B 站视频讲解

本文主要介绍一下如何使用 PyTorch 复现 Transformer，实现简单的机器翻译任务。请先花上 15 分钟阅读我的这篇文章 Transformer 详解，再来看本文，方能达到醍醐灌顶，事半功倍的效果

数据预处理

这里我并没有用什么大型的数据集，而是手动输入了两对德语→英语的句子，还有每个字的索引也是我手动硬编码上去的，主要是为了降低代码阅读难度，我希望读者能更关注模型实现的部分

```
import math
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data

# S: Symbol that shows starting of decoding input
# E: Symbol that shows starting of decoding output
# P: Symbol that will fill in blank sequence if current batch data size is short than time steps
sentences = [

        # enc_input           dec_input         dec_output

        ['ich mochte ein bier P', 'S i want a beer .', 'i want a beer . E'],
        ['ich mochte ein cola P', 'S i want a coke .', 'i want a coke . E']
]



# Padding Should be Zero

src_vocab = {'P' : 0, 'ich' : 1, 'mochte' : 2, 'ein' : 3, 'bier' : 4, 'cola' : 5}
src_vocab_size = len(src_vocab)



tgt_vocab = {'P' : 0, 'i' : 1, 'want' : 2, 'a' : 3, 'beer' : 4, 'coke' : 5, 'S' : 6, 'E' : 7, '.' : 8}
idx2word = {i: w for i, w in enumerate(tgt_vocab)}

tgt_vocab_size = len(tgt_vocab)


```

```
src_len = 5 # enc_input max sequence length

tgt_len = 6 # dec_input(=dec_output) max sequence length



def make_data(sentences):

    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):

  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_output
    s), 2, True)
```

模型参数

下面变量代表的含义依次是

1. 字嵌入 & 位置嵌入的维度，这俩值是相同的，因此用一个变量就行了
2. FeedForward 层隐藏神经元个数
3. Q、K、V 向量的维度，其中 Q 与 K 的维度必须相等，V 的维度没有限制，不过为了方便起见，我都设为 64
4. Encoder 和 Decoder 的个数
5. 多头注意力中 head 的数量

```
# Transformer Parameters
d_model = 512  # Embedding Size
d_ff = 2048 # FeedForward dimension
d_k = d_v = 64  # dimension of K(=Q), V
n_layers = 6  # number of Encoder of Decoder Layer
n_heads = 8  # number of heads in Multi-Head Attention
```

上面都比较简单，下面开始涉及到模型就比较复杂了，因此我会将模型拆分成以下几个部分进行讲解

- Positional Encoding
- Pad Mask（针对句子不够长，加了 pad，因此需要对 pad 进行 mask）
- Subsequence Mask（Decoder input 不能看到未来时刻单词信息，因此需要 mask）
- ScaledDotProductAttention（计算 context vector）
- Multi-Head Attention
- FeedForward Layer
- Encoder Layer
- Encoder
- Decoder Layer
- Decoder
- Transformer

关于代码中的注释，如果值为 `src_len` 或者 `tgt_len` 的，我一定会写清楚，但是有些函数或者类，Encoder 和 Decoder 都有可能调用，因此就不能确定究竟是 `src_len` 还是 `tgt_len`，对于不确定的，我会记作 `seq_len`

Positional Encoding

```
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)
```

Pad Mask

```
def get_attn_pad_mask(seq_q, seq_k):
    '''
    seq_q: [batch_size, seq_len]
    seq_k: [batch_size, seq_len]
    seq_len could be src_len or it could be tgt_len
    seq_len in seq_q and seq_len in seq_k maybe not equal
    '''
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # eq(zero) is PAD token

    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_siz
    e, len_q, len_k]
```

由于在 Encoder 和 Decoder 中都需要进行 mask 操作，因此就无法确定这个函数的参数中 `seq_len` 的值，如果是在 Encoder 中调用的， `seq_len` 就等于 `src_len`；如果是在 Decoder 中调用的， `seq_len` 就有可能等于 `src_len`，也有可能等于 `tgt_len` （因为 Decoder 有两次 mask）

这个函数最核心的一句代码是 `seq_k.data.eq(0)`，这句的作用是返回一个大小和 `seq_k` 一样的 tensor，只不过里面的值只有 True 和 False。如果 `seq_k` 某个位置的值等于 0，那么对应位置就是 True，否则即为 False。举个例子，输入为 `seq_data = [1, 2, 3, 4, 0]`， `seq_data.data.eq(0)` 就会返回 `[False, False, False, False, True]`

剩下的代码主要是扩展维度，**强烈建议读者打印出来，看看最终返回的数据是什么样子**

Subsequence Mask

```
    def get_attn_subsequence_mask(seq):
        '''
        seq: [batch_size, tgt_len]
        '''
        attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
        subsequence_mask = np.triu(np.ones(attn_shape), k=1) # Upper triangu
    lar matrix
        subsequence_mask = torch.from_numpy(subsequence_mask).byte()
        return subsequence_mask # [batch_size, tgt_len, tgt_len]
```

Subsequence Mask 只有 Decoder 会用到，主要作用是屏蔽未来时刻单词的信息。首先通过 `np.ones()` 生成一个全 1 的方阵，然后通过 `np.triu()` 生成一个上三角矩阵，下图是 `np.triu()` 用法

```
#k=0表示正常的上三角矩阵
upper_triangle = np.triu(data, 0)
[[1 2 3 4 5]
 [0 5 6 7 8]
 [0 0 7 8 9]
 [0 0 0 7 8]
 [0 0 0 0 5]]
#k=-1表示对角线的位置下移1个对角线
upper_triangle = np.triu(data, -1)
[[1 2 3 4 5]
 [4 5 6 7 8]
 [0 7 7 8 9]
 [0 0 6 7 8]
 [0 0 0 4 5]]
#k=1表示对角线的位置上移1个对角线
upper_triangle = np.triu(data, 1)
[[0 2 3 4 5]
 [0 0 6 7 8]
 [0 0 0 8 9]
 [0 0 0 0 8]
 [0 0 0 0 0]]
```

ScaledDotProductAttention

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaledDotProductAttention, self).__init__()

    def forward(self, Q, K, V, attn_mask):
        '''
        Q: [batch_size, n_heads, len_q, d_k]
        K: [batch_size, n_heads, len_k, d_k]
        V: [batch_size, n_heads, len_v(=len_k), d_v]
        attn_mask: [batch_size, n_heads, seq_len, seq_len]

        '''

        scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k) #
scores : [batch_size, n_heads, len_q, len_k]
        scores.masked_fill_(attn_mask, -1e9) # Fills elements of self t
ensor with value where mask is True.
```

```
1
4
1           attn = nn.Softmax(dim=-1)(scores)
5
1           context = torch.matmul(attn, V) # [batch_size, n_heads, len_q,
6   d_v]
1           return context, attn
7
```

这里要做的是，通过 Q 和 K 计算出 scores，然后将 scores 和 V 相乘，得到每个单词的 context vector

第一步是将 Q 和 K 的转置相乘没什么好说的，相乘之后得到的 scores 还不能立刻进行 softmax，需要和 attn_mask 相加，把一些需要屏蔽的信息屏蔽掉， attn_mask 是一个仅由 True 和 False 组成的 tensor，并且一定会保证 attn_mask 和 scores 的维度四个值相同（不然无法做对应位置相加）

mask 完了之后，就可以对 scores 进行 softmax 了。然后再与 V 相乘，得到 context

MultiHeadAttention

```
1   class MultiHeadAttention(nn.Module):
2       def __init__(self):
3           super(MultiHeadAttention, self).__init__()
4           self.W_Q = nn.Linear(d_model, d_k * n_heads, bias=False)
5           self.W_K = nn.Linear(d_model, d_k * n_heads, bias=False)
6           self.W_V = nn.Linear(d_model, d_v * n_heads, bias=False)
7           self.fc = nn.Linear(n_heads * d_v, d_model, bias=False)
8       def forward(self, input_Q, input_K, input_V, attn_mask):
9           '''
1           input_Q: [batch_size, len_q, d_model]
0
1           input_K: [batch_size, len_k, d_model]
1
1           input_V: [batch_size, len_v(=len_k), d_model]
2
1           attn_mask: [batch_size, seq_len, seq_len]
3
1           '''
4
1           residual, batch_size = input_Q, input_Q.size(0)
5
1           # (B, S, D) -proj-> (B, S, D_new) -split-> (B, S, H, W) -trans-
6   > (B, H, S, W)
1           Q = self.W_Q(input_Q).view(batch_size, -1, n_heads, d_k).transp
7   ose(1,2)  # Q: [batch_size, n_heads, len_q, d_k]
1           K = self.W_K(input_K).view(batch_size, -1, n_heads, d_k).transp
8   ose(1,2)  # K: [batch_size, n_heads, len_k, d_k]
1           V = self.W_V(input_V).view(batch_size, -1, n_heads, d_v).transp
9   ose(1,2)  # V: [batch_size, n_heads, len_v(=len_k), d_v]
```

```
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1) # attn_mask : [batch_size, n_heads, seq_len, seq_len]

        # 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)

        context = context.transpose(1, 2).reshape(batch_size, -1, n_heads * d_v) # context: [batch_size, len_q, n_heads * d_v]
        output = self.fc(context) # [batch_size, len_q, d_model]

        return nn.LayerNorm(d_model).cuda()(output + residual), attn
```

完整代码中一定会有三处地方调用 `MultiHeadAttention()`，Encoder Layer 调用一次，传入的 `input_Q`、`input_K`、`input_V` 全部都是 `enc_inputs`；Decoder Layer 中两次调用，第一次传入的全是 `dec_inputs`，第二次传入的分别是 `dec_outputs`，`enc_outputs`，`enc_outputs`

FeedForward Layer

```
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):
        '''

        inputs: [batch_size, seq_len, d_model]

        '''

        residual = inputs

        output = self.fc(inputs)

        return nn.LayerNorm(d_model).cuda()(output + residual) # [batch_size, seq_len, d_model]
```

这段代码非常简单，就是做两次线性变换，残差连接后再跟一个 Layer Norm

Encoder Layer

```
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_inputs: [batch_size, src_len, d_model]
        enc_self_attn_mask: [batch_size, src_len, src_len]

        '''

        # enc_outputs: [batch_size, src_len, d_model], attn: [batch_size, n_heads, src_len, src_len]
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs, enc_self_attn_mask) # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs) # enc_outputs: [batch_size, src_len, d_model]
        return enc_outputs, attn

```

将上述组件拼起来，就是一个完整的 Encoder Layer

Encoder

```
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_inputs: [batch_size, src_len]

        '''

        enc_outputs = self.src_emb(enc_inputs) # [batch_size, src_len, d_model]
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1) # [batch_size, src_len, d_model]
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs) # [batch_size, src_len, src_len]
        enc_self_attns = []

```

```
        for layer in self.layers:

            # enc_outputs: [batch_size, src_len, d_model], enc_self_attn: [batch_size, n_heads, src_len, src_len]
            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
```

使用 `nn.ModuleList()` 里面的参数是列表，列表里面存了 `n_layers` 个 Encoder Layer

由于我们控制好了 Encoder Layer 的输入和输出维度相同，所以可以直接用个 for 循环以嵌套的方式，将上一次 Encoder Layer 的输出作为下一次 Encoder Layer 的输入

Decoder Layer

```
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_inputs: [batch_size, tgt_len, d_model]

        enc_outputs: [batch_size, src_len, d_model]

        dec_self_attn_mask: [batch_size, tgt_len, tgt_len]

        dec_enc_attn_mask: [batch_size, tgt_len, src_len]

        '''

        # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len]
        dec_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs, dec_inputs, dec_self_attn_mask)
        # dec_outputs: [batch_size, tgt_len, d_model], dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]
        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) # [batch_size, tgt_len, d_model]
```

```
        return dec_outputs, dec_self_attn, dec_enc_attn
```

在 Decoder Layer 中会调用两次 `MultiHeadAttention` ，第一次是计算 Decoder Input 的 self-attention，得到输出 `dec_outputs` 。然后将 `dec_outputs` 作为生成 Q 的元素， `enc_outputs` 作为生成 K 和 V 的元素，再调用一次 `MultiHeadAttention` ，得到的是 Encoder 和 Decoder Layer 之间的 context vector。最后将 `dec_outptus` 做一次维度变换，然后返回

Decoder

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([DecoderLayer() for _ in range(n_layers)])

    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        '''
        dec_inputs: [batch_size, tgt_len]

        enc_intpus: [batch_size, src_len]

        enc_outputs: [batch_size, src_len, d_model]

        '''

        dec_outputs = self.tgt_emb(dec_inputs) # [batch_size, tgt_len, d_model]
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1).cuda() # [batch_size, tgt_len, d_model]
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs).cuda() # [batch_size, tgt_len, tgt_len]
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs).cuda() # [batch_size, tgt_len, tgt_len]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask), 0).cuda() # [batch_size, tgt_len, tgt_len]


        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]
            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

```

Decoder 中不仅要把 "pad"mask 掉，还要 mask 未来时刻的信息，因此就有了下面这三行代码，其中 `torch.gt(a, value)` 的意思是，将 a 中各个位置上的元素和 value 比较，若大于 value，则该位置取 1，否则取 0

```
dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs) # [batch_size, tgt_len, tgt_len]
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs) # [batch_size, tgt_len, tgt_len]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask), 0) # [batch_size, tgt_len, tgt_len]
```

Transformer

```
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder().cuda()
        self.decoder = Decoder().cuda()
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False).cuda()
    def forward(self, enc_inputs, dec_inputs):
        '''
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]

        '''

        # tensor to store decoder outputs

        # outputs = torch.zeros(batch_size, tgt_len, tgt_vocab_size).to(self.device)


        # enc_outputs: [batch_size, src_len, d_model], enc_self_attns: [n_layers, batch_size, n_heads, src_len, src_len]
```

```
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)

        # dec_outpus: [batch_size, tgt_len, d_model], dec_self_attns: [n_layers, batch_size, n_heads, tgt_len, tgt_len], dec_enc_attn: [n_layers, batch_size, tgt_len, src_len]
        dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
        dec_logits = self.projection(dec_outputs) # dec_logits: [batch_size, tgt_len, tgt_vocab_size]
        return dec_logits.view(-1, dec_logits.size(-1)), enc_self_attns, dec_self_attns, dec_enc_attns
```

Transformer 主要就是调用 Encoder 和 Decoder。最后返回 `dec_logits` 的维度是 [batch_size * tgt_len, tgt_vocab_size]，可以理解为，一个句子，这个句子有 batch_size*tgt_len 个单词，每个单词有 tgt_vocab_size 种情况，取概率最大者

模型 & 损失函数 & 优化器

```
model = Transformer().cuda()
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.99)
```

这里的损失函数里面我设置了一个参数 `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），关于这个参数更详细的说明，可以看我这篇文章的最下面，稍微提了一下

训练

```
for epoch in range(30):
    for enc_inputs, dec_inputs, dec_outputs in loader:
      '''
      enc_inputs: [batch_size, src_len]
      dec_inputs: [batch_size, tgt_len]
      dec_outputs: [batch_size, tgt_len]
      '''
      enc_inputs, dec_inputs, dec_outputs = enc_inputs.cuda(), dec_inputs.cuda(), dec_outputs.cuda()
      # outputs: [batch_size * tgt_len, tgt_vocab_size]
      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()
```

测试

```
def greedy_decoder(model, enc_input, start_symbol):
    """
    For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know the
    target sequence input. Therefore we try to generate the target input word by word, then feed it into the transformer.
    Starting Reference: http://nlp.seas.harvard.edu/2018/04/03/attention.html#greedy-decoding
    :param model: Transformer Model
    :param enc_input: The encoder input
    :param start_symbol: The start symbol. In this example it is 'S' which corresponds to index 4
    :return: The target input
    """

    enc_outputs, enc_self_attns = model.encoder(enc_input)

    dec_input = torch.zeros(1, 0).type_as(enc_input.data)

    terminal = False

    next_symbol = start_symbol

    while not terminal:

        dec_input = torch.cat([dec_input.detach(),torch.tensor([[next_symbol]],dtype=enc_input.dtype).cuda()],-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]

        next_word = prob.data[-1]

        next_symbol = next_word

        if next_symbol == tgt_vocab["."]:

            terminal = True

```

```
            print(next_word)

        return dec_input


# Test

enc_inputs, _, _ = next(iter(loader))

enc_inputs = enc_inputs.cuda()

for i in range(len(enc_inputs)):

    greedy_dec_input = greedy_decoder(model, enc_inputs[i].view(1, -1),
start_symbol=tgt_vocab["S"])
    predict, _, _, _ = model(enc_inputs[i].view(1, -1), greedy_dec_inpu
t)
    predict = predict.data.max(1, keepdim=True)[1]

    print(enc_inputs[i], '->', [idx2word[n.item()] for n in predict.squ
eeze()])
```

> 测试部分代码由网友 qq2014 提供，在此表示感谢

最后给出完整代码链接（需要科学的力量）
Github 项目地址：nlp-tutorial

最后编辑于: 2022 年 03 月 20 日

返回文章列表 打赏