使用Transformer模型搭建机器翻译项目

一、项目介绍

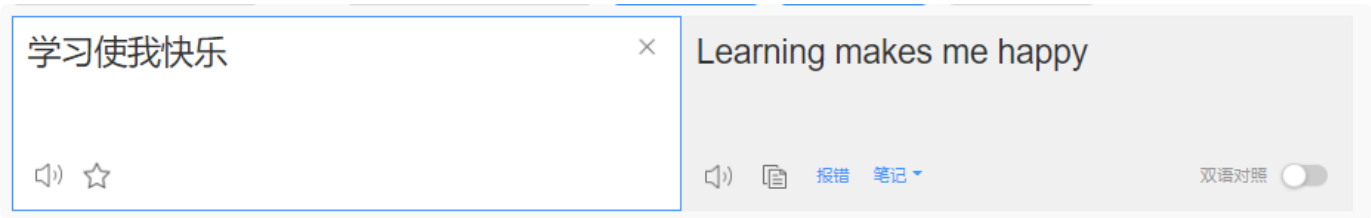
本项目是使用transformer框架实现机器翻译，将英文句子翻译成中文句子。在其中加入了一些更加有效的策略来使模型达到更好的效果。比如**BPE分词**、**Warm Up**、**Label Smoothing**、**Beam Search**等。

本项目采用的是自己复现的transformer框架，transformer理论知识学习参考博文Transformer 模型详解，代码参考Harvard开源的transformer-pytorch的机器翻译模型，引用的一些github代码已在下文对应位置标出。因为是小白，在编写过程中需要查很多的资料，也踩了不少的坑，不足之处还请大佬指出。下面我把项目的大体流程以及主要思想写在下面，也当做自己的一个学习心得。

本文中所列出的代码只为关键代码，完整的项目代码请查看Github：

二、机器翻译

机器翻译，又称为自动翻译，是利用计算机将一种自然语言(源语言)转换为另一种自然语言(目标语言)的过程。它是计算语言学的一个分支，是人工智能的终极目标之一，具有重要的科学研究价值。（来自百度百科）

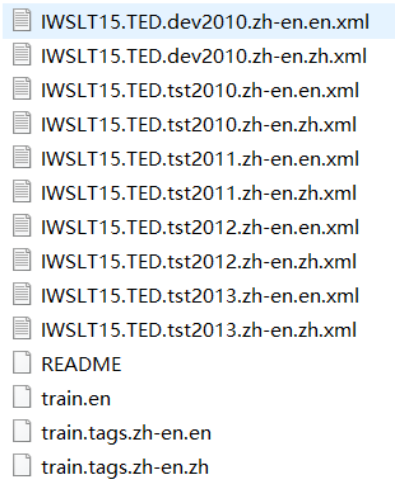


机器翻译的发展有基于规则的机器翻译到基于统计的机器翻译再到基于神经网络的机器翻译，机器翻译的模型与方法一直在发生改变。现在主流的机器翻译模型是transformer模型，和NMT一样依然采用的是编码加解码的结构，但他的翻译表现却比RNN优秀不少。

三、数据部分

1、数据集介绍

数据集使用的是**IWSLT 2015**，数据集的下载地址为： https://wit3.fbk.eu/2015-01 链接里面还有其他更多的数据，如果有时间和精力去尝试更多数据，最终的效果会更好。



因为本项目是实现中文翻译英文的任务，所以我们选中其中的中文和英文数据进行训练。另外，由于本人计算资源不足，无法完成大规模数据的训练工作，我只截取了一部分数据当做训练集。

2、数据预处理

（1）数据格式化

因为此数据集是在网页中提取出来的，会有一些网页标签，我们要先删掉含有网页标签的数据，并只截取80000条左右数据进行训练。

```
<url>http://www.ted.com/talks/david_gallo_on_life_in_the_deep_
<keywords>talks, TED Conference, animals, geology, life, ocean
<speaker>David Gallo</speaker>
<talkid>343</talkid>
<title>深海中的生命－大卫.盖罗</title>
<description>大卫.盖罗通过潜水艇拍下的影片把我们带到了地球最黑暗，最险恶同
大卫.盖罗：这位是比尔.兰格， 我是大卫.盖罗。
```

```python
def filter_and_truncation_data(src_path, tgt_path):
    f1 = open(src_path,'r')
    f2 = open(tgt_path,'r')
    fw1 = open(src_path+".txt",'w')
    fw2 = open(tgt_path+".txt",'w')
    for i, line1,line2 in tqdm(enumerate(zip(f1.readlines(),f2.readlines()))):
        line1 = line1.strip()
        line2 = line2.strip()
        if line1 and line2:
            if '<' not in line1 and '>' not in line1 and '<' not in line2 and '>' not in line2:
                if i < 100000:
                    fw1.write(line1+"\n")
                    fw2.write(line2+"\n")
    fw1.close()
    f1.close()
    fw2.close()
    f2.close()
```

下面处理验证集和测试集：

可以看到验证集和测试集的数据都是以xml形式给出，我们需要将他读取出来保存为txt格式。

```python
tree_source_dev = ET.parse('zh-en/IWSLT15.TED.dev2010.zh-en.zh.xml')
tree_source_dev = [seg.text for seg in tree_source_dev.iter('seg')]
with open('dev_cn.txt','w') as f:
    for item in tree_source_dev:
        f.write(item+'\n')
```

（2）BPE分词

BPE的基本思路是将使用最频繁的字节用一个新的组合来代替。更多的BPE理论不做过多陈述。

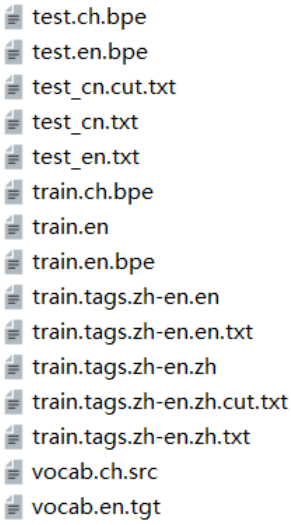
中文需要先进行jieba分词，在进行BPE分词，英文只需要进行BPE分词

```python
def jieba_cut(in_file,out_file):
    out_f = open(out_file,'w',encoding='utf8')
    with open(in_file,'r',encoding='utf8') as f:
        for line in f.readlines():
            line = line.strip()
            cut_line = ' '.join(jieba.cut(line)) # 分词
            out_f.write(cut_line+'\n')
    out_f.close()
```

下面进行BPE分词

```python
!subword-nmt learn-bpe -s 32000 < zh-en/train.tags.zh-en.zh.cut.txt > zh-en/bpe.ch.32000
!subword-nmt learn-bpe -s 32000 < zh-en/train.tags.zh-en.en.txt > zh-en/bpe.en.32000
```

对测试集和验证集也同样处理，在BPE分词后也会生成词表文件。



分词后的部分文件展示

另外。也可使用更加简便的方法对BPE进行分词，可以使用python的一个库sentencepiece来实现BPE分词操作。此方法不用先对中文、日文等一些词与词之间没有间隔的语言进行jieba分词，它可以将所有的字符编码成Unicode编码，通过训练直接将原始数据变为切分后的文本。

英文的词表和中文的词表的大小我们都采用32000，有研究表明，这样可以覆盖大多数的bpe组合，如果再扩大词表，会增加训练负担。en_character_coverage英文一般设置成1，中文一般设置成0.995。

（3）读取数据

```python
    def get_dataset(self, ch_data_path, en_data_path, sort=False):
        out_en_sent = []
        out_cn_sent = []
        with open(ch_data_path, 'r') as f:
            for ids, row in enumerate(f.readlines()):
                out_cn_sent.append(row.strip())
        with open(en_data_path, 'r') as f:
            for ids, row in enumerate(f.readlines()):
                out_en_sent.append(row.strip())
        if sort:
            sorted_index = self.len_argsort(out_en_sent) # 按照英文句子的长度进行排序
            out_en_sent = [out_en_sent[i] for i in sorted_index]
            out_cn_sent = [out_cn_sent[i] for i in sorted_index]
        return out_en_sent, out_cn_sent
```

Python | 复制代码

（4）生成数据对象

本阶段的主要目的是：

a. 将分词参照词表转化成对应的编号，并在句子的开头和结尾加上特定的标识

b. 将句子按照句子长度进行排序，对于长度不够的填充0

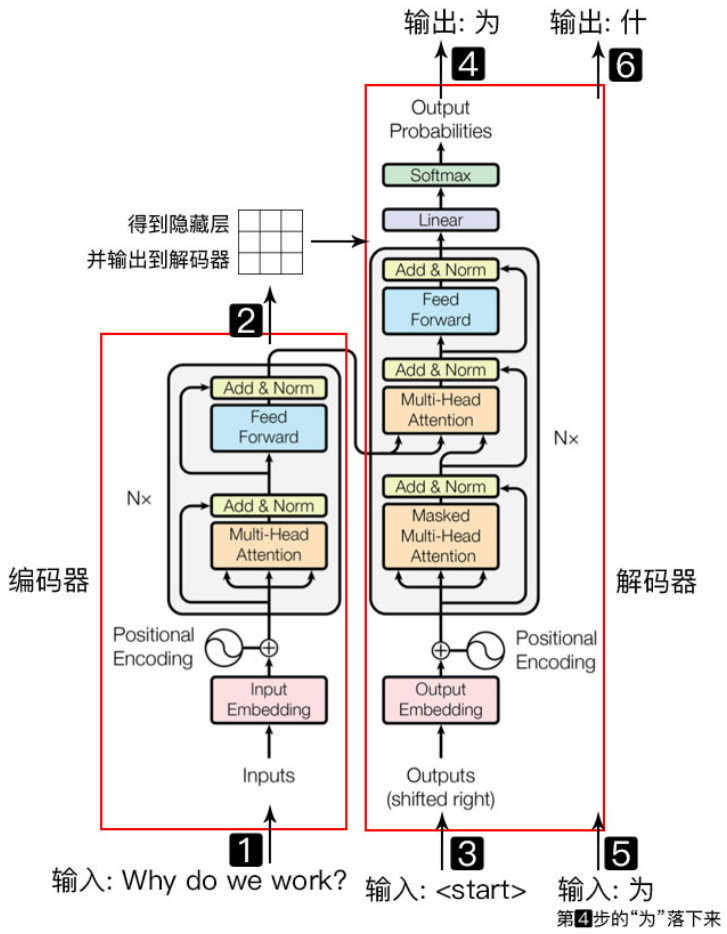
```python
class DatasetObj(Dataset):
    def __init__(self, ch_data_path, en_data_path):
        # 中英文原始句子
        self.out_en_sent, self.out_cn_sent = self.get_dataset(ch_data_path, en_data_path, sort=True)
        # 切好的句子
        self.sp_eng = get_en_tokenizer()
        self.sp_chn = get_ch_tokenizer()
        # 一些特殊符号
        self.PAD = self.sp_eng.pad_id()  # 0
        self.BOS = self.sp_eng.bos_id()  # 2
        self.EOS = self.sp_eng.eos_id()  # 3
    # 把句子转成编号，并加上开始和结束标志
    def collate_fn(self, batch):
        src_text = [x[0] for x in batch] # 所有的英语句子
        tgt_text = [x[1] for x in batch] # 所有的中文句子

        # 将字符串 编码为 id， 并加上开始和结束标志
        src_tokens = [[self.BOS] + self.sp_eng.EncodeAsIds(sent) + [self.EOS] for sent in src_text]
        tgt_tokens = [[self.BOS] + self.sp_chn.EncodeAsIds(sent) + [self.EOS] for sent in tgt_text]

        # 转化为 tensor 并 填充 0
        batch_input = pad_sequence([torch.LongTensor(np.array(l_)) for l_ in src_tokens],
                                   batch_first=True, padding_value=self.PAD)
        batch_target = pad_sequence([torch.LongTensor(np.array(l_)) for l_ in tgt_tokens],
                                    batch_first=True, padding_value=self.PAD)

        return Batch(src_text, tgt_text, batch_input, batch_target, self.PAD)
```

三、建立模型



1、位置编码

因为embedding后的数据并没有包含词间的位置信息，需要加上位置编码。

位置编码公式：

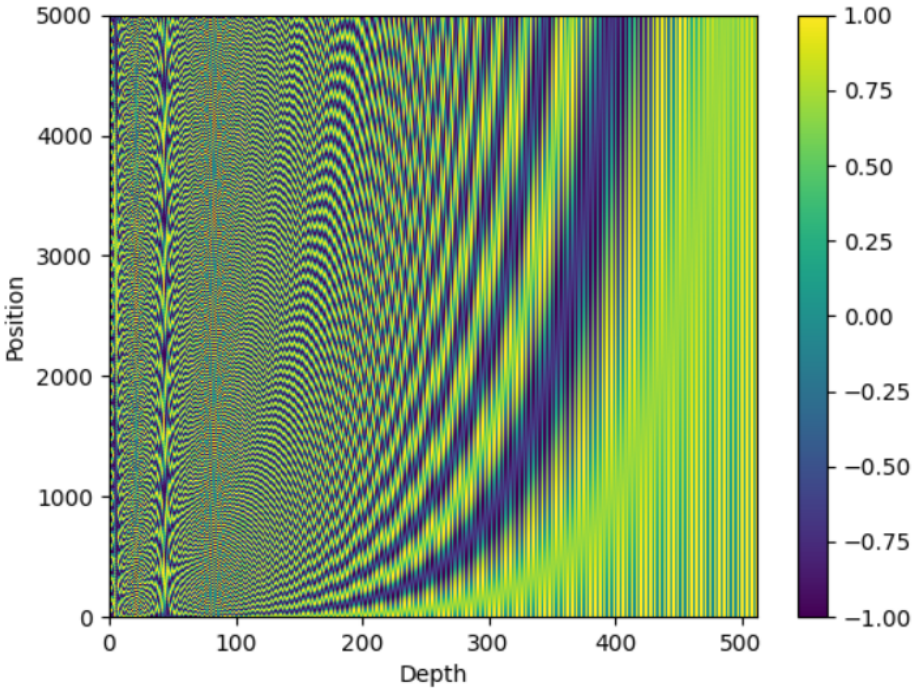
$$\mathrm{PE}_{(\mathrm{pos},2\mathrm{i})} = \sin(\mathrm{pos}/10000^{2\mathrm{i}/\mathrm{d}_{\mathrm{model}}})$$
$$\mathrm{PE}_{(\mathrm{pos},2\mathrm{i}+1)} = \cos(\mathrm{pos}/10000^{2\mathrm{i}/\mathrm{d}_{\mathrm{model}}})$$

Python | 复制代码

```python
# 进行位置编码
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)
        pos_encoding = 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))
        pos_encoding[:, 0::2] = torch.sin(position * div_term)
        pos_encoding[:, 1::2] = torch.cos(position * div_term)
        pos_encoding = pos_encoding.unsqueeze(0) # 将pe矩阵以持久的buffer状态存下
        self.register_buffer('pe', pos_encoding)
        # print("pe.shape: ", pe.shape) # [1, 100, 512]
        self.pos_encoding = pos_encoding

    def forward(self, x):
        # x: [seq_len, batch_size, d_model]
        # 输入的是已经embedding编码后的数据，输出的是在输入的基础上加上位置编码的结果
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)

```



2、两种mask

（1）Padding Mask：

将句子中被填充为0的部分Mask掉，即句子中为0的部分mask值为True。此mask在encoder和decoder部分都用的到。

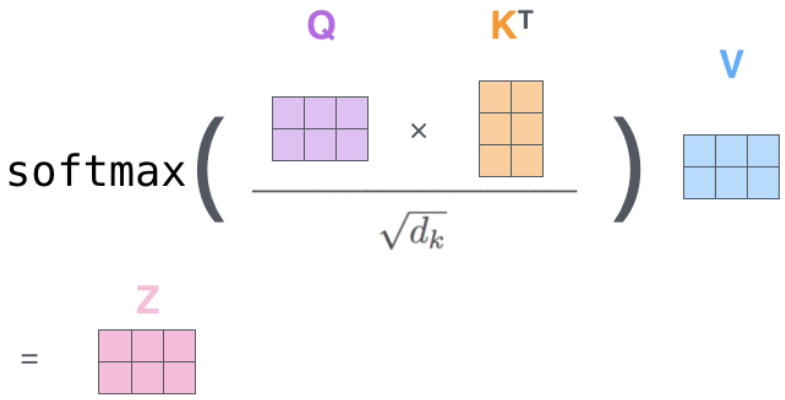
```python
def get_attn_pad_mask(seq):
    pad_attn_mask = seq.data.eq(0).unsqueeze(1)  # [batch_size, 1, len_k], False is masked
    return pad_attn_mask.expand(pad_attn_mask.shape[0], pad_attn_mask.shape[-1], pad_attn_mask.shape[-1])  # [batch_size, len_q, len_k]
```

（2）Subsequence Mask：

Subsequence Mask 只有 Decoder 会用到，主要作用是屏蔽未来时刻单词的信息。在穿传统的Seq2Seq中Decoder使用的是RNN模型，RNN模型是一个时间序列模型，在当前时刻是无法看到后面的词的，但Transformer没有使用RNN，这就导致在训练过程中将整个句子都暴露在Decoder中，这显然不合理。所以需要模仿着RNN的方式，每次只增加一个单词的视野，解决办法就是使用一个上三角矩阵作为mask。

```python
def get_attn_subsequence_mask(seq):
    # seq: [batch_size, tgt_len]
    attn_shape = [seq.size(1), seq.size(1)]
    subsequence_mask = np.triu(np.ones(attn_shape), k=1) # 上三角矩阵
    subsequence_mask = torch.from_numpy(subsequence_mask).byte()
    return subsequence_mask # [batch_size, tgt_len, tgt_len]
```

3、生成注意力矩阵



根据上面的公式，先让Q和K的转置相乘再除根号d_k的操作得到scores，然后需要将需要屏蔽的信息mask掉。之后进行SoftMax，然后与v相乘得到Contest。

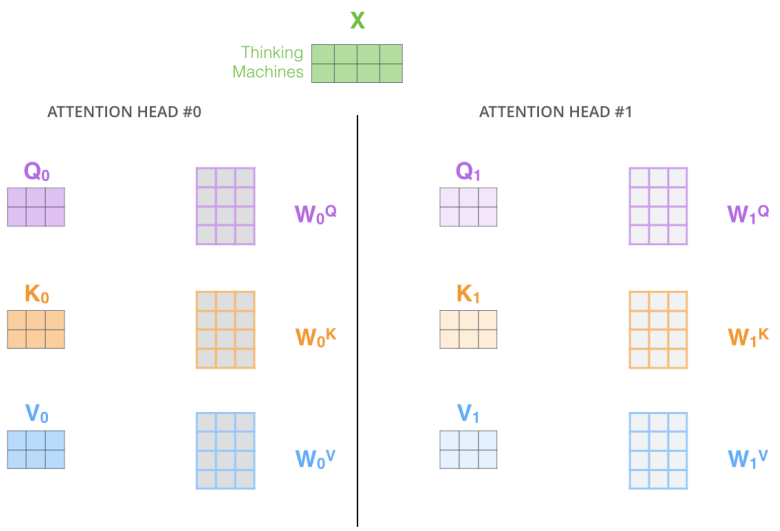
mask=True的位置是pad或者future token，如果按照以往的pad的值0的话，在进行softMax后e的0次幂是1，所以被mask掉的数据又参与了矩阵运算，所以需要乘以-1e9使之成为负无穷，经过softmax后会趋于0。起到了屏蔽的作用。

Python | 复制代码

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

    def forward(self, Q, K, V, mask=None, dropout=None):
        d_k = Q.shape[-1]
        scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k)  # scores : [batch_size, n_heads, len_q, len_k]
        if mask != None:
            scores.masked_fill_(mask, -1e9)  # mask值为True的位置换成 -1e9,
        attn = nn.Softmax(dim=-1)(scores)
        if dropout is not None:
            attn = dropout(attn)
        context = torch.matmul(attn, V)  # [batch_size, n_heads, len_q, d_v]
        return context, attn  # 最后返回注意力矩阵跟value的乘积，以及注意力矩阵
```

4、多头注意力机制



MultiHeadedAttention要经过四个步骤：

- linear layer 将 linear layer的结果分到到不同的头中
- 分别进行自注意力机制
- 将所有头的结果拼接起来

- 之后再进行一次全连接

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, d_model, h, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        ...
        self.L_Q = nn.Linear(d_model, d_model, bias=False)
        self.L_K = nn.Linear(d_model, d_model, bias=False)
        self.L_V = nn.Linear(d_model, d_model, bias=False)
        self.fc = nn.Linear(d_model, d_model, bias=False)

    def forward(self, query, key, value, mask=None):
        residual = query  # 保存输入 用作残差
        if mask is not None:
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)  # Q: [batch_size, n_heads, len_q, d_k]
        query = self.L_Q(query).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
        key = self.L_K(key).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
        value = self.L_V(value).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
        # 调用上述定义的attention函数计算得到h个注意力矩阵跟value的乘积，以及注意力矩阵
        x, self.attn = Attention()(query, key, value, mask=mask, dropout=self.dropout)
        # 将h个多头注意力矩阵concat起来
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)  # x: [batch_size, n_heads,len_q*d_k]
        # 使用self.linears中构造的最后一个全连接函数来存放变换后的矩阵进行返回
        # 进行层归一化 和 残差连接
        return nn.LayerNorm(self.d_model)(self.fc(x).to(DEVICE) + residual.to(DEVICE))
```

Transformer模型中一共有三次使用到了 MultiHeadedAttention。在编码器一使用一次，在解码器中使用了两次。但是他们的输入不同。编码器中MultiHeadedAttention的k、q、v都是输入数据在进行embedding编码和位置编码后的数据。decoder第一次调用k、q、v都是解码的输入经过编码后的数据，而第二次的输入q为解码器第一次MultiHeadedAttention的输出，q和v都是编码器的输出。

5、前馈神经网络

2层线性变换和一个ReLU激活：

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

Python | 复制代码

```python
class FeedForwardNet(nn.Module):
    def __init__(self, d_model, d_ff):
        super(FeedForwardNet, self).__init__()
        self.d_model = d_model
        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, x):  # x: [batch_size, seq_len, d_model]
        output = self.fc(x)
        # 进行残差链接和层归一化
        return nn.LayerNorm(self.d_model)(output.to(DEVICE) + x.to(DEVICE)
    )  # [batch_size, seq_len, d_model]
```

6、编码器

(1) 编码层

主要包括多头注意力机制 和 前馈神经网络。

注意在执行完多头注意力机制和前馈神经网络后都要进行残差连接和层归一化，因为我把这两步合并在各自的实现过程中，所以这里没有体现。

Python | 复制代码

```python
class EncoderLayer(nn.Module):
    def __init__(self, n_heads, n_hidden, d_model=512):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadedAttention(d_model, n_heads)
        self.feed_forward = FeedForwardNet(d_model, n_hidden)

    def forward(self, x, mask):
        # 这里self_attn的输入都为 编码器的原始输入
        x = self.self_attn(x, x, x, mask)
        # 注意到attn得到的结果x直接作为了下一层的输入
        x = self.feed_forward(x)
        return x
```

(2) 编码器

编码器中包含多个编码层，一般是6个。这里就完整的体现了整个编码器的构造，输入一组句子，先进行embedding编码，然后加上位置编码，继而经过N个编码层得到编码器的输出。其中N编码层中下一个编码层的输入是上一个编码层的输出，所以要保证编码层的输入和输出有相同的尺寸。

Python | 复制代码

```python
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.src_emb = Embeddings(d_model=config.d_model, vocab=config.src_vocab_size)
        self.pos_emb = PositionalEncoding(d_model=config.d_model, dropout=0.1)
        self.layers = nn.ModuleList([copy.deepcopy(layer) for _ in range(N)]) # 复制N个encoder layer

    def forward(self, x, mask):
        x = self.src_emb(x)  # [batch_size, src_len, d_model]
        x = self.pos_emb(x)  # [batch_size, src_len, d_model]
        for layer in self.layers:
            x = layer(x, mask)
        return x
```

7、解码器

(1) 解码层

由两个多头注意力机制加一个前馈神经网络组成。这里要注意两个注意力机制的输入的不同之处。另外这两个注意力机制的mask也不同，第一个需要padding mask和Subsequence Mask，既要盖住pad部分也要盖住后面的词。第二个注意力机制只需要padding mask即可。

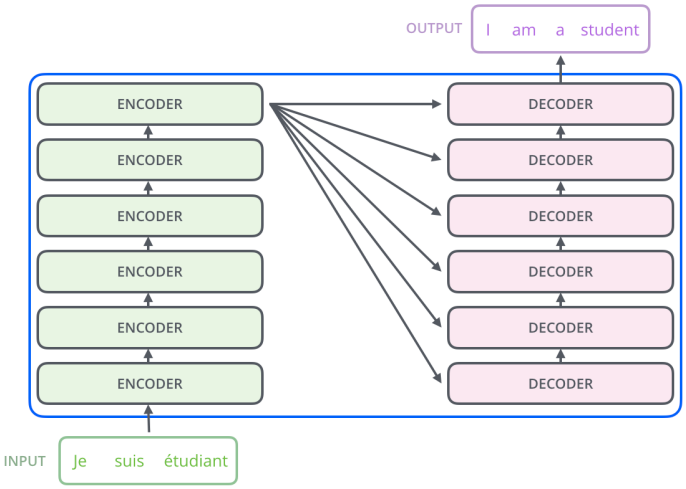
```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadedAttention(d_model, n_heads)
        self.src_attn = MultiHeadedAttention(d_model, n_heads)
        self.feed_forward = FeedForwardNet(d_model, d_ff)
    def forward(self, x, memory, src_mask, tgt_mask):
        # 存放编码器的输出
        m = memory
        # 注意输入都为解码器的输入
        x = self.self_attn(x, x, x, tgt_mask)
        # q为 解码器输入 k、v为编码器输出
        x = self.src_attn(x, m, m, src_mask)
        x = self.feed_forward(x)
        return x
```

（2）解码器

解码器中包含多个解码层，一般是6个。这里展示了整个解码器的构造，输入目标句子，先进行embedding编码，然后加上位置编码，继而经过N个解码层得到解码器的输出。此后只需要经过一个线性层将输出映射到词表大小，在进行一次SoftMax选出概率最大的那个词作为预测，即可得到预结果。得到预测结果后就可结算损失和准确率了。

```python
class Decoder(nn.Module):
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.tgt_emb = Embeddings(d_model=config.d_model, vocab=config.src_vocab_size)
        self.pos_emb = PositionalEncoding(d_model=config.d_model, dropout=0.1)
        self.layers = nn.ModuleList([copy.deepcopy(layer) for _ in range(N)])# 复制N个encoder layer
    def forward(self, x, memory, src_mask, tgt_mask):
        x = self.tgt_emb(x)  # [batch_size, tgt_len, d_model]
        x = self.pos_emb(x)  # [batch_size, tgt_len, d_model]
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return x
```

8、Transformer



将编码器和解码器合在一起。

```python
class Transformer(nn.Module):
    def __init__(self, encoder, decoder, generator):
        super(Transformer, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.generator = generator

    def encode(self, src, src_mask):
        return self.encoder(src, src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(tgt, memory, src_mask, tgt_mask)

    def forward(self, src, tgt, src_mask, tgt_mask):
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)
```

四、训练

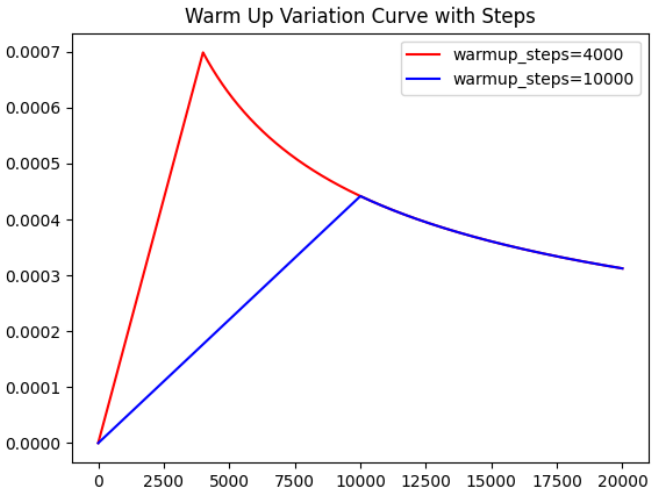
1、Warm Up

使用Warm Up策略来自适应优化器的学习率，Warm Up在一开始使用比较小的学习率，随着训练次数的增加，学习率也逐渐的增加，直到预先设置好的阈值步数停止增加，随后逐渐缓慢的减小。学习率随步数变化的公式如下：

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

```python
optimizer = torch.optim.Adam(model.parameters(), lr=config.lr)
scheduler = LambdaLR(optimizer, lr_lambda=lambda epoch: (d_model ** (-0.5)
* min((epoch + 1) ** (-0.5), (epoch + 1) * warmup_steps ** (-1.5))))
```

图像为：



2、Label Smoothing（标签平滑）

对于一个数据集来说，对于数据集标签的标注，不同人可能会有不同的标注结果，或者人工标注难免会有些标注错误，如果模型过分的相信数据的的标签，可能会导致系统过拟合。

可以使用标签平滑的策略来解决这一问题，基本思想是降低对标签的信任。公式如下：

$$q_i = \begin{cases} 1-\varepsilon & \text{if } i = y \\ \varepsilon/(K-1) & \text{otherwise} \end{cases}$$

就是说在预测正确时将概率降低为1- ε，预测错误时将概率提高为 ε/(K-1)。这样可以降低对label的信任，减小过拟合的发生，是一种正则化方法。

> 此方法代码实现来自：hemingkx 的github

```python
class LabelSmoothing(nn.Module):
    def __init__(self, tgt_vocab_size, padding_idx, smoothing=0.0):
        ...
    def forward(self, x, target):
        assert x.size(1) == self.tgt_vocab_size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.tgt_vocab_size - 2))  # 将true_dist所有的值都填充为
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, Variable(true_dist, requires_grad=False))
```

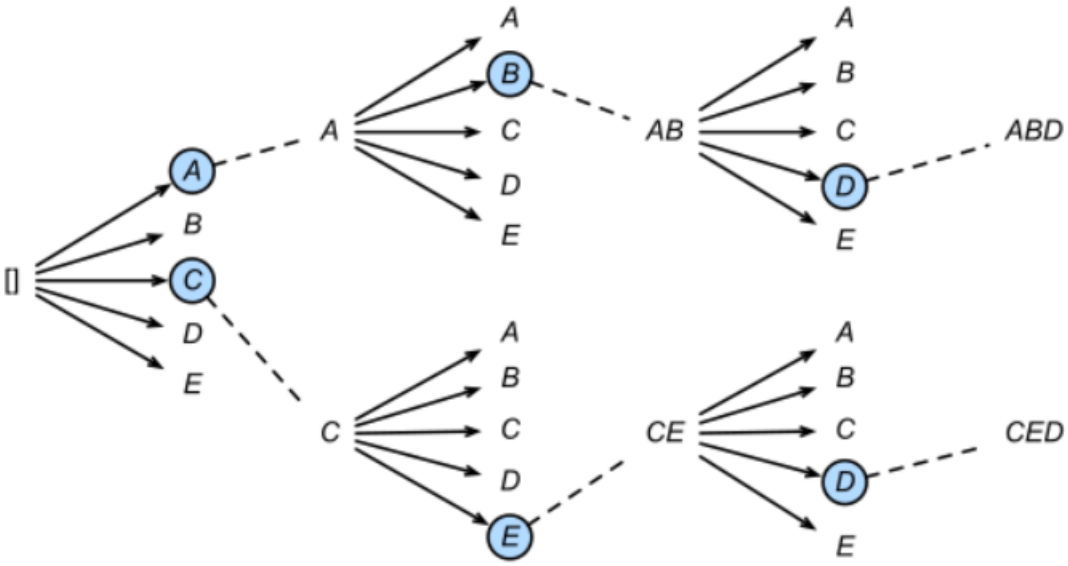
3、Greedy Search

每一个时间步都取出一个条件概率最大的输出，再将从开始到当前步的结果作为输入去获得下一个时间步的输出，直到模型给出生成结束的标志。 但此策略由于丢弃了绝大多数的可能解，这种关注当下的策略无法保证最终得到的序列概率是最优的。

```python
def greedy_search(model, src, src_mask, max_len=64):
    # 先对输入的内容进行编码
    memory = model.encode(src, src_mask)
    # 因为一开始还没有输出的内容，所以解码器输入的只有开始符
    predict = torch.ones(1, 1).fill_(2).type_as(src.data)
    for i in range(max_len - 1):
        # 一次次的执行解码操作
        out = model.decode(memory, src_mask, Variable(predict), Variable(subsequent_mask(ys.size(1)).type_as(src.data)))
        prob = model.generator(out[:, -1])
        # 根据解码器的输出获得最大概率的预测词id
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        if next_word == 3: # 遇到结束符就停止吧。
            break
        # 将当前位置预测的字符id与之前的预测内容拼接起来，作为下一次解码的输入。
        predict = torch.cat([predict, torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return predict
```

4、Beam Search

集束搜索本质上也是贪心的思想，只不过它考虑了更多的候选搜索空间，因此可以得到更多的翻译结果。在每一个时间步，不再只保留当前分数最高的1个输出，而是保留num_beams个。当num_beams=1时集束搜索就退化成了贪心搜索。



Beam Search示意图

> 此方法代码实现来自：hemingkx 的github，详情请看beam_decoder.py文件

5、训练

Python | 复制代码

```python
def train(train_data, dev_data, model, criterion, optimizer):
    best_bleu_score = 0
    save_dir = "/tmp/nlp/save_models/save_self/"
    for epoch in range(args.epoch_num):
        loss_sum = 0.
        step = 0
        for step, batch in tqdm(enumerate(train_data, start=1)):
            model.train()  # 设置train mode
            optimizer.zero_grad()  # 梯度清零
            # forward
            outputs = model(batch.src, batch.trg, batch.src_mask, batch.trg_mask)
            outputs = model.generator(outputs)
            loss = criterion(outputs.contiguous().view(-1, outputs.size(-1)), batch.trg_y.contiguous().view(-1))  # loss: [batch_size * tgt_len]
            loss_sum += loss
            loss.backward()  # 反向传播计算梯度
            optimizer.step()  # 更新参数
        print("Epoch: {}, loss: {}".format(epoch, loss_sum/step))
        val_loss_sum = 0.
        val_step = 0
        for val_step, batch_dev in tqdm(enumerate(dev_data, start=1)):
            val_loss = validate_step(model, batch_dev.src, batch_dev.trg, batch_dev.trg_y, batch_dev.src_mask, batch_dev.trg_mask, criterion)
            val_loss_sum += val_loss
            # val_metric_sum += acc
        bleu_score = evaluate(dev_data, model)
        print('Epoch: {}, Dev loss: {}, Bleu Score: {}'.format(epoch, val_loss_sum/val_step, bleu_score))
        # 保存模型
        if bleu_score > best_bleu_score:
            print("-------- Save Best Model! --------")
            best_bleu_score = bleu_score
            checkpoint = save_dir + 'best_model.pth'
            model_sd = copy.deepcopy(model.state_dict())
            torch.save({
                'loss': loss_sum / step,
                'epoch': epoch,
                'net': model_sd,
                'opt': optimizer.state_dict(),
            }, checkpoint)
```

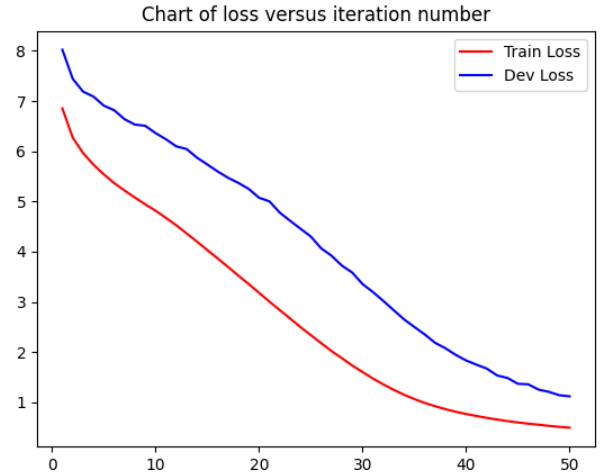
训练过程截图：

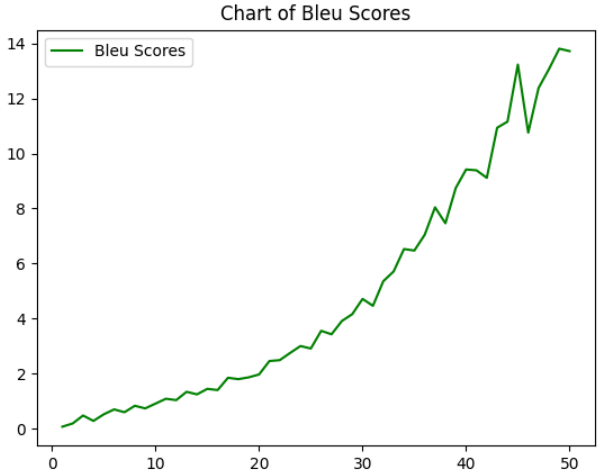
```
Epoch: 4, loss: 5.7412333488464355
100%|██████████| 56/56 [00:03<00:00, 17.40it/s]
100%|██████████| 56/56 [00:16<00:00,  3.33it/s]
Epoch: 4, Dev loss: 7.051167011260986, Bleu Score: 0.38406772104853815
-------- Save Best Model! --------
100%|██████████| 4994/4994 [05:46<00:00, 14.40it/s]
Epoch: 5, loss: 5.55543851852417
100%|██████████| 56/56 [00:03<00:00, 17.06it/s]
100%|██████████| 56/56 [00:18<00:00,  3.05it/s]
Epoch: 5, Dev loss: 6.906639575958252, Bleu Score: 0.5609590224137739
-------- Save Best Model! --------
100%|██████████| 4994/4994 [05:47<00:00, 14.37it/s]
Epoch: 6, loss: 5.377557754516602
100%|██████████| 56/56 [00:03<00:00, 17.48it/s]
100%|██████████| 56/56 [00:16<00:00,  3.42it/s]
Epoch: 6, Dev loss: 6.807255744934082, Bleu Score: 0.634057640060279
-------- Save Best Model! --------
100%|██████████| 4994/4994 [05:41<00:00, 14.63it/s]
Epoch: 7, loss: 5.21540641784668
100%|██████████| 56/56 [00:03<00:00, 17.52it/s]
100%|██████████| 56/56 [00:24<00:00,  2.26it/s]
Epoch: 7, Dev loss: 6.670889854431152, Bleu Score: 0.7625750403668318
-------- Save Best Model! --------
-------- Save Best Model! --------
100%|██████████| 4994/4994 [05:35<00:00, 14.88it/s]
Epoch: 48, loss: 0.5291149616241455
100%|██████████| 56/56 [00:03<00:00, 17.45it/s]
100%|██████████| 56/56 [00:29<00:00,  1.91it/s]
Epoch: 48, Dev loss: 1.1615221500396729, Bleu Score: 17.03287297131582
-------- Save Best Model! --------
100%|██████████| 4994/4994 [05:39<00:00, 14.71it/s]
Epoch: 49, loss: 0.5088739991188049
100%|██████████| 56/56 [00:03<00:00, 17.42it/s]
100%|██████████| 56/56 [00:24<00:00,  2.31it/s]
Epoch: 49, Dev loss: 1.1405609846115112, Bleu Score: 16.978185697631716
Early Stop Left: 9
100%|██████████| 4994/4994 [05:39<00:00, 14.72it/s]
Epoch: 50, loss: 0.4943782091140747
100%|██████████| 56/56 [00:03<00:00, 16.61it/s]
100%|██████████| 56/56 [00:26<00:00,  2.08it/s]
Epoch: 50, Dev loss: 1.0978878736495972, Bleu Score: 17.407745417487785
```

6、结果

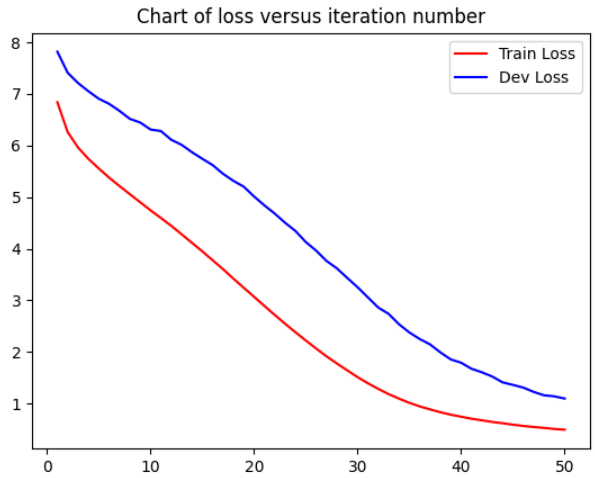
探究Beam Search的使用对结果的提升作用

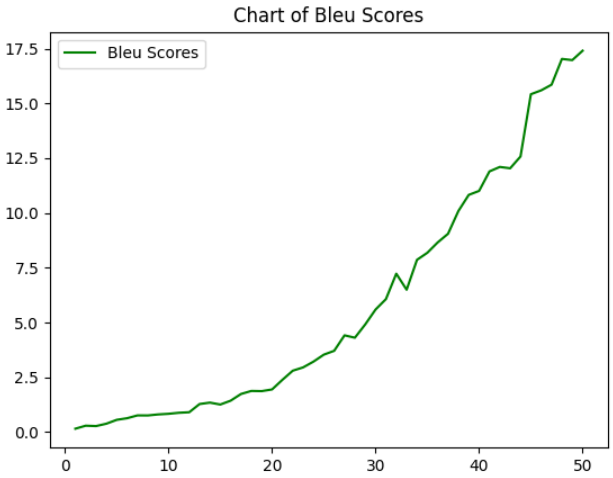
（1）使用Greedy Search的结果





（2）使用Beam Search的结果





从上图可以看出随着训练轮数的增加，无论是使用Beam Search还是Greedy Search，训练Loss和验证Loss都在逐渐的下降，但训练Loss比验证Loss下降的比较快，因为训练轮数有限，验证Loss没有降到很低，选择更大的迭代轮数可能会有更好的表现。

Blue分数大体上随着训练轮数的增加呈上升趋势，使用Greedy Search最高达到13.8，使用Beam Search最高达到17.40。由此可见，Beam Search相比于Greedy Search更有效。

五、评估

```python
# 单句预测
def translate(src, model):
    chin_tok = get_ch_tokenizer()
    with torch.no_grad():
        model.eval()
        src_mask = (src != 0).unsqueeze(-2)
        pred, _ = beam_search(model, src, src_mask, args.max_len,0, 2, 3,args.beam_size, DEVICE)
        pred = [h[0] for h in pred]
        translation = [chin_tok.decode_ids(_s) for _s in pred]
        print(translation[0])

sent = "It gets up to about 150 feet long."
# sent = " Now in our town, where the volunteers supplement a highly skilled career staff, you have to get to the fire scene pretty early to get in on any action."
# tgt_sent = " 在我们的小镇 在一个志愿者都是成功人士的地方 你必须要很早到现场 才有可能加入战况 "
model_dict = torch.load("/tmp/nlp/work/model.pth", map_location=torch.device("cpu"))
model = Transformer(args.tgt_vocab_size, args.n_layers, args.d_model, args.d_ff, args.n_heads, args.dropout).to(DEVICE)
model.load_state_dict(model_dict)
src_tokens = [[2] + get_en_tokenizer().EncodeAsIds(sent) + [3]]
batch_input = torch.LongTensor(np.array(src_tokens)).to(args.device)
translate(batch_input, model)
```

结果：

首先我们使用训练集中的数据进行测试，发现预测的翻译与标准翻译一模一样，说明模型对于训练数据的拟合效果还挺好。

```
要翻译的句子： It gets up to about 150 feet long.
标准翻译： 它可以伸展到150英尺长。
/tmp/nlp/ChineseNMT2/beam_decoder.py:58: UserWarning:
  prev_k = best_scores_id // num_words
预测翻译： 它可以伸展到150英尺长。
```

然后，使用测试集中的数据进行测试，发现预测的翻译与标准翻译的结果相差较大，说明模型的泛化能力不是很好，还需要通过调整参数，增加数据量等方式来优化。

```
要翻译的句子：  Now in our town, where the volunteers supplement a highly skilled career
标准翻译：  在我们的小镇 在一个志愿者都是成功人士的地方 你必须要很早到现场 才有可能加入战况
/tmp/nlp/ChineseNMT2/beam_decoder.py:58: UserWarning: __floordiv__ is deprecated, and
  prev_k = best_scores_id // num_words
预测翻译：  但是在我们的团队里 有能力会支持这样的能力
```