

2101978-胡泽航-自然语言处理作业报告

基于Transformer的机器翻译PyTorch实现报告

基础知识

nn.Transformer的使用

在PyTorch中有Transformer的API接口，可直接调用，但其并没有实现Embedding和Positional Encoding以及Linear和Softmax层。

- Embedding：主要是将tokens映射成为高维向量。通常使用 nn.Embedding实现，并且其参数会参与梯度下降随之变化的。

nn.Embedding的使用

nn.Embedding传入两个重要参数：

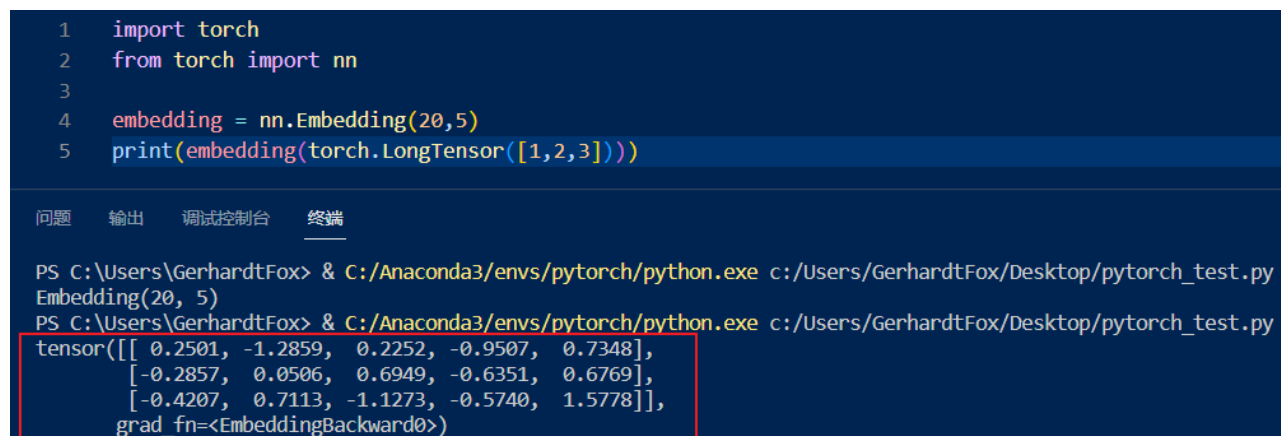
1. num_embedding：字典的大小。
2. embedding_dim：单词被编码的维度。

假如：设定词典大小为20，需要对3个单词进行编码，每个单词的维度为5：

```
import torch
from torch import nn

embedding = nn.Embedding(20,5)
print(embedding(torch.LongTensor([0,1,2])))
```

输出结果：



```
1 import torch
2 from torch import nn
3
4 embedding = nn.Embedding(20,5)
5 print(embedding(torch.LongTensor([1,2,3])))
```

问题 输出 调试控制台 终端

```
PS C:\Users\GerhardtFox> & C:/Anaconda3/envs/pytorch/python.exe c:/Users/GerhardtFox/Desktop/pytorch_test.py
Embedding(20, 5)
PS C:\Users\GerhardtFox> & C:/Anaconda3/envs/pytorch/python.exe c:/Users/GerhardtFox/Desktop/pytorch_test.py
tensor([[ 0.2501, -1.2859,  0.2252, -0.9507,  0.7348],
        [-0.2857,  0.0506,  0.6949, -0.6351,  0.6769],
        [-0.4207,  0.7113, -1.1273, -0.5740,  1.5778]],
        grad_fn=<EmbeddingBackward0>)
```

通过nn.Embedding: 0 → [0.251, -1.2859, 0.2252, -0.9507, 0.7348]

注意事项：

1. Embedding只接受LongTensor类型的数据。
2. Embedding的数据不能 \geq 词典的大小，上例中的编码索引大小需 ≤ 20 。

nn.Embedding的其它常用参数

padding_idx: 填充索引，即索引用全0表示。通常情况，字典中的unknown代表未知的单词，即可用填充索引为0表示。

```
import torch
from torch import nn

embedding = nn.Embedding(20,6,padding_idx=2)
print(embedding(torch.LongTensor([0,1,2,3])))
```

输出结果：

```
1  import torch
2  from torch import nn
3
4  embedding = nn.Embedding(20,6,padding_idx=2)
5  print(embedding(torch.LongTensor([0,1,2,3])))
```

问题	输出	调试控制台	终端
PS C:\Users\GerhardtFox> & C:/Anaconda3/envs/pytorch/python.exe c:/Users/GerhardtFox/Desktop/pytorch_test.py tensor([[-9.2998e-01, 7.9557e-01, 1.3679e+00, -1.4539e+00, -1.5968e-03, 1.9273e-02], [-9.3077e-01, -8.8793e-02, -1.9001e-01, 7.1656e-01, 1.8599e+00, 4.1985e-01], [0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00], [-1.6637e+00, -2.2817e-01, 8.2686e-01, 6.9462e-01, -1.7253e-01, -4.5389e-01]], grad_fn=<EmbeddingBackward0>) PS C:\Users\GerhardtFox> []			

索引2的编码表示全为0，即不编码表示。

nn.Embedding的可学习性

nn.Embedding的参数在参与梯度下降后，也会更新，即nn.Embedding也是模型参数的一部分。

```
import torch
from torch import nn

embedding = nn.Embedding(20,5,padding_idx=3)
print(embedding(torch.LongTensor([0,1,2,3,4])))
optimizer = torch.optim.SGD(embedding.parameters(),lr=0.1)
criteira = nn.MSELoss()

for i in range(1000):
    outputs = embedding(torch.LongTensor([0,1,2,3,4]))
    loss = criteria(outputs, torch.ones(5,5))
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(embedding(torch.LongTensor([0,1,2,3,4])))
```

输出结果：

```

1 import torch
2 from torch import nn
3
4 embedding = nn.Embedding(20,5,padding_idx=3)
5 print(embedding(torch.LongTensor([0,1,2,3,4])))
6 optimizer = torch.optim.SGD(embedding.parameters(),lr=0.1)
7 criteria = nn.MSELoss()
8
9 for i in range(1000):
10     outputs = embedding(torch.LongTensor([0,1,2,3,4]))
11     loss = criteria(outputs, torch.ones(5,5))
12     loss.backward()
13     optimizer.step()

```

问题 输出 调试控制台 终端

```

PS C:\Users\GerhardtFox> & C:/Anaconda3/envs/pytorch/python.exe c:/Users/GerhardtFox/Desktop/pytorch_test.py
tensor([[ 1.2509,  0.7499, -1.2425,  0.9883, -0.9073],
        [ 0.2978, -2.4826, -0.5413, -0.7130, -2.4439],
        [-1.5918,  0.3338,  0.4108, -0.2762,  1.2084],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 2.3941,  0.2868, -0.6068,  0.9432, -1.2022]],
        grad_fn=<EmbeddingBackward0>)
tensor([[1.0045, 0.9955, 0.9594, 0.9998, 0.9655],
        [0.9873, 0.9369, 0.9721, 0.9690, 0.9376],
        [0.9531, 0.9879, 0.9893, 0.9769, 1.0038],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [1.0253, 0.9871, 0.9709, 0.9990, 0.9601]],
        grad_fn=<EmbeddingBackward0>)
PS C:\Users\GerhardtFox>

```

能发现通过1000次计算损失和梯度下降后，编码表示向1的方向靠近。

Positional Encoding 位置编码

增加tokenss编码的位置信息，以丰富tokenss的相对位置信息。

Linear+Softmax

即一个线性层加一个Softmaxm,作用是对nn.Transformer输出的结果进行tokenss预测。

```

# 设定词典大小为10，向量维度128
embedding = nn.Embedding(10,128)
# 定义transformer，模型维度设为128，即词向量的维度
transformer = nn.Transformer(d_model=128,batch_first=True)

# 生成一个源句子向量
src = torch.LongTensor([0,3,4,5,6,7,8,1,2,2])
# 定义一个需要翻译的目标句子
# 需要注意的是索引编号 < 10
tgt = torch.LongTensor([0,3,4,5,6,7,8,1,2])
# 将tokens编码传入transformer
outputs = transformer(embedding(src),embedding(tgt))

```

输出结果：

```

embedding = nn.Embedding(10,128)
transformer = nn.Transformer(d_model=128,batch_first=True)

src = torch.LongTensor([0,3,4,5,6,7,8,1,2,2])

tgt = torch.LongTensor([0,1,2,4,5,6,7,2])

outputs = transformer(embedding(src),embedding(tgt))
outputs

```

[2] ✓ 0.1s

```

... tensor([[ 0.4890, -0.6730, 1.3424, ..., 1.1506, -0.0687, 0.0827],
           [-0.2459, -0.8831, 1.3243, ..., 1.5498, 0.7505, 0.5060],
           [ 0.5415, -0.1368, 1.4894, ..., 0.3562, -0.0960, 0.8446],
           ...,
           [ 0.1857, 0.0040, 1.6900, ..., 1.6941, 0.0437, 0.6859],
           [ 0.7285, -1.0453, 0.4917, ..., 1.0653, -0.2404, 0.6447],
           [-0.1664, -0.5307, 0.7943, ..., 0.3912, 0.4403, 0.7004]],
        grad_fn=<NativeLayerNormBackward0>)

```

nn.Transformer的构造参数

nn.Transformer主要有两部分构成：nn.TransformerEncoder和nn.TransformerDecoder。

而这两部分又是由nn.TransformerEncoderLayer和nn.TransformerDecoderLayer堆叠而成。

主要的构造参数

- d_model: Encoder和Decoder输入参数的特征维度，即词向量的维度，一般默认512。
- nhead: 多头注意力机制中，head的数量，默认值为8。
- num_encoder_layers、num_decoder_layers: 熟练越大，网络越深，参数维多，计算量越高，默认值为8。
- dim_feedforward: Feed Forward层的隐藏层的神经元数量。默认为2048。
- dropout: 默认值为0.1。
- activation: Feed Forward层的激活函数。取值可为“relu”和“gelu”，或者自定一个一元可调参数。默认为relu
- custom_encoder、custom_decoder: 自定义Encoder、Decoder，默认为None。
- layer_norm_eps: Add和Norm层中，BatchNorm的eps参数值，默认为0.00001。
- batch_first: batch维度是否是第一个。如果为True，则输入的shape应该为（batch_size，词数，词向量的维度），否则应为（词数，batch_size，词向量维度）。默认为False。
 - 需要注意的是，很多习惯性把batch_size放在最前面，而参数值默认False，因此会报错。
- norm_first: 是否先执行norm。一般地顺序为：Attention -> Add -> Norm。若该值为True，则执行顺序变为：Norm -> Attention -> Add。

Transformer的forward参数

- src: Encoder的输入，将tokens进行Embedding并假设Position Encoing的Tensor。
 - 必设定参数，Shape为（batch_size，词数，词向量维度）

- tgt: Decoder的输入，同上。
- src_mask: 对src进行mask。不常用
 - Shape为 (词数, 词数)
- tgt_mask: 对tgt进行mask。常用
 - Shape为 (词数, 词数)
- memory_mask: 对Encoder的输出memory进行mask。不常用。
 - Shape为 (batch_size, 词数, 词数)
- src_key_padding_mask: 对src的tokens进行mask。常用
 - Shape为 (batch_size, 词数)
- tgt_key_padding_mask: 对tgt的token进行mask。常用
 - Shape为 (batch_size, 词数)
- memory_key_padding_mask: 对tgt的tokens进行mask。不常用。
 - Shape为 (batch_size, 词数)

mask为0代表不遮掩，-inf代表遮掩。

src_mask、tgt_mask和memory_mask不需要传入batch_size参数。

PyTorch1.12版本后，key_padding_mask参数只能为True/False，True表示遮掩，False表示不遮掩，与原论文的实现相反，如果设置错了会造成nan。

src和tgt的使用

src和tgt两个参数是对tokens编码后，再加上Position Encoding的结果，最后作为Encoder和Decoder的输入参数。

假设输入：[[0,3,4,5,6,7,8,1,2,2]]，Shape为 (1,10) ，其中batch_size为1，每句10个词。

经过Embedding后，Shape就变成了 (1,10,128) ，表示batch_size为1，每局10个词，每个词被编码为128维的词向量。

而src就是这个(1,10,128)的向量，tgt同理。

src_mask、tgt_mask和memory_mask的使用

- 例句1: 苹果 很 好吃。
- 例句2: 苹果 手机 很好玩。

显然如果不考虑上下文关系，两个苹果很可能被归为同一类。

再Attention机制中，通过一个方阵描述词与词的关系：

```
1| 苹果 很 好吃
2|苹果 [[ 0 0.1 0.4],
3|很 [ 0.1 0.8 0.1],
4|好吃 [0.3 0.1 0.6],]
```

于是，苹果' = 苹果*0.5+很*0.1+好吃*0.4。

但是实际推理的时候，词是一次吞吐的。若”苹果很好吃”是tgt的话，苹果没有上下文信息。

因此：苹果'=苹果*0.5

同理：很=苹果*0.1+很*0.8

综上：方阵被更新为：

```
1| 苹果 很 好吃
2| 苹果 [[ 0.5 0 0],
3| 很 [ 0.1 0.8 0],
4| 好吃 [0.3 0.1 0.6],]
```

经过掩码的方阵就是

```
1| 苹果 很 好吃
2| 苹果 [[ 0 -inf -inf],
3| 很 [ 0 0 -inf],
4| 好吃 [ 0 0 0],]
```

其中0表示不遮掩，-inf表示遮掩。由于方阵会经过一个softmax所以，最后会被更新为0。

所以tgt_mask只需要生成一个斜覆盖的方阵即可

```
# 需要传入tgttokens的数量
nn.Transformer.generate_square_subsequent_mask(5)
```

```
nn.Transformer.generate_square_subsequent_mask(5)
[5] ✓ 0.8s
... tensor([[0., -inf, -inf, -inf, -inf],
           [0., 0., -inf, -inf, -inf],
           [0., 0., 0., -inf, -inf],
           [0., 0., 0., 0., -inf],
           [0., 0., 0., 0., 0.]])
```

src和memory一般不需要mask操作，因此不常用。

key_padding_mask

在src和tgt的语句中，除了本身的词向量编码外，还包括三种tokens：

- <bos>：开始
- <eos>：结束
- <pad>：通过改变句子长度，将不同长度的句子统一化组成batch，没有实际意义。
 - 因此，在attention计算式，需要mask处理。通过key_padding_mask

例如：sr为[[0,3,4,5,6,7,8,1,2,2]]，其中2表示<pad>，因此src_key_padding_mask后则为[[0,0,0,0,0,0,0,0,-inf,-inf]]。而memory_key_padding_mask就没有更多地必要了。

而Transformer源码或者实现过程，tgt_mask和tgt_key_padding_mask是合并在一起的。

nn.Transformer的使用

```
# 定义src和tgt
src = torch.LongTensor([
    [0, 8, 3, 5, 5, 9, 6, 1, 2, 2, 2],
    [0, 6, 6, 8, 9, 1, 2, 2, 2, 2, 2],
])

tgt = torch.LongTensor([
    [0, 8, 3, 5, 5, 9, 6, 1, 2, 2],
    [0, 6, 6, 8, 9, 1, 2, 2, 2, 2],
])

# 定义辅助函数生成src_key_padding_mask
def get_key_padding_mask(tokens):
    key_padding_mask = torch.zeros(tokens.size())
    # 2表示<pad>的token, 所以进行掩码操作, 设置为-inf
    key_padding_mask[tokens == 2] = -torch.inf
    return key_padding_mask

src_key_padding_mask = get_key_padding_mask(src)
tgt_key_padding_mask = get_key_padding_mask(tgt)
print(tgt_key_padding_mask)

tensor
```

输出结果：

```
src = torch.LongTensor([
    [0, 8, 3, 5, 5, 9, 6, 1, 2, 2, 2],
    [0, 6, 6, 8, 9, 1, 2, 2, 2, 2, 2],
])
tgt = torch.LongTensor([
    [0, 8, 3, 5, 5, 9, 6, 1, 2, 2],
    [0, 6, 6, 8, 9, 1, 2, 2, 2, 2],
])

def get_key_padding_mask(tokens):
    key_padding_mask = torch.zeros(tokens.size())
    key_padding_mask[tokens == 2] = -torch.inf
    return key_padding_mask

src_key_padding_mask = get_key_padding_mask(src)
tgt_key_padding_mask = get_key_padding_mask(tgt)
print(tgt_key_padding_mask)
```

✓ 0.5s

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],
       [0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf]])
```

```
tgt_mask = nn.Transformer.generate_square_subsequent_mask(tgt.size(-1))
print(tgt_mask)
```

输出结果：

```
tgt_mask = nn.Transformer.generate_square_subsequent_mask(tgt.size(-1))
tgt_mask

[4] ✓ 0.3s Python

... tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
          [0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
          [0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
          [0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
          [0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf],
          [0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf],
          [0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf],
          [0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],
          [0., 0., 0., 0., 0., 0., 0., 0., 0., -inf],
          [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
# 定义编码器，词典大小为10，把tokens编码成128维的向量
embedding = nn.Embedding(10,128)
# 定义transformer，模型的词向量维度为128
# batch_first一定要打开，否则的话，后面不注意就会报错
transformer = nn.Transformer(d_model=128, batch_first=True)
# 将token编码后传入 transformer
outputs = transformer(embedding(src),embedding(tgt),
                      tgt_mask=tgt_mask,
                      src_key_padding_mask=src_key_padding_mask,
                      tgt_key_padding_mask=tgt_key_padding_mask)
outputs.size()
```

输出结果：

```
# 定义编码器，词典大小为10，把token编码为128维向量
embedding = nn.Embedding(10,128)
# 定义transformer，模型维度为128
transformer = nn.Transformer(d_model=128,batch_first=True)
# 将token编码传入Transformer
outputs = transformer(embedding(src),embedding(tgt),tgt_mask=tgt_mask,
                      src_key_padding_mask=src_key_padding_mask,tgt_key_padding_mask=tgt_key_padding_mask )
outputs.size()

[6] ✓ 0.1s

... torch.Size([2, 10, 128])
```

实现简单的机器翻译-Copy任务

任务描述【附代码】

通过Transformer模型预测输入。例如：输入向量[0, 3, 4, 6, 7, 1, 2, 2]，则期望的输出为[0, 3, 4, 6, 7, 1]


```

# 定义句子最大长度
max_length = 16

# 定义PositionEncoding
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding,self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # 初始化Shape为 (max_len, d_model) 的PE (Position Encoding)
        pe = torch.zeros(max_len, d_model)
        # 初始化一个tensor
        position = torch.arange(0, max_len).unsqueeze(1)
        # sin和cos括号的内容, 通过e和ln进行变换
        div_term = torch.exp(
            torch.arange(0,d_model, 2) * -(math.log(10000.0) / d_model)
        )
        # 计算PE (pos, 2i)
        pe[:,1::2] = torch.sin(position * div_term)
        # 方便计算, 在最外面再unsqueeze出一个batch
        pe = pe.unsqueeze(0)
        self.register_buffer("pe",pe)

    def forward(self,x):
        """
        x为embedding后的inputs, 例如: (1, 7, 128), batch size为1, 7个单词, 单词维度为128
        """
        # 将x和position encoding 相加
        x = x + self.pe[:,x.size(1)].requires_grad_(False)
        return self.dropout(x)

```

```

# 定义Copy 模型
class CopyTaskModel(nn.Module):
    def __init__(self, d_model=128):
        super(CopyTaskModel, self).__init__()

        # 定义词向量, 词典数定义为10
        self.embedding = nn.Embeddin(num_embedding=10, embedding_dim=128)
        # 定义Trandformer
        self.Transformer = nn.Transformer(d_model=128, num_encoder_layers=2,num_decoder_layers=2,
        dim_feedforward=512,batch_first=True)

    # 定义位置编码器
    self.positional_encoding = PositionalEncoding(d_model, dropout=0)

    # 定义线性层
    self.predictor = nn.Linear(128,10)

    def forward(self, src, tgt):
        # 生成mask
        tgt_mask = nn.Transformer.generate_square_subsequent_mask(tgt.size()[-1])
        src_key_padding_mask = CopyTaskModel.get_key_padding_mask(src)
        tgt_key_padding_mask = CopyTaskModel.get_key_padding_mask(src)

        # 对src和tgt进行编码
        src = self.embedding(src)

```

```

tgt = self.embedding(tgt)
# 增加src、tgt的位置信息
src = self.positional_encoding(src)
tgt = self.positional_encoding(tgt)

# 将参数传入transformer模型中
out = self.transformer(src, tgt,
                       tgt_mask=tgt_mask,
                       src_key_padding_mask=src_key_padding_mask,
                       tgt_key_padding_mask=tgt_key_padding_mask)

return out

```

```

def get_key_padding_mask(tokens):
    key_padding_mask = torch.zeros(tokens.size())
    key_padding_mask[tokens==2] = -torch.inf
    return key_padding_mask

```

```
model = CopyTaskModel()
```

```

# 定义模型并打印
src = torch.LongTensor([[0, 3, 4, 5, 6, 1, 2, 2]])
tgt = torch.LongTensor([[3, 4, 5, 6, 1, 2, 2]])
print(out.size())
print(out)

```

输出结果：

```

# 定义模型
src = torch.LongTensor([[0, 3, 4, 5, 6, 1, 2, 2]])
tgt = torch.LongTensor([[3, 4, 5, 6, 1, 2, 2]])
out = model(src, tgt)
print(out.size())
print(out)

```

[4] ✓ 0.1s

... Output exceeds the [size limit](#). Open the full output data [in a text editor](#)

torch.Size([1, 7, 128])

```

tensor([[[[ 7.7937e-01, -2.6861e-01,  1.0908e-01,  3.6226e-01, -1.3632e-01,
            -1.8220e-01,  1.8007e+00, -1.5151e+00, -7.9253e-01,  1.0203e-01,
            -5.5601e-01,  4.2742e-01, -2.7759e-02, -7.5134e-01,  1.7668e+00,
            -2.5165e-01, -7.0847e-01,  1.2675e-01, -2.2410e-01, -7.3232e-01,
            9.2316e-01, -2.1264e+00,  7.7233e-02,  1.6516e+00,  3.0351e-01,
            -1.5142e+00, -2.0497e+00, -4.0777e-01, -6.6338e-01, -2.1573e-01,
            1.2512e+00, -7.0840e-01,  5.8132e-01, -2.1783e-01, -8.3253e-01,
            -5.6877e-01, -3.2882e-02,  1.4784e+00,  1.9444e+00,  1.2071e+00,
            3.7830e-01,  3.5669e-01, -7.0190e-01,  1.5495e+00, -3.4972e-01,
            -1.1375e+00,  1.4918e+00,  9.1582e-01, -1.9562e+00, -1.2564e-01,
            3.4012e-02,  1.0158e-01, -9.8235e-02, -2.3314e+00,  6.2970e-02,

```

```

# 定义优化器
criteria = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=3e-4)

# 定义个生成随机数据的工具函数
def generate_random_batch(batch_size, max_length=16):
    src = []
    for i in range(batch_size):
        # 随机生成句子长度
        random_len = random.randint(1, max_length - 2)
        # 如果句子长度不够, 自动填充
        random_nums = random_nums + [2] * (max_length - random_len - 2)
        src.append(random_nums)
    src = torch.LongTensor(src)
    # tgt不需要最后一个token
    tgt = src[:, :-1]
    # tgt_y不需要第一个token
    tgt_y = src[:, 1:]
    # 计算tgt_y, 预测有效tokens的数量
    n_tokens = (tgt_y != 2).sum()
    return src, tgt, tgt_y, n_tokens

generate_random_batch(batch_size=2, max_length=6)

```

```

# 模型训练
total_loss = 0

for step in range(7000):
    src, tgt, tgt_y, n_tokens = generate_random_batch(batch_size=2, max_length=max_length)

    # 清空梯度
    optimizer.zero_grad()
    # transformer计算
    out = model(src, tgt)
    # 结果传入最后的线性层进行预测
    out = model.predictor(out)

    """
    计算损失。
    因为训练时是对所有的输出进行预测, 因此需要对out进行reshape, 其中out.shape为 (batch_size, 词数, 词典大小),
    view后:
    (batch_size*词数, 词典大小)
    在这些预测结果中, 只对非<pad>部分进行, 所以需要正则化, 即除以n_tokens。
    """
    loss = criteria(out.contiguous().view(-1, out.size(-1)), tgt_y.contiguous().view(-1)) / n_tokens

    # 计算梯度
    loss.backward()
    # 更新参数
    optimizer.step()

    total_loss += loss

# 每40词打印一下loss

```

```
if step != 0 and step % 40 == 0:
    print("Step{}, total_loss:{}".format(step, total_loss))
    total_loss = 0
```

输出结果：

```
Step 120, total_loss: 1.8996224403381348
Step 160, total_loss: 1.811518669128418
Step 200, total_loss: 1.7493947744369507
Step 240, total_loss: 1.5963976383209229
Step 280, total_loss: 1.6734224557876587
Step 320, total_loss: 1.4389110803604126
Step 360, total_loss: 1.4872270822525024
Step 400, total_loss: 1.4142926931381226
Step 440, total_loss: 1.2817364931106567
Step 480, total_loss: 1.2306493520736694
Step 520, total_loss: 1.1341867446899414
Step 560, total_loss: 1.1111063957214355
Step 600, total_loss: 1.1130568981170654
Step 640, total_loss: 1.1855270862579346
Step 680, total_loss: 1.0862767696380615
Step 720, total_loss: 0.9880492091178894
Step 760, total_loss: 0.9561367630958557
Step 800, total_loss: 0.9759232401847839
Step 840, total_loss: 0.8041670918464661
Step 880, total_loss: 0.802263081073761
Step 920, total_loss: 0.7462620735168457
Step 960, total_loss: 0.8303849101066589
Step 1000, total_loss: 0.7199651598930359
...
Step 6600, total_loss: 0.04935519024729729
Step 6640, total_loss: 0.33785974979400635
Step 6680, total_loss: 0.23801358044147491
Step 6720, total_loss: 0.17435364425182343
```

```
# 模型预测
model = model.eval()

# 随意定义一个src
src = torch.LongTensor([[0, 4, 3, 4, 6, 8, 9, 9, 8, 1, 2, 2]])
# tgt从<bos>开始，预测能否重新输出src的值
tgt = torch.LongTensor([[0]])

# 依次对单个词预测，直到<eos>或者达到句子最大长度为止
for i in range(max_length):
    # 进行transformer计算
    out = model(src, tgt)
```

```

# 预测结果，因为只需要看到最后一个词，因此取out[:, -1]
predict = model.predictor(out[:, -1])
# 找到最大值的索引
y = torch.argmax(predict, dim=1)
# 和之前的预测拼接在一起
tgt = torch.concat([tgt, y.unsqueeze(0)], dim=1)

# 如果为<eos>说明一侧结束，跳出循环
if y == 1:
    break
print(tgt)

```

输出结果：

```

model = model.eval()

# 随机定义一个src
src = torch.LongTensor([[0, 4, 3, 4, 6, 8, 9, 9, 8, 1, 2, 2]])
# tgt从<bos>开始
tgt = torch.LongTensor([[0]])

# 依次对单个词预测，知道预测为<eos>，或达到句子最大长度
for i in range(max_length):
    # 进行Transformer计算
    out = model(src, tgt)
    # 预测结果，只需要看到最后一个词是否为<eos>，所以取out[:, -1]
    predict = model.predictor(out[:, -1])
    # 找出最大值的索引
    y = torch.argmax(predict, dim=1)
    # 与之前预测结果拼接在一起
    tgt = torch.concat([tgt, y.unsqueeze(0)], dim=1)

    # 如果为<eos>，说明预测结束，跳出循环
    if y == 1:
        break
print(tgt)

```

[7] ✓ 0.1s

... tensor([[0, 4, 3, 4, 6, 8, 8, 5, 8, 1]])

代码架构

具体见代码注释，自己参考网上的教程巧了一遍，因为版本的差异，部分小细节做了些小修改。

- 环境配置
- 数据预处理
- 文本分词与词典构造
- 数据集与数据导入

- 构建模型
- 模型预测

个人笔记本GPU Quadro P600

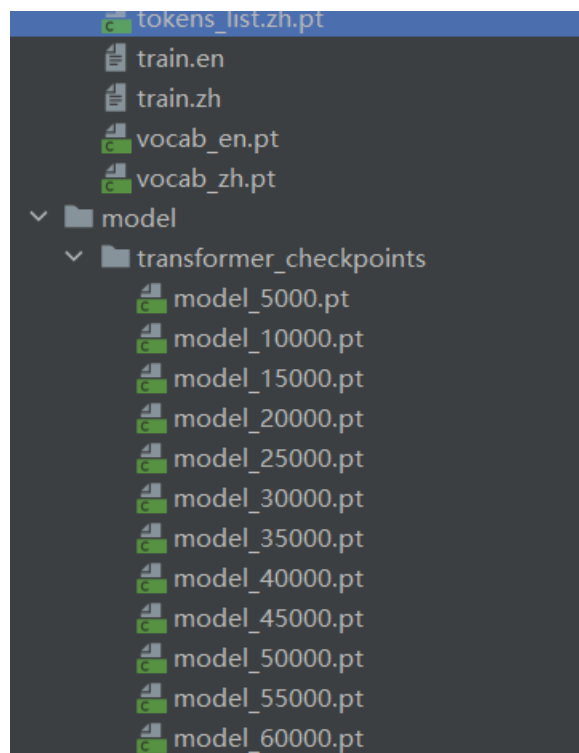
参数设置

一开始Batch Size=64，电脑显存带不动，改设置为32，但是训练效果如下：

```
47
48     if step != 0 and step % save_after_step == 0:
49         torch.save(model, model_dir / f"model_{step}.pt")

Epoch 0/10:  0%|          | 714/312500 [09:14<93:18:11,  1.08s/it, loss=5.43]
```

这预计在笔记本上训练时间太久了，大概一个Epoch50个小时，我自己训练了10个小时第一个epoch到”6000.pt“就结束了，真的要训练完，我电脑估计要很久，可能一周吧。



测试结果

```
In 25 1 translate("Okay, this is all what I want to say. Can you guess how much I love you?")

Out 25      '好吧，这是我想说的。你想我多么想。我怎么想？'
```

效果不是很好哎，后面的疑问复合句翻译的不是很好：Can you guess how much I love you?

云服务器测试

结果如下：

```
ServerApp.rate_limit_window=3.0 (secs)
Epoch 0/1: 100%|██████████| 39063/39063 [6:44:03<00:00, 1.05s/it, loss=2.21]

[24]: model = model.eval()

[25]: def translate(src: str):
      # 将与原句子分词后，通过词典转为index，然后增加<bos>和<eos>
      src = torch.tensor([0] + en_vocab(en_tokenizer(src)) + [1]).unsqueeze(0).to(device)
      # 翻译句子
```

配置NVIDIA V100 CPU8核 32内存，只跑了一个Epoch用时大约6个半小时。Btch_size=256，相比于自己的笔记

```
    return tgt

[26]: translate("Okay, this is all what I want to say. Can you guess how much I love you?")

[26]: '好吧，这就是我想说的。你猜你猜我怎么想我多爱你你能猜我多爱你？'

[27]: translate("I saw a saw.")

[27]: '我看见见见见见见见见见见见见见见见见见。'
```

同样的好像效果也不是很好。可能跟语料库没有问句有关吧。

我又增加了一个测试“I saw a saw”，同一个单词的动词过去式和名词的用法，它好像处理的也挺一般的。

也许是我找模型架构设计还是不是很合理，这种棘手的问题，需要一定的设计。

参考资料

[Pytorch nn.Embedding的基本使用](#)

[nn.Embedding官方文档](#)

[Pytorch中 nn.Transformer的使用详解与Transformer的黑盒讲解](#)

[层层剖析，让你彻底搞懂Self-Attention、MultiHead-Attention和Masked-Attention的机制和原理](#)

[Pytorch中DataLoader和Dataset的基本用法](#)