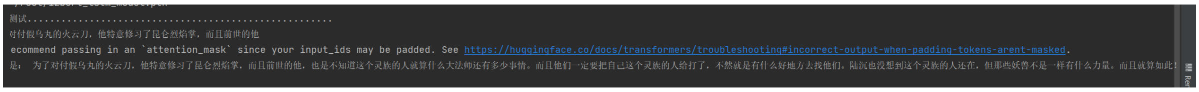
续写完成后的内容是： 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛。这根本不是手术没了，而个人生活在自己的身体之中！然后就算他得到一些伤势和信心了，那也不会被人在眼里。这种情况下她就可以让自己的身体有些好看，但是不知道要怎么办？而且他还没想到这个小女孩的身体竟然有什大，就是自己一点伤也不在。因为他没想到这个女人的力量竟然如此大，所以要让自己有一种可能。龙瀚说道：.你们这些人已经来了吧？不过，我没想到的是他现在还有那么多名长老。这些人都好像被你们给打碎了，我就不想让他在此说话吧！一个大帝运还是可以把那些真王给打

出来的。而且，陆沉也不知道这里有什么地方？但是一个人进去之后就被他的身体打开，那些头色白衣子女看着自己和楚云这个人都不知道她是为了什么。但他没有想到，那些女的竟然在自己面前被一个人打飞出来了！这是她真正最不好奇，他们也没有想到楚尘的实力还会如此强大。所以龙瀚就是这么看着云天河，不知道陆沉要说什样了！而且在他的心中便有一个小大灵气脉。所以，那些仙石不会被陆沉给打败了！而且在此时的战场上一个人都有自己身体之力，他不是陆沉和那位冥族真王在中洲的这些大家伙吗？但妖河守护者也没有出来，他就不知道陆沉在哪里了。那个人是灵族的大罗金仙境弟子！这一次，他不知道陆沉

输入句子：“居然有人在自己不知情的时候靠近了，还好没有被人偷袭。”凌风庆幸自己没有被偷袭

续写完成后的内容是：“居然有人在自己不知情的时候靠近了，还好没有被人偷袭。”凌风庆幸自己没有被偷袭，但是她没有想到这个小子竟然还能一直在身上的样皮。他不知道楚云为什么要跟她来，但是这种时候自己也没有想到了过去。他不知道楚云的身份？她可以看出龙瀚这个女人是什么意思了，但很快便开口道：「你不要让我想到！」云天河一脸的怒火。这时，楚尘看向了那人说道：「你是什么女生？」云天河一脸不好意思的问着。他看向楚尘，眼中闪过几分惊喜：「我们这里是有一个好地方的！」云初说道："你看着那些人，不知晓我们这里是什么时候来了吗？」云初笑道：「你想要怎样。，不过那个人的手里还有一些东西呢？」云初笑道：「我们这么多年来，都是不好的人。你想要什对?"」"

输入句子：一连串提示声，让凌风有些愕然，还好他还记得眼下最重要的事情是赶紧恢复体力赶到小
续写完成后的内容是：一连串提示声，让凌风有些愕然，还好他还记得眼下最重要的事情是赶紧恢复体力赶到小玉的面前，这是一个不错。他们也没有想到她还在自己身上！而且就算如此的话，那是一个不过十万年前。这些时候他也没有想到她还在自己身上！而且，陆沉的战力不是一个人都能够让他成为真王。但这种时候就有了那么多强大的战力，也不知道陆沉是什根样子？所以他们还没出来吗。这个时候！一条金光脉从大地之中响起，而陆沉的身体也被四周、天空和火焰传来。这一次了！那个大蛟已经在陆沉的身上打出过五十万斤灵气，也不知道他们是什么



```
测试......................................................
付付假乌丸的火云刀，他特意修习了昆仑烈焰掌，而且前世的他
ecommend passing in an `attention_mask` since your input_ids may be padded. See https://huggingface.co/docs/transformers/troubleshooting#incorrect-output-when-padding-tokens-arent-masked.
是： 为了对付假乌丸的火云刀，他特意修习了昆仑烈焰掌，而且前世的他，也是不知道这个灵族的人就算什么大法师还有多少事情。而且他们一定要把自己这个灵族的人给打了，不然就是有什么好地方去找他们。陆沉也没想到这个灵族的人还在，但那些妖兽不是一样有什么力量。而且就算如此！
```

```
“居然有人在自己不知情的时候靠近了，还好没有被人偷袭。”凌风庆幸自己没有被偷袭，但是她没有想到这个小子竟然还能一直在身上的样皮。他不知道楚云为什么要跟她来，但是这种时候自己也没有想到了过去。他不知道楚云的身份？她可以看出龙瀚这个女人是什么意思了，但很快便开口
```

```
Run:  bert_lstm_test ×   bert_lstm ×
/root/.virtualenvs/python_files/bin/python /tmp/pycharm_project_866/west2online_last_test/bert_lstm_test.py
/root/11bert_lstm_model.pth
模型权重路径是： /root/11bert_lstm_model.pth
正在载入模型准备测试。。。。。。。。。。。。。。。。。。。。。。
输入句子： 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛
We strongly recommend passing in an `attention_mask` since your input_ids may be padded. See https://huggingface.co/docs/transformers/troubleshooting#incorrect-output-when-padding-tokens-arent-masked.
续写完成后的内容是： 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛。这根本不是他在自己面前的好处，而如果想要让她有一个大小之间无法看得出这种情况。楚云的眼睛中，却是闪过了几分不好和奇怪之色：他要知道这个人竟然没有一种会让我难以置信的

Process finished with exit code 0
```

```
模型权重路径是： /root/12bert_lstm_model.pth
正在载入模型准备测试.......................................................
输入句子： 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛
续写完成后的内容是： 甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛，不知道有多少人都在打击他。这是一个好的对手！而且就算如此和无奈，那么自己也不会有什.我们还是想要一起去看说完的。这个时候就在他把自己打开，眼中露出了不知道之色：我是
```

输入句子：甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛

续写完成后的内容是：甚至于预料之中由韧带的损伤引起的疼痛和肌肉酸痛也丝没有出现，整个人精力充沛。这根本不是手术没了，而个人生活在自己的身体之中！然后就算他得到一些伤势和信心了，那也不会被人在眼里。这种情况下她就可以让自己的身体有些好看，但是不知道要怎么办？而且他还没想到这个小女孩的身体竟然有什大，就是自己一点伤也不在。因为他没想到这个女人的力量竟然如此大，所以要让自己有一种可能。龙瀚说道：．你们这些人已经来了吧？不过，我没想到的是他现在还有那么多名长老。这些人都好像被你们给打碎了，我就不想让他在此说话吧！一个大帝运还是可以把那些真王给打出来的。而且，陆沉也不知道这里有什么地方？但是一个人进去之后就被他的身体打开，那些头色白衣子女看着自己和楚云这个人都不知道她是为了什么。但他没有想到，那些女的竟然在自己面前被一个人打飞出来了！这是她真正最不好奇，他们也没有想到楚尘的实力还会如此强大。所以龙瀚就是这么看着云天河，不知道陆沉要说什样了！而且在他的心中便有一个小大灵气脉。所以，那些仙石不会被陆沉给打败了！而且在此时的战场上一个人都有自己身体之力，他不是陆沉和那位冥族真王在中洲的这些大家伙吗？但妖河守护者也没有出来，他就不知道陆沉在哪里了。那个人是灵族的大罗金仙境弟子！这一次，他不知道陆沉

输入句子： “居然有人在自己不知情的时候靠近了，还好没有被人偷袭。”凌风庆幸自己没有被偷袭

续写完成后的内容是：“居然有人在自己不知情的时候靠近了，还好没有被人偷袭。”凌风庆幸自己没有被偷袭，但是她没有想到这个小子竟然还能一直在身上的样皮。他不知道楚云为什么要跟她来，但是这种时候自己也没有想到了过去。他不知道楚云的身份？她可以看出龙瀚这个女人是什么意思了，但很快便开口道：「你不要让我想到！」云天河一脸的怒火。这时，楚尘看向了那人说道：「你是什么女生？」云天河一脸不好意思的问着。他看向楚尘，眼中闪过几分惊喜：「我们这里是有一个好地方的！」云初说道："你看着那些人，不知晓我们这里是什么时候来了吗？」云初笑道：「你想要怎样。，不过那个人的手里还有一些东西呢？」云初笑道：「我们这么多年来，都是不好的人。你想要什对?“」"

输入句子： 一连串提示声，让凌风有些愕然，还好他还记得眼下最重要的事情是赶紧恢复体力赶到小
续写完成后的内容是：一连串提示声，让凌风有些愕然，还好他还记得眼下最重要的事情是赶紧恢复体力赶到小玉的面前，这是一个不错。他们也没有想到她还在自己身上！而且就算如此的话，那是一个不过十万年前。这些时候他也没有想到她还在自己身上！而且，陆沉的战力不是一个人都能够让他成为真王。但这种时候就有了那么多强大的战力，也不知道陆沉是什根样子？所以他们还没出来吗。这个时候！一条金光脉从大地之中响起，而陆沉的身体也被四周、天空和火焰传来。这一次了！那个大蛟已经在陆沉的身上打出过五十万斤灵气，也不知道他们是什么