|  |
| --- |
| def make\_data(sentences):  # 把单词转化为数字，然后构建enc\_inputs, dec\_inputs, dec\_outputs，分别存放每个语句的数字表示  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) # 将enc\_input中的内容逐一添加到enc\_inputs中  dec\_inputs.extend(dec\_input)  dec\_outputs.extend(dec\_output)  # 返回张量tensor  return torch.LongTensor(enc\_inputs), torch.LongTensor(dec\_inputs), torch.LongTensor(dec\_outputs)    enc\_input, dec\_input, dec\_output = make\_data(sentences)  class MyDataSet(Data.Dataset):  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\_input, dec\_input, dec\_output), 2, True) |

* extend()将内容逐一添加，例：

enc\_inputs = torch.LongTensor([

[1, 2, 3, 4, 0], # 对应句子 "ich mochte ein bier P"

[1, 2, 3, 5, 0] # 对应句子 "ich mochte ein cola P"

])

* batch\_size = 2，表示：在每次数据加载时，DataLoader 返回的数据样本数。例：

enc\_inputs = torch.tensor([ [1, 2, 3, 4, 0], [1, 2, 3, 5, 0], [1, 6, 7, 8, 0], [2, 3, 4, 5, 0] ])

第一次加载：enc\_batch = torch.tensor([ [1, 2, 3, 4, 0], [2, 3, 4, 5, 0] ])

第二次加载：enc\_batch = torch.tensor([ [1, 2, 3, 5, 0], [1, 6, 7, 8, 0] ])

（顺序可能不一样，因为shuffle=True）

|  |
| --- |
| # 位置编码，补充了自注意力机制中对序列位置的敏感性  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) |

* pe = torch.zeros(max\_len, d\_model)

作用： 创建一个形状为 [max\_len, d\_model] 的零张量，用来**存储位置编码矩阵**。

max\_len: 序列最大长度，即位置的数量。

d\_model: 每个位置的编码维度。

目的： 初始化位置编码矩阵，稍后会填入正弦和余弦值。

* position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

作用： 生成从 0 到 max\_len-1 的整数序列，表示每个位置的索引，并将其形状调整为 [max\_len, 1]。

torch.arange(0, max\_len, dtype=torch.float): 生成一个 1D 张量，包含从 0 到 max\_len-1 的值。

.unsqueeze(1): 将张量增加一个维度，变为 [max\_len, 1]，方便与后续的广播操作。

例：

max\_len = 5

position = torch.arange(0, 5, dtype=torch.float).unsqueeze(1)

# position = [[0.],

# [1.],

# [2.],

# [3.],

# [4.]]

**· 生成的 pe 张量的形状是 [max\_len, 1, d\_model]。**

**· 在实际使用时，经过切片和广播，位置编码的维度会动态适配为 [seq\_len, batch\_size, d\_model]**

|  |
| --- |
| get\_attn\_pad\_mask(seq\_q, seq\_k) |
| Seq是指序列长度，不是维度 |

例：

输入：seq\_q = [[1, 2, 3], [4, 5, 6]] # [batch\_size=2, seq\_len\_q=3]

seq\_k = [[1, 0, 3, 0], [7, 8, 0, 0]] # [batch\_size=2, seq\_len\_k=4]

输出：

pad\_attn\_mask =

[[[False, True, False, True],

[False, True, False, True],

[False, True, False, True]],

[[False, False, True, True],

[False, False, True, True],

[False, False, True, True]]]