**NLP期末实践报告**

**姓 名** 陈静

**学 号**  2272072

|  |
| --- |
| 实验时间： 1 月 14 日 星期 六 |
| 实验目的：运用Transformer模型实现机器翻译并识别实体 |
| 实验环境：  Python 3.7.2  Torch 1.13.1  Pycharm 2019.3.3  Git |
| 实验理论：  在机器翻译任务中使用Transformer 模型，将一种语言的一个句子作为输入，然后将其翻译成另一种语言的一个句子作为输出。Transformer 本质上是一个 Encoder-Decoder 架构。因此中间部分的 Transformer 可以分为两个部分：编码组件和解码组件。其中，编码组件由多层编码器（Encoder）组成。解码组件也是由相同层数的解码器（Decoder）组成.每个编码器由两个子层组成：Self-Attention 层（自注意力层）和 Position-wise Feed Forward Network（前馈网络，缩写为 FFN）每个编码器的结构都是相同的，但是它们使用不同的权重参数。解码器也有编码器中这两层，但是它们之间还有一个注意力层（即 Encoder-Decoder Attention），其用来帮忙解码器关注输入句子的相关部分（类似于 seq2seq 模型中的注意力）。通过解码器解码后再进行线性变换和归一化输出预测的结果。  IMG_256  [实体识别](https://so.csdn.net/so/search?q=%E5%91%BD%E5%90%8D%E5%AE%9E%E4%BD%93%E8%AF%86%E5%88%AB&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42464569/article/details/_blank)的任务是对每一个token都进行分类。比如，识别这个token是不是一个人名、组织名或地名。命名实体识别的一个数据集是CoNLL-2003，这个数据集完全契合这个任务。 |
| 实验步骤：   1. 环境搭建 2. 下载python3.7.2 3. 安装Anaconda3 4. 下载torch1.13.1 5. 安装Pycharm2019.3.3   PyCharm是一种[Python](https://baike.baidu.com/item/Python/407313?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/PyCharm/_blank) [IDE](https://baike.baidu.com/item/IDE/8232086?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/PyCharm/_blank)（Integrated Development Environment，集成开发环境）   1. 安装git 2. 代码设计与实现   (1)代码设计部分  i#  # 数据构建  import math  import torch  import numpy as np  import torch.nn as nn  import torch.optim as optim  import torch.utils.data as Data  from transformers import AutoModelForTokenClassification, AutoTokenizer  device = 'cpu'  # device = 'cuda'  # transformer.py epochs  epochs = 100  # 这里手动输入了两对中文→英语的句子  # S: 显示解码输入开始的符号  # E: 显示解码输出开始的符号  # P: 如果当前批处理数据大小小于时间步长，将填充空白序列的符号  label\_list = [  "O", # Outside of a named entity  "B-PER", # Beginning of a person's name right after another person's name  "I-PER", # Person's name  "B-ORG", # Beginning of an organisation right after another organisation  "I-ORG", # Organisation  "B-LOC", # Beginning of a location right after another location  "I-LOC" # Location  ]  # 训练集  sentences = [  # enc\_input dec\_input dec\_output  ['亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司 P', 'S Amazon is the largest online e-commerce company in the US . ', 'Amazon is the largest online e-commerce company in the US . E'],  ['亚 马 逊 位 于 华 盛 顿 州 的 西 雅 图 P P P P P P P P P', 'S Amazon is located in Seattle , Washington . . . . ', 'Amazon is located in Seattle , Washington . . . . E']  ]  # 测试集（  # 输入："亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司"  # 输出："Amazon is the largest online e-commerce company in the US."  # 分别建立中文和英文词库  src\_vocab = {'P': 0, '亚': 1, '马': 2, '逊': 3, '公': 4, '司': 5, '是': 6, '美': 7, '国': 8, '最': 9,  '大': 10, '的': 11 ,'一': 12, '家': 13, '网': 14, '络': 15, '电': 16, '子': 17, '商': 18, '务': 19,  '位': 20, '于': 21, '华': 22, '盛': 23, '顿': 24, '州': 25, '西': 26, '雅': 27, '图': 28 }  src\_idx2word = {i: w for i, w in enumerate(src\_vocab)}  src\_vocab\_size = len(src\_vocab)  tgt\_vocab = {'P': 0, 'Amazon': 1, 'is': 2, 'the': 3, 'largest': 4, 'online': 5, 'e-commerce': 6,  'company': 7, 'in': 8, 'US': 9, 'located': 10, 'Seattle': 11, 'Washington': 12,  'S': 13, 'E':14, '.':15, ',':16 }  idx2word = {i: w for i, w in enumerate(tgt\_vocab)}  tgt\_vocab\_size = len(tgt\_vocab)  src\_len = 22 # enc\_input max sequence length  tgt\_len = 14 # dec\_input max sequence length  # 超参数  d\_model = 512 # Embedding Size（token embedding和position编码的维度）  d\_ff = 2048 # FeedForward dimension (两次线性层中的隐藏层 512->2048->512，线性层是用来做特征提取的），当然最后会再接一个projection层  d\_k = d\_v = 64 # dimension of K(=Q), V（Q和K的维度需要相同，这里为了方便让K=V）  n\_layers = 6 # number of Encoder of Decoder Layer（Block的个数）  n\_heads = 8 # number of heads in Multi-Head Attention（有几套头）  # 数据构建  def make\_data(sentences):  """把单词序列转换为数字序列"""  enc\_inputs, dec\_inputs, dec\_outputs = [], [], []  for i in range(len(sentences)):  enc\_input = [[src\_vocab[n] for n in sentences[i][0].split()]] # [[1, 2, 3, 4, 0], [1, 2, 3, 5, 0]]  dec\_input = [[tgt\_vocab[n] for n in sentences[i][1].split()]] # [[6, 1, 2, 3, 4, 8], [6, 1, 2, 3, 5, 8]]  dec\_output = [[tgt\_vocab[n] for n in sentences[i][2].split()]] # [[1, 2, 3, 4, 8, 7], [1, 2, 3, 5, 8, 7]]  enc\_inputs.extend(enc\_input)  dec\_inputs.extend(dec\_input)  dec\_outputs.extend(dec\_output)  return torch.LongTensor(enc\_inputs), torch.LongTensor(dec\_inputs), torch.LongTensor(dec\_outputs)  enc\_inputs, dec\_inputs, dec\_outputs = make\_data(sentences)  class MyDataSet(Data.Dataset):  """自定义DataLoader"""  def \_\_init\_\_(self, enc\_inputs, dec\_inputs, dec\_outputs):  super(MyDataSet, self).\_\_init\_\_()  self.enc\_inputs = enc\_inputs  self.dec\_inputs = dec\_inputs  self.dec\_outputs = dec\_outputs  def \_\_len\_\_(self):  return self.enc\_inputs.shape[0]  def \_\_getitem\_\_(self, idx):  return self.enc\_inputs[idx], self.dec\_inputs[idx], self.dec\_outputs[idx]  loader = Data.DataLoader(MyDataSet(enc\_inputs, dec\_inputs, dec\_outputs), 2, True)  # Transformer模型  class PositionalEncoding(nn.Module):  def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):  super(PositionalEncoding, self).\_\_init\_\_()  self.dropout = nn.Dropout(p=dropout)  pe = torch.zeros(max\_len, d\_model)  position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)  div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))  pe[:, 0::2] = torch.sin(position \* div\_term)  pe[:, 1::2] = torch.cos(position \* div\_term)  pe = pe.unsqueeze(0).transpose(0, 1)  self.register\_buffer('pe', pe)  def forward(self, x):  """  x: [seq\_len, batch\_size, d\_model]  """  x = x + self.pe[:x.size(0), :]  return self.dropout(x)  def get\_attn\_pad\_mask(seq\_q, seq\_k):  # pad mask的作用：在对value向量加权平均的时候，可以让pad对应的alpha\_ij=0，这样注意力就不会考虑到pad向量  batch\_size, len\_q = seq\_q.size() # 这个seq\_q只是用来expand维度的  batch\_size, len\_k = seq\_k.size()  # eq(zero) is PAD token  # 例如:seq\_k = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 4, 5, 0]]  pad\_attn\_mask = seq\_k.data.eq(0).unsqueeze(1) # [batch\_size, 1, len\_k], True is masked  return pad\_attn\_mask.expand(batch\_size, len\_q, len\_k) # [batch\_size, len\_q, len\_k] 构成一个立方体(batch\_size个这样的矩阵)  def get\_attn\_subsequence\_mask(seq):  attn\_shape = [seq.size(0), seq.size(1), seq.size(1)]  # attn\_shape: [batch\_size, tgt\_len, tgt\_len]  subsequence\_mask = np.triu(np.ones(attn\_shape), k=1) # 生成一个上三角矩阵  subsequence\_mask = torch.from\_numpy(subsequence\_mask).byte()  return subsequence\_mask # [batch\_size, tgt\_len, tgt\_len]  class ScaledDotProductAttention(nn.Module):  def \_\_init\_\_(self):  super(ScaledDotProductAttention, self).\_\_init\_\_()  def forward(self, Q, K, V, attn\_mask):  scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d\_k) # scores : [batch\_size, n\_heads, len\_q, len\_k]  # mask矩阵填充scores（用-1e9填充scores中与attn\_mask中值为1位置相对应的元素）  scores.masked\_fill\_(attn\_mask, -1e9)  attn = nn.Softmax(dim=-1)(scores) # 对最后一个维度(v)做softmax  # scores : [batch\_size, n\_heads, len\_q, len\_k] \* V: [batch\_size, n\_heads, len\_v(=len\_k), d\_v]  context = torch.matmul(attn, V) # context: [batch\_size, n\_heads, len\_q, d\_v]  # context：[[z1,z2,...],[...]]向量, attn注意力稀疏矩阵（用于可视化的）  return context, attn  class MultiHeadAttention(nn.Module):  def \_\_init\_\_(self):  super(MultiHeadAttention, self).\_\_init\_\_()  self.W\_Q = nn.Linear(d\_model, d\_k \* n\_heads, bias=False)  self.W\_K = nn.Linear(d\_model, d\_k \* n\_heads, bias=False)  self.W\_V = nn.Linear(d\_model, d\_v \* n\_heads, bias=False)  # 这个全连接层可以保证多头attention的输出仍然是seq\_len x d\_model  self.fc = nn.Linear(n\_heads \* d\_v, d\_model, bias=False)  def forward(self, input\_Q, input\_K, input\_V, attn\_mask):  residual, batch\_size = input\_Q, input\_Q.size(0)  Q = self.W\_Q(input\_Q).view(batch\_size, -1, n\_heads, d\_k).transpose(1, 2)  K = self.W\_K(input\_K).view(batch\_size, -1, n\_heads, d\_k).transpose(1, 2)  V = self.W\_V(input\_V).view(batch\_size, -1, n\_heads, d\_v).transpose(1, 2)  # 因为是多头，所以mask矩阵要扩充成4维的  # attn\_mask: [batch\_size, seq\_len, seq\_len] -> [batch\_size, n\_heads, seq\_len, seq\_len]  attn\_mask = attn\_mask.unsqueeze(1).repeat(1, n\_heads, 1, 1)  # context: [batch\_size, n\_heads, len\_q, d\_v], attn: [batch\_size, n\_heads, len\_q, len\_k]  context, attn = ScaledDotProductAttention()(Q, K, V, attn\_mask)  # 下面将不同头的输出向量拼接在一起  # context: [batch\_size, n\_heads, len\_q, d\_v] -> [batch\_size, len\_q, n\_heads \* d\_v]  context = context.transpose(1, 2).reshape(batch\_size, -1, n\_heads \* d\_v)  # 这个全连接层可以保证多头attention的输出仍然是seq\_len x d\_model  output = self.fc(context) # [batch\_size, len\_q, d\_model]  return nn.LayerNorm(d\_model).to(device)(output + residual), attn  class PoswiseFeedForwardNet(nn.Module):  def \_\_init\_\_(self):  super(PoswiseFeedForwardNet, self).\_\_init\_\_()  self.fc = nn.Sequential(  nn.Linear(d\_model, d\_ff, bias=False),  nn.ReLU(),  nn.Linear(d\_ff, d\_model, bias=False)  )  def forward(self, inputs):  residual = inputs  output = self.fc(inputs)  return nn.LayerNorm(d\_model).to(device)(output + residual)  class EncoderLayer(nn.Module):  def \_\_init\_\_(self):  super(EncoderLayer, self).\_\_init\_\_()  self.enc\_self\_attn = MultiHeadAttention()  self.pos\_ffn = PoswiseFeedForwardNet()  def forward(self, enc\_inputs, enc\_self\_attn\_mask):  enc\_outputs, attn = self.enc\_self\_attn(enc\_inputs, enc\_inputs, enc\_inputs, enc\_self\_attn\_mask)  enc\_outputs = self.pos\_ffn(enc\_outputs)  return enc\_outputs, attn  class DecoderLayer(nn.Module):  def \_\_init\_\_(self):  super(DecoderLayer, self).\_\_init\_\_()  self.dec\_self\_attn = MultiHeadAttention()  self.dec\_enc\_attn = MultiHeadAttention()  self.pos\_ffn = PoswiseFeedForwardNet()  def forward(self, dec\_inputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask):  dec\_outputs, dec\_self\_attn = self.dec\_self\_attn(dec\_inputs, dec\_inputs, dec\_inputs,dec\_self\_attn\_mask)  dec\_outputs, dec\_enc\_attn = self.dec\_enc\_attn(dec\_outputs, enc\_outputs, enc\_outputs,dec\_enc\_attn\_mask)  dec\_outputs = self.pos\_ffn(dec\_outputs)  return dec\_outputs, dec\_self\_attn, dec\_enc\_attn  class Encoder(nn.Module):  def \_\_init\_\_(self):  super(Encoder, self).\_\_init\_\_()  self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model)  self.pos\_emb = PositionalEncoding(d\_model)  self.layers = nn.ModuleList([EncoderLayer() for \_ in range(n\_layers)])  def forward(self, enc\_inputs):  enc\_outputs = self.src\_emb(enc\_inputs)  enc\_outputs = self.pos\_emb(enc\_outputs.transpose(0, 1)).transpose(0, 1)  enc\_self\_attn\_mask = get\_attn\_pad\_mask(enc\_inputs, enc\_inputs)  enc\_self\_attns = []  for layer in self.layers:  enc\_outputs, enc\_self\_attn = layer(enc\_outputs,enc\_self\_attn\_mask)  enc\_self\_attns.append(enc\_self\_attn)  return enc\_outputs, enc\_self\_attns  class Decoder(nn.Module):  def \_\_init\_\_(self):  super(Decoder, self).\_\_init\_\_()  self.tgt\_emb = nn.Embedding(tgt\_vocab\_size, d\_model) # Decoder输入的embed词表  self.pos\_emb = PositionalEncoding(d\_model)  self.layers = nn.ModuleList([DecoderLayer() for \_ in range(n\_layers)]) # Decoder的blocks  def forward(self, dec\_inputs, enc\_inputs, enc\_outputs):  dec\_outputs = self.tgt\_emb(dec\_inputs)  dec\_outputs = self.pos\_emb(dec\_outputs.transpose(0, 1)).transpose(0, 1).to(device)  dec\_self\_attn\_pad\_mask = get\_attn\_pad\_mask(dec\_inputs, dec\_inputs).to(device)  dec\_self\_attn\_subsequence\_mask = get\_attn\_subsequence\_mask(dec\_inputs).to(device)  dec\_self\_attn\_mask = torch.gt((dec\_self\_attn\_pad\_mask + dec\_self\_attn\_subsequence\_mask),0).to(device)  # [batch\_size, tgt\_len, tgt\_len]; torch.gt比较两个矩阵的元素，大于则返回1，否则返回0  # 这个mask主要用于encoder-decoder attention层  # get\_attn\_pad\_mask主要是enc\_inputs的pad mask矩阵(因为enc是处理K,V的，求Attention时是用v1,v2,..vm去加权的，要把pad对应的v\_i的相关系数设为0，这样注意力就不会关注pad向量)  # dec\_inputs只是提供expand的size的  dec\_enc\_attn\_mask = get\_attn\_pad\_mask(dec\_inputs, enc\_inputs) # [batc\_size, tgt\_len, src\_len]  dec\_self\_attns, dec\_enc\_attns = [], []  for layer in self.layers:  # dec\_outputs: [batch\_size, tgt\_len, d\_model], dec\_self\_attn: [batch\_size, n\_heads, tgt\_len, tgt\_len], dec\_enc\_attn: [batch\_size, h\_heads, tgt\_len, src\_len]  # Decoder的Block是上一个Block的输出dec\_outputs（变化）和Encoder网络的输出enc\_outputs（固定）  dec\_outputs, dec\_self\_attn, dec\_enc\_attn = layer(dec\_outputs, enc\_outputs, dec\_self\_attn\_mask,dec\_enc\_attn\_mask)  dec\_self\_attns.append(dec\_self\_attn)  dec\_enc\_attns.append(dec\_enc\_attn)  return dec\_outputs, dec\_self\_attns, dec\_enc\_attns  class Transformer(nn.Module):  def \_\_init\_\_(self):  super(Transformer, self).\_\_init\_\_()  self.encoder = Encoder().to(device)  self.decoder = Decoder().to(device)  self.projection = nn.Linear(d\_model, tgt\_vocab\_size, bias=False).to(device)  def forward(self, enc\_inputs, dec\_inputs):  # 经过Encoder网络后，得到的输出还是[batch\_size, src\_len, d\_model]  enc\_outputs, enc\_self\_attns = self.encoder(enc\_inputs)  dec\_outputs, dec\_self\_attns, dec\_enc\_attns = self.decoder(dec\_inputs, enc\_inputs, enc\_outputs)  # dec\_outputs: [batch\_size, tgt\_len, d\_model] -> dec\_logits: [batch\_size, tgt\_len, tgt\_vocab\_size]  dec\_logits = self.projection(dec\_outputs)  return dec\_logits.view(-1, dec\_logits.size(-1)), enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns  model = Transformer().to(device)  # 这里的损失函数里面设置了一个参数 ignore\_index=0，因为 "pad" 这个单词的索引为 0，这样设置以后，就不会计算 "pad" 的损失（因为本来 "pad" 也没有意义，不需要计算）  criterion = nn.CrossEntropyLoss(ignore\_index=0)  optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.99)  # =============================================  for epoch in range(epochs):  for enc\_inputs, dec\_inputs, dec\_outputs in loader:  enc\_inputs, dec\_inputs, dec\_outputs = enc\_inputs.to(device), dec\_inputs.to(device), dec\_outputs.to(device)  outputs, enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns = model(enc\_inputs, dec\_inputs)  loss = criterion(outputs, dec\_outputs.view(-1))  print('Epoch:', '%04d' % (epoch + 1), 'loss =', '{:.6f}'.format(loss))  optimizer.zero\_grad()  loss.backward()  optimizer.step()  def greedy\_decoder(model, enc\_input, start\_symbol):  enc\_outputs, enc\_self\_attns = model.encoder(enc\_input)  dec\_input = torch.zeros(1, 0).type\_as(enc\_input.data) # 初始化一个空的tensor: tensor([], size=(1, 0), dtype=torch.int64)  terminal = False  next\_symbol = start\_symbol  while not terminal:  # 预测阶段：dec\_input序列会一点点变长（每次添加一个新预测出来的单词）  dec\_input = torch.cat([dec\_input.to(device), torch.tensor([[next\_symbol]], dtype=enc\_input.dtype).to(device)],  -1)  dec\_outputs, \_, \_ = model.decoder(dec\_input, enc\_input, enc\_outputs)  projected = model.projection(dec\_outputs)  prob = projected.squeeze(0).max(dim=-1, keepdim=False)[1]  # 增量更新（我们希望重复单词预测结果是一样的）  # 我们在预测是会选择性忽略重复的预测的词，只摘取最新预测的单词拼接到输入序列中  next\_word = prob.data[-1]  next\_symbol = next\_word  if next\_symbol == tgt\_vocab["E"]:  terminal = True  greedy\_dec\_predict = dec\_input[:, 1:]  return greedy\_dec\_predict  # 预测阶段  # 测试集  sentences = [  # enc\_input dec\_input dec\_output  ['亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司 P', '', '']  ]  enc\_inputs, dec\_inputs, dec\_outputs = make\_data(sentences)  test\_loader = Data.DataLoader(MyDataSet(enc\_inputs, dec\_inputs, dec\_outputs), 2, True)  enc\_inputs, \_, \_ = next(iter(test\_loader))  print()  print("="\*45)  print("利用训练好的Transformer模型将中文句子'亚 马 逊 公 司 是 美 国 最 大 的 一 家 网 络 电 子 商 务 公 司 ' 翻译成英文句子: ")  for i in range(len(enc\_inputs)):  greedy\_dec\_predict = greedy\_decoder(model, enc\_inputs[i].view(1, -1).to(device), start\_symbol=tgt\_vocab["S"])  print(enc\_inputs[i], '->', greedy\_dec\_predict.squeeze())  print([src\_idx2word[t.item()] for t in enc\_inputs[i]], '->',  [idx2word[n.item()] for n in greedy\_dec\_predict.squeeze()])  dec\_predict=[idx2word[n.item()] for n in greedy\_dec\_predict.squeeze()]  sequence=" ".join(dec\_predict)  print(sequence)  cache\_dir = "./transformersModels/ner"  """  ,cache\_dir = cache\_dir  """  model = AutoModelForTokenClassification.from\_pretrained("dbmdz/bert-large-cased-finetuned-conll03-english",  cache\_dir=cache\_dir, return\_dict=True)  tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased", cache\_dir=cache\_dir)  # Bit of a hack to get the tokens with the special tokens  tokens = tokenizer.tokenize(tokenizer.decode(tokenizer.encode(sequence)))  inputs = tokenizer.encode(sequence, return\_tensors="pt")  outputs = model(inputs).logits  predictions = torch.argmax(outputs, dim=2)  for token, prediction in zip(tokens, predictions[0].numpy()):  print(token, label\_list[prediction])   1. 测试与实验结果   当epoch很小比如为6时，Transformer模型的loss还很大，如下：  IMG_256  当epoch=100时，会发现这时的loss已经很小了，模型拟合的比较好，已经能够输出结果了；IMG_256  将上述模型获得的结果稍做处理后来进行标签预测。  获得的结果是9个分类的概率分布。  一般是使用最高概率的那个标签最为最终预测结果。  将每个标记与其预测标签一起打印出来。  IMG_256  然后将实践项目通过git上传到github。 |
| 实验体会：  通过这次实验，我对Transformer模型有了更进一步的认识。  1、transformer是编码器－解码器架构的一个实践，在实际情况中编码器或解码器可以单独使用。  2、在transformer中，多头自注意力用于表示输入序列和输出序列，不过解码器必须通过掩蔽机制来保留自回归属性。  3、transformer中的残差连接和层规范化是训练非常深度模型的重要工具。  4、transformer模型中基于位置的前馈网络使用同一个多层感知机，作用是对所有序列位置的表示进行转换。  在transform的基础上加入实体识别功能，通过bert模型实例化预训练模型和对应文本标记器，不过最后输出结果不能删除“O”实体，需要手动整理，仍需进一步改进。  在实验过程中对Torch也有了初步的理解，是一个有大量机器学习算法支持的科学计算框架，是一个与Numpy类似的张量（Tensor）操作库。因为刚开始不了解，在定义数据时就出现了差错：ValueError: expected sequence of length 22 at dim 1 (got 14)。发现这个问题出现在对tensor的转换中，tensor的转换要求内部的数组维度相同。后修改数据长度一致即可。 |