Step-by-step to Transformer：深入解析工作原理（以Pytorch机器翻译为例）

青青山螺应如是 夕小瑶的卖萌屋 2019-12-13

大家好，我是青青山螺应如是，大家可以叫我**青青**，工作之余是一名独立摄影师。喜欢美食、旅行、看展，偶尔整理下NLP学习笔记，不管技术文还是生活随感，都会分享本人摄影作品，希望文艺的技术青年能够喜欢~~如果有想拍写真的妹子也可以在个人公号**【青影三弄】**留言~

Photograhy Sharing

拍摄参数 | f2.8/ 190mm/ ISO1500

设备 | Canon6D2 / 70-200mm f2.8L III USM

先分享一张在佛罗伦萨的人文摄影~

今年去意呆的时候特别热，每天都是白晃晃的大太阳，所以我总喜欢躲到附近的教堂，那里是“免费的避暑胜地”。

“诸圣教堂”离我住处很近，当时遇到神父祷告，他还缓缓唱了首歌，第一次觉得美声如此动人，怪不得意呆人那么热爱歌剧，连我这种音乐小白都被感染到了~

喜欢“诸圣教堂”的另一个原因是这里埋葬着基尔兰达约、波提切利和他爱慕的女神西蒙内塔。生前“小桶”因她创作了《维纳斯的诞生》，离开人世他又得偿所愿和心爱之人共眠。情深如此，我能想到的中国式浪漫大概也只有“庭有枇杷树，吾妻死之年所手植也，今已亭亭如盖矣”相比了..

AI sharing

之前介绍过Seq2Seq+SoftAttention这种序列模型实现机器翻译，那么抛弃RNN，全面拥抱attention的transformer又是如何实现的呢。

本篇介绍Transformer的原理及Pytorch实现，包括一些细枝末节的trick和个人感悟，这些都是在调试代码过程中深切领会的。网上查了很多文章，大部分基于哈佛那片论文注释，数据集来源于tochtext自带的英-德翻译，但是本篇为了和上面摄影分享对应以及灵活的自定义数据集，采用意大利-英语翻译。

CONTENT

【资料索取】

公众号回复：Transformer

可获取完整代码

1. Transfomer简介

《Attention Is All Your Need》 是一篇Google提出全面使用self-Attention的论文。这篇论文中提出一个全新的模型，叫 Transformer，抛弃了以往深度学习任务里面使用到的 CNN 和 RNN。目前大热的Bert就是基于Transformer构建的，这个模型广泛应用于NLP领域，例如机器翻译，问答系统，文本摘要和语音识别等等方向。

众所周知RNN虽然模型设计小巧精妙，但是其线性序列模型决定了无法实现并行，从两个任意输入和输出位置获取依赖关系都需要大量的运算，运算量严重受到距离的制约；而且距离不但影响性能也影响效果，随着记忆时序的拉长，记忆削弱，导致学习能力削弱。

为了抛弃RNN step by step线性时序，Transformer使用了可以biself-attention，不依靠顺序和距离就能获得两个位置（实质是key和value）的依赖关系（hidden）。这种计算成本减少到一个固定的运算量，虽然注意力加权会减少有效的resolution表征力，但是使用多头multi-head attention可以弥补平均注意力加权带来的损失。

自注意力是一种关注自身序列不同位置的注意力机制，能计算序列的表征representation。

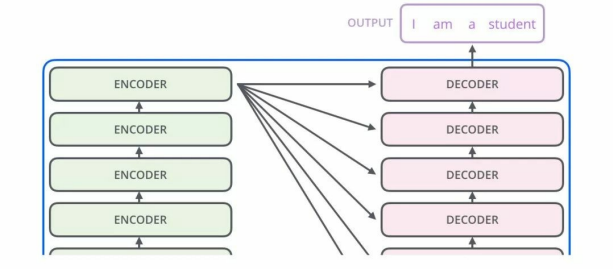
和之前分享的Seq2Seq+SoftAttention相比，Transformer不仅关注encoder和decoder之间的attention，也关注encoder和decoder各自内部自己的attention。也就是说前者的hidden是靠lstm来实现，而transformer的encoder或者decoder的hidden是靠self-attention来实现。

2. 模型概览

Transformer结构和Seq2Seq模型是一样的，也采用了Encoder-Decoder结构，但Transformer更复杂。

2.1宏观组成

Encoder由6个EncoderLayer构成，Decoder由6个DecoderLayer构成：



对应的代码逻辑如下，make_model包含EncoderDecoder模块，可以看到N=6表示Encoder和Decoder的子层数量，d_ff是前馈神经网络的中间隐层维度，h=代表的是注意力层的头数。

后面还规定了初始化的策略，如果每层参数维度大于1，那么初始化服从均匀分布init.xavier_uniform

```
def make_model(src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab))
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform(p)
    return model
```

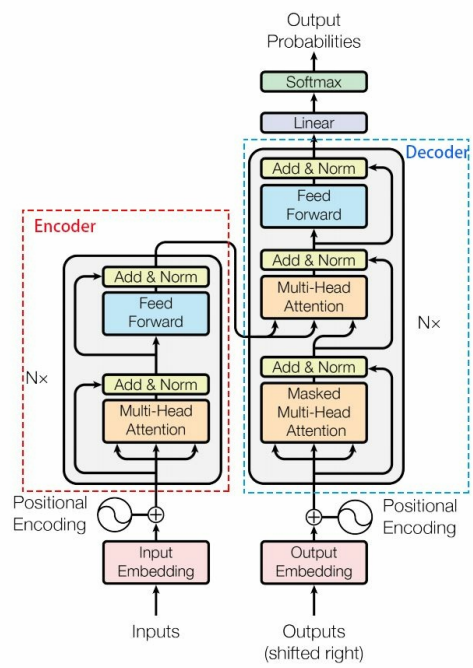
EncoderDecoder里面除了Encoder和Decoder两个模块，还包含embed和generator。Embed层是对对输入进行初始化，词嵌入包含普通的Embeddings和位置标记PositionEncoding；Generator作用是对输出进行full linear+softmax

其中可以看到Decoder输入的memory就是来自前面Encoder的输出，memory会分别喂入Decoder的6个子层。

```
class EncoderDecoder(nn.Module):
def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator
    def forward(self, src, tgt, src_mask, tgt_mask):
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)
    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)
    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
class Generator(nn.Module):
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)
    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

2.2内部结构

上面是Transformer宏观上的结构，那Encoder和Decoder内部都有哪些不同于Seq2Seq的技术细节呢：



(1)Encoder的输入序列经过**word embedding**和**positional encoding**后，输入到encoder。

(2)在EncoderLayer里面先经过8个头的self-attention模块处理source序列自身。这个模块目的是求得序列的hidden，利用的就是自注意力机制，而非之前RNN需要step by step算出每个hidden。然后经过一些 norm和drop基本处理，再使用残差连接模块，目的是为了避免梯度消失问题。（后面代码实现和上图在实现顺序上有一点出入）

(3)在EncoderLayer里面再进入Feed-Forward前馈神经网络，实际上就是做了两次dense，linear2(activation(linear1))。然后同上经过一些norm和drop基本处理，再使用残差连接模块。

(4)Decoder的输入序列处理方式同上

(5)在DecoderLayer里面也要经过8个头的self-attentention模块处理target序列自身。不同于Encoder层，这里只需要关注输入时刻t之前的部分，目的是为了符合decoder看不到未来信息的逻辑，所以这里的mask是融合了pad-mask和sequence-mask两种。同Encoder，这个模块目的也是为了求得target序列自身的hidden，然后经过一些norm和drop基本处理，再使用残差连接模块。

(6)在DecoderLayer里面再进入src-attention模块，这个模块也是相比Encoder增加的注意力层。其实注意力结构都是相似的，只是(query,key,value)不同，对于self-attention这三个值都是一致的，对于src-attention，query来自decoder的hidden，key和value来自encoder的hidden

(7)在DecoderLayer里面最后进入Feed-Forward前馈神经网络，同上。

介绍完Transformer整体结构，下面从数据集处理到各层代码实现细节进行详细说明~

3. 数据加载及预处理

GPU环境使用Google Colab 单核16g，数据集eng-ita.txt，普通的英意翻译对的文本数据。数据集预处理使用的是torchtext+spacy工具，他使用的整体思路是构造Dataset，字典、Iterator实现批量化、对矩阵进行mask pad。

数据预处理非常重要，这里涉及很多提高训练性能的trick。下面具体看一下如何使用自定义数据集来完成这些预处理步骤。

3.1原始数据构造DataFrame

先加载文本，并将source和target两列转换为两个独立的list：

```
corpus = open('./dataset/%s-%s.txt' % ('eng', 'ita') , 'r', encoding='utf-8').readlines()
random.shuffle(corpus)
def prepare_data(lang1_name, lang2_name, reverse=False):
    print("Reading lines...")
    input_lang, output_lang = [], [] # raw
    for parallel in corpus:
      so,ta = parallel[:-1].split('\t') #一行实际是英文和法文对儿 用tab来分割
      if so.strip() == "" or ta.strip() == "":
        continue
      input_lang.append(so)
      output_lang.append(ta)
    if reverse:
      return output_lang,input_lang
    else:
      return input_lang,output_lang
input_lang, output_lang = prepare_data('eng', 'ita', True)
```

因为torchtext的dataset的输入需要DataFrame格式，所以这里先利用上面的source和target list构造DataFrame。训练集、验证集、测试集按照实际要求进行划分：

```
train_list=corpus[:-3756]
valid_list=corpus[-1622:]
test_list=corpus[-3756:-1622]
c_train={'src':input_lang[:-3756],'trg':output_lang[:-3756]}
train_df=pd.DataFrame(c_train)
c_valid={'src':input_lang[-1622:],'trg':output_lang[-1622:]}
```

```
valid_df=pd.DataFrame(c_valid)
c_test={'src':input_lang[-3756:-1622],'trg':output_lang[-3756:-1622]}
test_df=pd.DataFrame(c_test)
```

3.2构造Dataset

这里主要包含分词、指定起止符和补全字符以及限制序列最大长度。

因为torchtext的Dataset是由example组成，example的含义就是一条翻译对记录。

```
# 分词
spacy_it = spacy.load('it')
spacy_en = spacy.load('en')
def tokenize_it(text):
    return [tok.text for tok in spacy_it.tokenizer(text)]
def tokenize_en(text):
    return [tok.text for tok in spacy_en.tokenizer(text)]
# 定义FIELD配置信息
# 主要包含以下数据预处理的配置信息，比如指定分词方法，是否转成小写，起始字符，结束字符，补全字符以及词典等等
BOS_WORD = '<s>'
EOS_WORD = '</s>'
BLANK_WORD = "<blank>"
SRC = data.Field(tokenize=tokenize_it, pad_token=BLANK_WORD)
TGT = data.Field(tokenize=tokenize_en, init_token=BOS_WORD, eos_token=EOS_WORD, pad_token=BLANK_WORD)
# get_dataset构造并返回Dataset所需的examples和fields
def get_dataset(csv_data, text_field, label_field, test=False):
    fields = [('id', None), ('src', text_field), ('trg', label_field)]
    examples = []
    if test:
        for text in tqdm(csv_data['src']):  # tqdm的作用是添加进度条
            examples.append(data.Example.fromlist([None, text, None], fields))
    else:
        for text, label in tqdm(zip(csv_data['src'], csv_data['trg'])):
            examples.append(data.Example.fromlist([None, text, label], fields))
    return examples, fields
# 得到构建Dataset所需的examples和fields
train_examples, train_fields = get_dataset(train_df, SRC, TGT)
valid_examples, valid_fields = get_dataset(valid_df, SRC, TGT)
test_examples, test_fields = get_dataset(test_df, SRC, None, True)
构建Dataset数据集
# 构建Dataset数据集
# 这里的ita最大长度也就56
MAX_LEN = 100
train = data.Dataset(train_examples,train_fields,
        filter_pred=lambda x: len(vars(x)['src'])<= MAX_LEN and len(vars(x)['trg']) <= MAX_LEN)
valid = data.Dataset(valid_examples, valid_fields,
        filter_pred=lambda x: len(vars(x)['src']) <= MAX_LEN and len(vars(x)['trg']) <= MAX_LEN)
test = data.Dataset(test_examples, test_fields,
        filter_pred=lambda x: len(vars(x)['src']) <= MAX_LEN)
```

3.3构造字典

```
MIN_FREQ = 2 #统计字典时要考虑词频
SRC.build_vocab(train.src, min_freq=MIN_FREQ)
TGT.build_vocab(train.trg, min_freq=MIN_FREQ)
```

3.4构造Iterator

torchtext的Iterator主要负责把Dataset进行批量划分、字符转数字、矩阵pad。

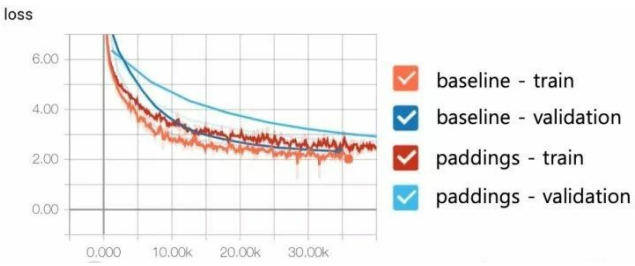
这里有个很重要的点就是批量化，本例使用的是动态批量化，即每个batch的批大小是不同的，是以每个batch的token数量作为统一划分标准，也就是说每个batch的token数基本一致，这个机制是通过batch_size_fn来实现的，比如batch1[16,20]，batch2是[8,40]，两者的批大小是不同的分别是16和8，但是token总数是一样的都是320。

采取这种方式的特点就是他会把长度相同序列的聚集到一起，然后进行pad，从而减少了pad的比例，为什么要减少pad呢：

(1)padding是对计算资源的浪费，pad越多训练耗费的时间越长。

(2)padding的计算会引入噪声，nsformer 中，Lay erNorm 会使 padding 位置的值变为非0，这会使每个 padding 都会有梯度，引起不必要的权重更新。

下图是随意组织pad的batch(paddings)和长度相近原则组织pad的batch(baseline)



实现代码部分，可以看到这里MyIterator实现了data.Iterator的create_batch()函数，其实真正起作用的是里面的torchtext.data.batch()函数部分，他的功能是把原始的字符数据按照batch_size_fn算法来进行batch并且shuffle，没有做数字化也没有做pad。

```
class MyIterator(data.Iterator):
    def create_batches(self):
        if self.train:
            def pool(d, random_shuffler):
                for p in data.batch(d, self.batch_size * 100):
                    p_batch = data.batch(
                        sorted(p, key=self.sort_key),
                        self.batch_size, self.batch_size_fn)
                    for b in random_shuffler(list(p_batch)):
                        yield b
            self.batches = pool(self.data(), self.random_shuffler)
        else:
            self.batches = []
            for b in data.batch(self.data(), self.batch_size,self.batch_size_fn):
                self.batches.append(sorted(b, key=self.sort_key))
global max_src_in_batch, max_tgt_in_batch
def batch_size_fn(new, count, sofar):
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch, len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch, len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

那么什么时候对batch进行数字化和pad呢，看了下源码，实际这些操作封装在data.Iterator.__iter__的torchtext.data.Batch这个类中。

这里还有一个问题，BATCH_SIZE这个参数设置多少合适呢，这个参数在这里的含义代表每个batch的token总量，我测试了下单核16G的

colabGPU训练环境需要6000，如果超过这个数值，内存容易爆。

```
BATCH_SIZE = 6000
train_iter = MyIterator(train, batch_size=BATCH_SIZE, device=0, repeat=False,
                        sort_key=lambda x: (len(x.src), len(x.trg)),
                        batch_size_fn=batch_size_fn, train=True)
valid_iter = MyIterator(valid, batch_size=BATCH_SIZE, device=0, repeat=False,
                        sort_key=lambda x: (len(x.src), len(x.trg)),
                        batch_size_fn=batch_size_fn, train=False)
```

3.5生成mask

我们知道整个模型的输入就是src,src_mask,tgt,tgt_mask，现在src和tgt已经比较明确了，那mask部分呢，总的来说mask就是对上面的pad部分做一个统计行程对应的mask矩阵。

但是前面在讲整体结构的时候提到，Decoder的mask比Encoder的mask多一层含义，就是sequence_mask，下面说下这两类mask。

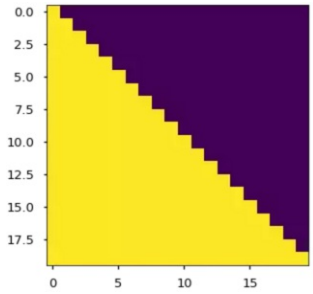
(1)padding mask

Seq2Seq+SoftAttention里面计算的mask就是padding mask。每个批次输入序列长度是不一样的，要对输入序列进行对齐。具体来说，就是给在较短的序列后面填充0。因为这些填充的位置，其实是没什么意义的，所以我们的attention机制不应该把注意力放在这些位置上，所以我们需要进行一些处理。具体的做法是，把这些位置的值机上一个非常大的复数，经过softmax这些位置就会接近0。而我们的padding mask实际上是一个张量，每个值都是一个Bool，值为False的地方就是我们要进行处理的地方。

(2)sequence mask

自然语言生成(例如机器翻译，文本摘要)是auto-regressive的，在推理的时候只能依据之前的token生成当前时刻的token，正因为生成当前时刻的token的时候并不知道后续的token长什么样，所以为了保持训练和推理的一致性，训练的时候也不能利用后续的token来生成当前时刻的token。这种方式也符合人类在自然语言生成中的思维方式。

那么具体怎么做呢，也很简单，产生一个上三角矩阵，上三角的值全为1，下三角的值全为0，对角线也是0。把这个矩阵作用在每一个序列上，就达到我们的目的。如图：



总结一下transformer里面的mask使用情况：

* encoder里的self-attention使用的是padding mask

* decoder里面的self-attention使用的是padding mask+sequence mask；context-attention使用的是padding mask

下面看代码来实现上面两种mask，其中make_std_mask里面实现了pad mask+sequence mask;

除了mask，代码还对target部分做了处理，self.trg表示输入，self.trg表示最终loss里面标签角色，比self.trg往后挪一列。

```
class BatchMask:
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()
    @staticmethod
    def make_std_mask(tgt, pad):
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & Variable(subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
        return tgt_mask
def subsequent_mask(size):
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
def batch_mask(pad_idx, batch):
    src, trg = batch.src.transpose(0, 1), batch.trg.transpose(0, 1)
    return BatchMask(src, trg, pad_idx)
pad_idx = TGT.vocab.stoi["<blank>"]
```

4. Embedding层

这一部分针对输入模型的数据的词嵌入处理，主要包含三个过程：普通词嵌入word Embeddings、位置编码PositionalEncoding、层归一化LayerNorm。

(word Embedding是对词汇本身编码；Positional encoding是对词汇的位置编码)

4.1普通Embedding

初始化embedding matrix，通过embedding lookup将Inputs映射成token embedding，大小是[batch size, max seq length, embedding size]，然后乘以embedding size的开方。那么这里为什么要乘以$\sqrt{d_{model}}$？论文并没有讲为什么这么做，我看了代码，猜测是因为

embedding matrix的初始化方式是xavier init，这种方式的方差是1/embedding size，因此乘以embedding size的开方使得embedding matrix的方差是1，在这个scale下可能更有利于embedding matrix的收敛。

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model
    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

4.2位置编码PositionalEncoding

我们知道RNN使用了step by step这种时序算法保持了序列本有的顺序特征，但缺点是无法串行降低了性能， transformer的主要思想self-attention在本质上抛弃了rnn这种时序特征，也就抛弃了所谓的序列顺序特征，如果缺失了序列顺序这个重要信息，那么结果就是所有词语都对了，但是就是无法组成有意义的语句。那么他是怎么弥补的呢?

为了处理这个问题，transformer给encoder层和decoder层的输入添加了一个额外的向量Positional Encoding，就是位置编码 ，维度和embedding的维度一样，这个向量采用了一种很独特的方法来让模型学习到这个值，这个向量能决定当前词的位置，或者说在一个句子中不同的词之间的距离。

这个位置向量的具体计算方法有很多种，论文中的计算方法如下sinusoidal version，这里 的2i就是指的d_model这个维度上的，和pos不是一回事，pos就是可以自己定义1-5000的序列,每个数字代表一个序列位置：

$$PE(pos,2i) = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

$$PE(pos,2i+1) = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

上式右端表达式中pos下面的除数如果使用exp来表示的话，手写推导如下：

$$10000^{\frac{-2i}{d}} = x$$

$$\log(10000^{\frac{-2i}{d}}) = \log x$$

$$2i * -\frac{(\log 10000)}{d} = \log x$$

$$\exp(2i * -\frac{(\log 10000)}{d}) = x$$

代码表示：

```
position = torch.arange(0, max_len).unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2) * -(math.log(10000.0)/d_model))
pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)
```

其中pos是指当前词在句子中的位置，i是指向量d_model中每个值的index，可以看出，在d_model偶数位置index，使用正弦编码，在奇数位置，使用余弦编码。上面公式的dmodel就是模型的维度，论文默认是512。

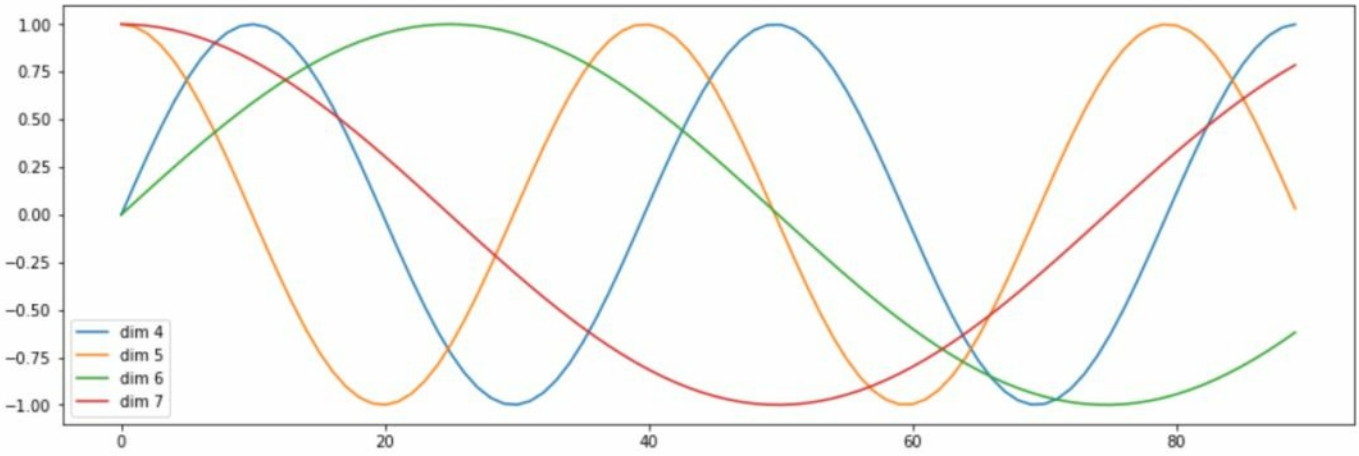
这个编码的公式的意思就是：给定词语的位置pos,我们可以把它编码成dmodel维的向量，也就是说位置编码的每个维度对应正弦曲线，波长构成了从2π到10000*2π的等比序列。上面的位置编码是绝对位置编码，但是词语的相对位置也非常重要，这就是论文为什么使用三角函数的原因。

正弦函数能够表达相对位置信息。主要数学依据是以下两个公式，对于词汇之间的位置偏移k，PE(pos+k)可以表示成PE(pos)和PE(k)的组合形式，这就是表达相对位置的能力：

$$sin(\alpha+\beta) = sin\alpha cos\beta + cos\alpha sin\beta$$

$$cos(\alpha+\beta) = cos\alpha cos\beta - sin\alpha sin\beta$$

可视化位置编码效果如下，横轴是位置序列seq_len这个维度，竖轴是d_model这个维度：



代码中加入了dropout，具体实现如下。self.register_buffer可以将tensor注册成buffer，网络存储时也会将buffer存下，当网络load模型是，会将存储模型的buffer也进行赋值，buffer在forward中更新而不再梯度下降中更新，optim.step只能更新nn.Parameter类型参数。

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, 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).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * -(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)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)], requires_grad=False)
        return self.dropout(x)
```

4.3层归一化

层归一化layernorm是一种基础数据里手段，作用于输入和输出，那么本例都什么时候用到呢，一个是source和target数据经过词嵌入后进入子层前要进行层归一化；另一个就是encoder和decoder模块输出的时候要进行层归一化。

layernorm不同于batchnorm，他是在d_model这个维度上计算平均值和方差，公式如下：

$$LN(x_i) = \alpha * \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \varepsilon}} + \beta$$

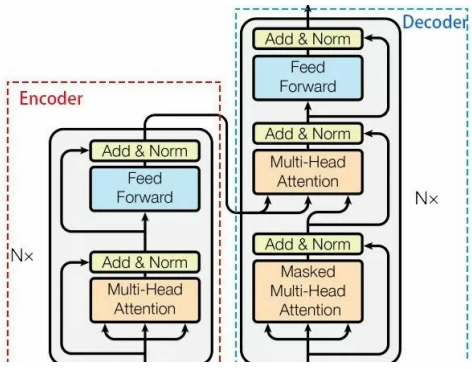
可以看到这里引入了参数α和β，所以可以使用torch里面的nn. Parameter，他的作用就是初始化一个可进行训练优化的参数，并将这个参数绑定到module里面。

代码如下：

```
class LayerNorm(nn.Module):
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps
    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

5. SubLayer子层组成

这里的sublayer存在于每个encoderlayer和decoderlayer中，是公用的部分，encoderlayer里面有两个子层（mulhead-self attention/feed-forward）靠残差层连接；decoderlayer里面有三个子层（mulhead-self attention/mulhad-context attention/feed-forward）。



这里的mulhead-self attention和mulhad-context attention是整个transformer最核心的部分。前者是自注意力机制，出现在encoder或decoder内部序列自身学习hidden；后者上下文注意力机制，相当于之前分享文章里的Seq2Seq+softattention，是encoder和decoder之间的注意力为了学习context。这两种注意力机制结构实际上是相同的，不同的在于输入部分(query,key,value)：self-attention的三个数值都是一致的；而contex-attention的query来自decoder，key和value来自encoder。

5.1多头MuHead Attention(self+context attention)

由于之前分享的文章详细讲解过context-attention(相当于soft-attention)，所以这里详细说明self-attention和multi-head两种机制。

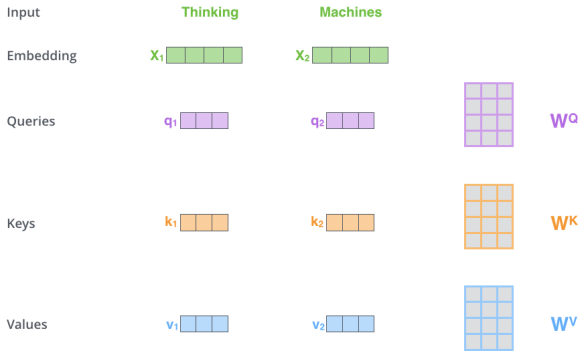
(1)self-attention

我们看个例子：

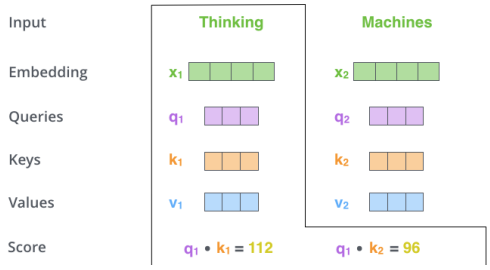
The animal didn't cross the street because it was too tired

这里的 it 到底代表的是 animal 还是 street 呢，对于人来说能很简单的判断出来，但是对于机器来说，是很难判断的，self-attention就能够让机器把 it 和 animal 联系起来，接下来我们看下详细的处理过程。

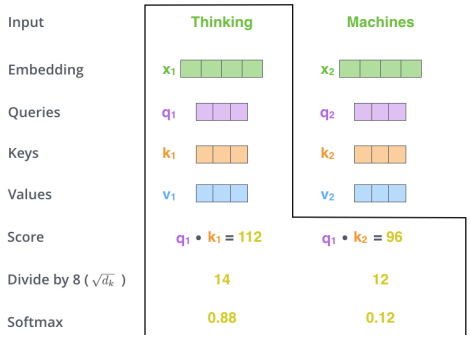
首先，self-attention会计算出三个新的向量，在论文中，向量的维度是512维，我们把这三个向量分别称为Query、Key、Value，这三个向量是用embedding向量与一个参数矩阵W相乘得到的结果，这个矩阵是随机初始化的。



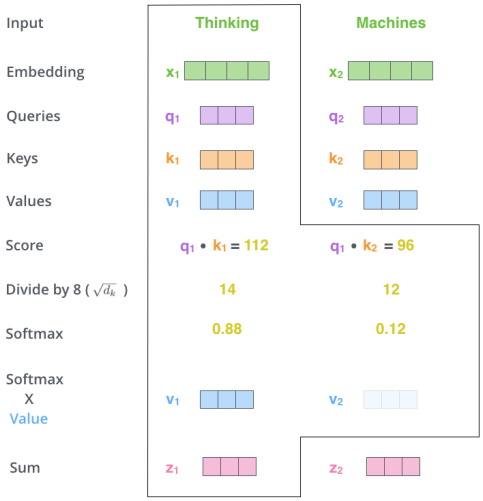
计算self-attention的分数值，该分数值决定了当我们在某个位置encode一个词时，对输入句子的其他部分的关注程度。这个分数值的计算方法是Query与Key做点成，以下图为例，首先我们需要针对Thinking这个词，计算出其他词对于该词的一个分数值，首先是针对于自己本身即q1·k1，然后是针对于第二个词即q1·k2。



接下来，把点成的结果除以一个常数，这里我们除以8，这个值一般是采用上文提到的矩阵的第一个维度的开方即64的开方8，当然也可以选择其他的值，然后把得到的结果做一个softmax的计算。得到的结果即是每个词对于当前位置的词的相关性大小，当然，当前位置的词相关性肯定会会很大。

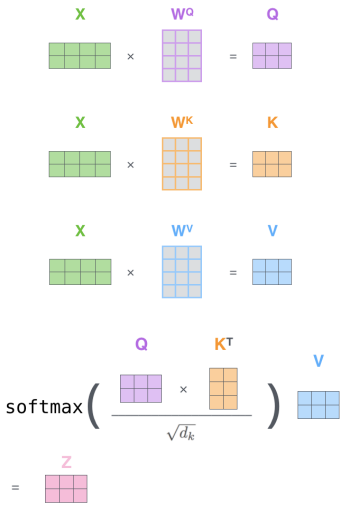


下一步就是把Value和softmax得到的值进行相乘，并相加，得到的结果即是self-attetion在当前节点的值。

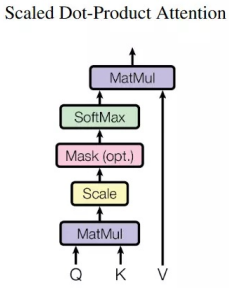


在实际的应用场景，为了提高计算速度，我们采用的是矩阵的方式，直接计算出Query，Key，Value的矩阵，然后把embedding的值与三个矩阵直接相乘，把得到的新矩阵 Q 与 K 相乘，乘以一个常数，做softmax操作，最后乘上 V 矩阵。

这种通过 query 和 key 的相似性程度来确定 value 的权重分布的方法被称为scaled dot-product attention。



结构图如下：



(2)attention score:scaled dot-product

那么这里Transformer为什么要使用scaled dot-product来计算attention score呢?论文的描述含义就是通过确定Q和K之间的相似度来选择V，公式：

$$\text{Attention}(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

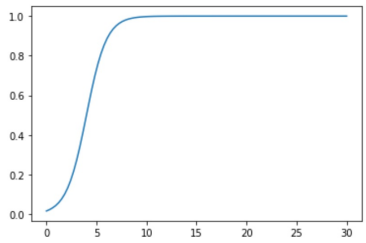
scaled dot-product attention和dot-product attention唯一的区别就是，scaled dot-product attention有一个缩放因子：

$$\frac{1}{\sqrt{d_k}}$$

上面公式中的dk表示的k的维度，在论文里面默认是64。为什么需要加上这个缩放因子呢，论文解释：对于dk很大的时候，点积得到结果量级很大，方差很大，使得处于softmax函数梯度很小的区域。我们知道，梯度很小的情况，对反向传播不利。下面简单的测试反映出不同

量级对，最大值的概率变化。

```
f = lambda x: exp(6*x) / (exp(2*x)+exp(2*x+1)+exp(3*x)+exp(4*x)+exp(5*x+4)+exp(6*x))
x = np.linspace(0, 30, 100)
y_3 = [f(x_i) for x_i in x]
plt.plot(x, y_3)
plt.show()
```



可以看到f是softmax的最大值6x的曲线，当x处于1~50之间不同的量级的时候，所表示的概率，当x量级在>7的时候，差不多其分配的概率就接近1了。也就是说输入量级很大的时候，就会造成梯度消失。

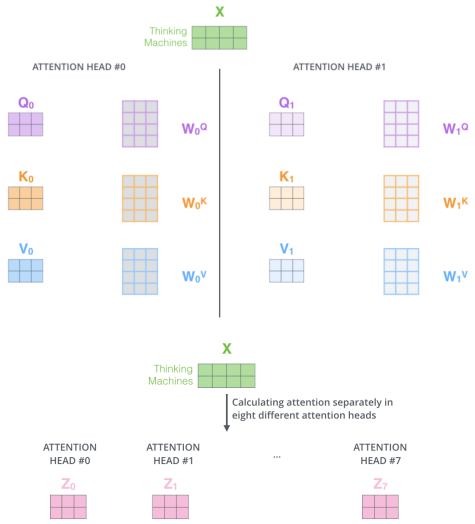
为了克服这个负面影响，除以一个缩放因子可以一定程度上减缓这种情况。点积除以$\sqrt{d_{model}}$，将控 制方差为1，也就有效的控制了梯度消失的问题。

注意部分的代码表示：

```
def attention(query, key, value, mask=None, dropout=None):
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    context = torch.matmul(p_attn, value)
    return context, p_attn
```

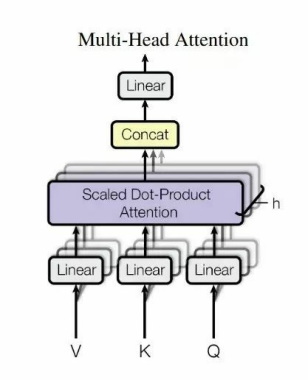
(3)multi-head

这篇论文另一个牛的地方是给attention加入另外一个机制——multi head，该机制理解起来很简单，就是说不仅仅只初始化一组Q、K、V的矩阵，而是初始化多组，tranformer是使用了8组，所以最后得到的结果是8个矩阵。



论文提到，他们发现将Q、K、V通过一个线性映射之后，分成h份，对每一份进行**scaled dot-product attention**效果更好。然后，把各个部分的结果合并起来，再次经过线性映射，得到最终的输出。这就是所谓的**multi-head attention**。上面的超参数h就是**heads**数量。论文默认是8。

下面是multi-head attention的结构图。可以看到QKV在输入前后都有线性变换，总共有四次，上面dk=64=512/8



代码表示：

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)
    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)
        query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
                             for l, x in zip(self.linears, (query, key, value))]
        x, self.attn = attention(query, key, value, mask=mask, dropout=self.dropout)
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

原论文中说到进行Multi-head Attention的原因是将模型分为多个头，形成多个子空间，可以让模型去关注不同方面的信息，最后再将各个方面的信息综合起来。其实直观上也可以想到，如果自己设计这样的一个模型，必然也不会只做一次attention，多次attention综合的结果至少能够起到增强模型的作用，也可以类比CNN中同时使用多个卷积核的作用，直观上讲，多头的注意力有助于网络捕捉到更丰富的特征/信息。

5.2Position-wise Feed-forward前馈传播

这一层很简单，就是一个全连接网络，包含两个线性变换和一个非线性函数Relu；

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

代码：

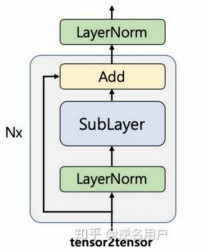
```
class PositionwiseFeedForward(nn.Module):
    # 这里input和output都是d_model，中间层维度是d_ff，本例设置为2048
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        # 两次线性变换，第一次activation是relu，第二次没有
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

这个线性变换在不同的位置(encoder or decoder)都表现地一样，并且在不同的层之间使用不同的参数。论文提到，这个公式还可以用两个核大小为1的一维卷积来解释，卷积的输入输出都是dmodel=512，中间层的维度是dff=2048

那么为什么要在multi-attention后面加一个fnn呢，类比cnn网络中，cnn block和fc交替连接，效果更好。相比于单独的multi-head attention，在后面加一个ffn，可以提高整个block的非线性变换的能力。

5.3残差连接ResidualConnec

残差连接其实很简单，在encoderlayer和decoderlayer里面都一样，本文结构如下：



那么**残差结构**有什么好处呢？显而易见：因为增加了一项x，那么该层网络对x求偏导的时候，多了一个常数项1！所以在反向传播过程中，梯度连乘，也不会造成**梯度消失**！

文章开始的transformer架构图中的Add & Norm中的Add也就是指的这个**shortcut**。

代码如下：

```
class SublayerConnection(nn.Module):
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer):  # 这里的x是指的输入层src 需要对其进行归一化
        norm_x = self.norm(x)
        sub_x = sublayer(norm_x)
        sub_x = self.dropout(sub_x)
```

6. Encoder组合

EncoderLayer由上面两个sublayer(multihead-selfattention和residualconnection)组成；Encoder由6个EncoderLayer组成。

代码如下：

```
class EncoderLayer(nn.Module):
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.residual_conn = clones(SublayerConnection(size, dropout), 2)
        self.size = size
    def forward(self, x, mask):
        x = self.residual_conn[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.residual_conn[1](x, self.feed_forward)
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)
    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

7. Decoder组合

DecoderLayer由上面三个sublayer(multihead-selfattention、multihead-contextattention和residualconnection)组成；Encoder由6个EncoderLayer组成。

代码如下：

```
class DecoderLayer(nn.Module):
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.residual_conn = clones(SublayerConnection(size, dropout), 3)
    def forward(self, x, memory, src_mask, tgt_mask):
        m = memory
        x = self.residual_conn[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.residual_conn[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.residual_conn[2](x, self.feed_forward)
class Decoder(nn.Module):
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)
    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

8. 损失函数和优化器

Transfomer里的损失函数引入标签平滑的概念；梯度下降的优化器引入了动态学习率。下面详细说明。

8.1损失函数实现标签平滑

Transformer使用的标签平滑技术属于discount类型的平滑技术。这种算法简单来说就是把最高点砍掉一点，多出来的概率平均分给所有人。

为什么要实现标签平滑呢，其实就是增加困惑度perplexity，每个时间步都会在一个分布集合里面随机挑词，那么平均情况下挑多少个词才能挑到正确的那个呢。多挑几次那么就意味着困惑度越高使得模型不确定性增加，但是这样子的好处是提高了模型精度和BLEU score。

在实际实现时，这里使用KL div loss实现标签平滑。没有使用one-hot目标分布，而是创建了一个分布，对于整个词汇分布表，这个分布含有正确单词度和剩余部分平滑块的置信度。

代码如下，简单解释下，一般来说损失函数的输入crit(x,target)，标签平滑主要是在处理实际标签target

(1)先使用clone来把target构造成和x一样维度的矩阵

(2)然后使用fill_在上面的新矩阵里面填充平滑因子smoothing

(3)然后使用scatter_把confidence(1-smoothing)填充到上面的矩阵中，按照target index数值，填充到维度对应位置上。比如scatter_(1, (1,2,3),0.6) target新矩阵是(3,10) 那么就在10这个维度上找到index 1、2、3

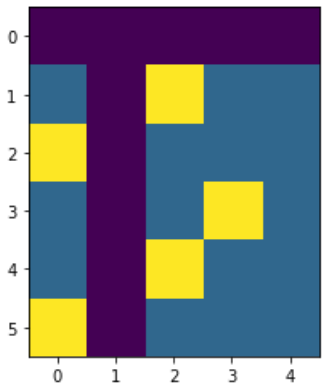
(4)按照一定规则对target矩阵进行pad mask

(5)最后使用损失函数KLDivLoss相对熵，他是求两个概率分布之间的差异，size_averge=False损失值是sum类型，也就是说求得所有token的loss总量。

```
class LabelSmoothing(nn.Module):
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        self.criterion = nn.KLDivLoss(size_average=False)
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None
    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        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))
```

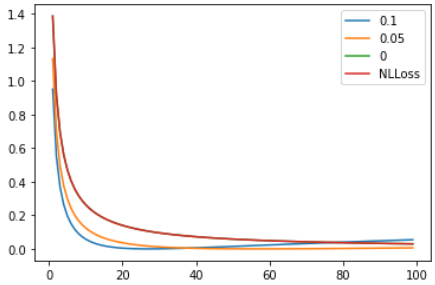
举个简单的例子，可视化感受下标签平滑。深蓝色的T形表示target被pad的部分，黄色部分是可信confidence部分，普蓝色（颜色介于黄和深蓝）代表模糊区间。

```
crit = LabelSmoothing(5, 1, 0.5)
predict =torch.FloatTensor([[0.25, 0, 0.25, 0.25, 0.25],
                            [0.4, 0, 0.2, 0.2, 0.2],
                            [0.625, 0, 0.125, 0.125, 0.125],
                            [0.25, 0, 0.25, 0.25, 0.25],
                            [0.4, 0, 0.2, 0.2, 0.2],
                            [0.625, 0, 0.125, 0.125, 0.125]])
v = crit(Variable(predict.log()),Variable(torch.LongTensor([1,2,0,3,2,0])))
```



那么平滑率到底对loss下降曲线有什么影响呢，举个简单的例子看一下。可以看到当smooth越大，也就是说confidence越小，也就是标签越模糊，loss下降效果反而更好。

```
crits = [LabelSmoothing(5, 0, 0.1),
    LabelSmoothing(5, 0, 0.05),
    LabelSmoothing(5, 0, 0),
    nn.NLLLoss()
    ]
def loss(x,crit):
    d = x + 3 * 1
    predict = torch.FloatTensor([[0, x/d, 1/d, 1/d, 1/d],])
    return crit(Variable(predict.log()),Variable(torch.LongTensor([1]))).item()
plt.plot(np.arange(1, 100), [[loss(x,crit) for crit in crits] for x in range(1, 100)])
plt.legend(["0.1","0.05","0","NLLoss"])
```



8.2优化器实现动态学习率

我们知道学习率是梯度下降的重要因素，随着梯度的下降，使用动态变化的学习率，往往取的较好的效果。

这里的算法实现的是先warmup增大学习率，达到摸个合适的step再减小学习率。公式如下。：

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

可以看出来在开始的warmup steps(本例是8000)的时候学习率随着step线性增加，然后学习率随着步数的导数平方根step_num^(-0.5)成比例的减小，8000就是那个转折点。

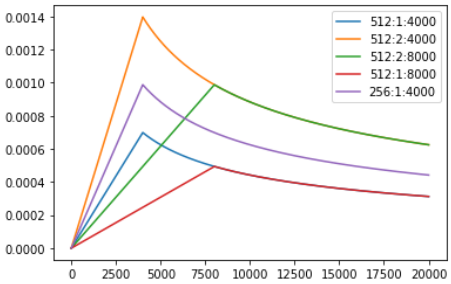
代码如下。除去step，learningrate还和d_model，factor，warmup有关。这个优化器封装了对lr的修改算法

```
class NoamOpt:
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0
    def step(self):
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
```

```
        self.optimizer.step()
    def rate(self, step=None):
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
             min(step ** (-0.5), step * self.warmup ** (-1.5)))
```

下面把影响学习率变化的三个超参数model_size/factor/warmup进行可视化：

```
opts = [NoamOpt(512, 1, 4000, None),
        NoamOpt(512, 2, 4000, None),
        NoamOpt(512, 2, 8000, None),
        NoamOpt(512, 1, 8000, None),
        NoamOpt(256, 1, 4000, None)]
plt.plot(np.arange(1, 20000), [[opt.rate(i) for opt in opts] for i in range(1, 20000)])
plt.legend(["512:1:4000","512:2:4000","512:2:8000","512:1:8000", "256:1:4000"])
```



可以看到，随着step的增加，可以看到学习率随着三个超参数的变化曲线：

（1）d_model越小，学习率峰值越大；

（2）factor越大，学习率峰值越大

（3）warmupsteps越大，学习率的峰值越往后推迟，且学习率峰值相对降低一些

【本文采取的超参数是512,2,8000】

8.3整合

SimpleLossCompute这个类包含了softmax+loss+optimizer三个功能。

(1)这里包含了三个功能先用generator计算linear+softmax

(2)利用criterion来计算loss 这里的loss function是KLDivLoss 求得loss是个sum的形式 所以要根据ntokens数量来求平均loss

(3)优化器是opt 也就是梯度下降算法 这个优化器里面有对lr的算法

```
class SimpleLossCompute:
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt
    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)), y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.item() * norm
```

9. 模型训练Train

因为我只是在colab上训练，所以就是单核16GPU，20个epoch，约10000次迭代，花费了3个多小时。

训练模型中包含了时间计数、loss记录、数据和model的cuda()、step计数、学习率记录。

代码如下：

```
USE_CUDA = torch.cuda.is_available()
print_every = 50
plot_every = 100
plot_losses = []
def time_since(t):
    now = time.time()
    s = now - t
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start_epoch = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    plot_loss_total = 0
    plot_tokens_total = 0
    for i, batch in enumerate(data_iter):
        src = batch.src.cuda() if USE_CUDA else batch.src
        trg = batch.trg.cuda() if USE_CUDA else batch.trg
        src_mask = batch.src_mask.cuda() if USE_CUDA else batch.src_mask
        trg_mask = batch.trg_mask.cuda() if USE_CUDA else batch.trg_mask
        model = model.cuda() if USE_CUDA else model
        out = model.forward(src, trg, src_mask, trg_mask)
        trg_y = batch.trg_y.cuda() if USE_CUDA else batch.trg_y
        ntokens = batch.ntokens.cuda() if USE_CUDA else batch.ntokens
        loss = loss_compute(out, trg_y, ntokens)
        total_loss += loss
        plot_loss_total += loss
        total_tokens += ntokens
        plot_tokens_total += ntokens
        tokens += ntokens
        if i % print_every == 1:
            elapsed = time.time() - start_epoch
            print("Epoch Step: %3d  Loss: %10f  time:%8s  Tokens per Sec: %6.0f  Step: %6d  Lr: %0.8f" %
                  (i, loss / ntokens, time_since(start), tokens / elapsed,
                   loss_compute.opt._step if loss_compute.opt is not None else 0,
```

```
                loss_compute.opt._rate if loss_compute.opt is not None else 0))
            tokens = 0
            start_epoch = time.time()
        if i % plot_every == 1:
            plot_loss_avg = plot_loss_total / plot_tokens_total
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0
            plot_tokens_total = 0
    return total_loss / total_tokens
model = make_model(len(SRC.vocab), len(TGT.vocab), N=6)
criterion = LabelSmoothing(size=len(TGT.vocab), padding_idx=pad_idx, smoothing=0.1)
model_opt = NoamOpt(model.src_embed[0].d_model, 2, 8000,
                    torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

训练结果：

```
start = time.time()
for epoch in range(20):
  print('EPOCH',epoch,'----------------------------------------------------------------')
  model.train()
  run_epoch((batch_mask(pad_idx, b) for b in train_iter),
        model,
        SimpleLossCompute(model.generator, criterion, opt=model_opt))
  model.eval()
  loss=run_epoch((batch_mask(pad_idx, b) for b in valid_iter),
        model,
        SimpleLossCompute(model.generator, criterion, opt=None))
  print(loss)
```

```
EPOCH 0 ------------------------------------------------------------
Epoch Step:   1   Loss:   8.148599    time:    0m 4s    Tokens per Sec:   2350    Step:      2    Lr: 0.00000025
Epoch Step:  51   Loss:   7.085064    time:    1m 1s    Tokens per Sec:   4500    Step:     52    Lr: 0.00000642
Epoch Step: 101   Loss:   6.790075    time:    2m 0s    Tokens per Sec:   4545    Step:    102    Lr: 0.00001260
Epoch Step: 151   Loss:   6.893210    time:   2m 58s    Tokens per Sec:   4377    Step:    152    Lr: 0.00001878
Epoch Step: 201   Loss:   5.479636    time:   3m 56s    Tokens per Sec:   4465    Step:    202    Lr: 0.00002495
Epoch Step: 251   Loss:   4.828885    time:   4m 55s    Tokens per Sec:   4370    Step:    252    Lr: 0.00003113
Epoch Step: 301   Loss:   4.173584    time:   5m 53s    Tokens per Sec:   4352    Step:    302    Lr: 0.00003730
Epoch Step: 351   Loss:   4.263603    time:   6m 52s    Tokens per Sec:   4463    Step:    352    Lr: 0.00004348
Epoch Step: 401   Loss:   4.160561    time:   7m 51s    Tokens per Sec:   4460    Step:    402    Lr: 0.00004966
Epoch Step: 451   Loss:   3.140965    time:   8m 50s    Tokens per Sec:   4464    Step:    452    Lr: 0.00005583
Epoch Step:   1   Loss:   3.484842    time:   9m 30s    Tokens per Sec:   3630    Step:      0    Lr: 0.00000000
tensor(3.6286, device='cuda:0')
```

.....step达到8000后的训练情况

```
EPOCH 16 -----------------------------------------------------------
Epoch Step:   1   Loss:   0.938953    time:151m 51s    Tokens per Sec:   1746    Step:   7746    Lr: 0.00095684
Epoch Step:  51   Loss:   0.227301    time:152m 50s    Tokens per Sec:   4459    Step:   7796    Lr: 0.00096301
Epoch Step: 101   Loss:   0.216163    time:153m 48s    Tokens per Sec:   4446    Step:   7846    Lr: 0.00096919
Epoch Step: 151   Loss:   0.179892    time:154m 45s    Tokens per Sec:   4507    Step:   7896    Lr: 0.00097537
Epoch Step: 201   Loss:   0.201305    time:155m 43s    Tokens per Sec:   4524    Step:   7946    Lr: 0.00098154
Epoch Step: 251   Loss:   0.273947    time:156m 39s    Tokens per Sec:   4588    Step:   7996    Lr: 0.00098772
Epoch Step: 301   Loss:   0.216780    time:157m 38s    Tokens per Sec:   4517    Step:   8046    Lr: 0.00098538
Epoch Step: 351   Loss:   0.216690    time:158m 36s    Tokens per Sec:   4548    Step:   8096    Lr: 0.00098234
Epoch Step: 401   Loss:   0.211250    time:159m 33s    Tokens per Sec:   4472    Step:   8146    Lr: 0.00097932
Epoch Step: 451   Loss:   0.292448    time:160m 31s    Tokens per Sec:   4511    Step:   8196    Lr: 0.00097632
Epoch Step:   1   Loss:   0.181924    time:161m 10s    Tokens per Sec:   3684    Step:      0    Lr: 0.00000000
tensor(0.2073, device='cuda:0')
```

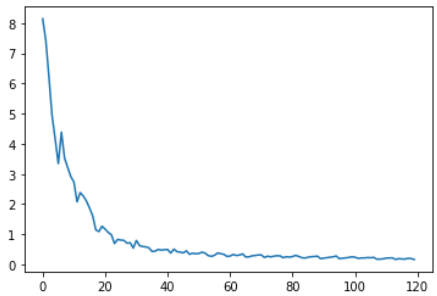
.....最后epoch

```
EPOCH 19 -----------------------------------------------------------
Epoch Step:   1   Loss:   0.131061    time: 180m 4s    Tokens per Sec:   2603    Step:   9198    Lr: 0.00092161
Epoch Step:  51   Loss:   0.317867    time: 181m 1s    Tokens per Sec:   4481    Step:   9248    Lr: 0.00091912
Epoch Step: 101   Loss:   0.150125    time:181m 59s    Tokens per Sec:   4493    Step:   9298    Lr: 0.00091664
Epoch Step: 151   Loss:   0.173796    time:182m 56s    Tokens per Sec:   4617    Step:   9348    Lr: 0.00091419
Epoch Step: 201   Loss:   0.176395    time:183m 54s    Tokens per Sec:   4509    Step:   9398    Lr: 0.00091175
Epoch Step: 251   Loss:   0.167047    time:184m 53s    Tokens per Sec:   4395    Step:   9448    Lr: 0.00090934
Epoch Step: 301   Loss:   0.162348    time:185m 51s    Tokens per Sec:   4397    Step:   9498    Lr: 0.00090694
Epoch Step: 351   Loss:   0.177182    time:186m 49s    Tokens per Sec:   4469    Step:   9548    Lr: 0.00090456
Epoch Step: 401   Loss:   0.264141    time:187m 48s    Tokens per Sec:   4444    Step:   9598    Lr: 0.00090220
Epoch Step: 451   Loss:   0.184936    time:188m 45s    Tokens per Sec:   4571    Step:   9648    Lr: 0.00089986
Epoch Step:   1   Loss:   0.138760    time:189m 24s    Tokens per Sec:   3643    Step:      0    Lr: 0.00000000
tensor(0.1592, device='cuda:0')
```

保存模型：

```
state = {'model': model.state_dict(), 'optimizer': model_opt, 'epoch': epoch, 'loss': loss,
         'plot_losses': plot_losses}
torch.save(state, 'mt_transformer_it&en%02d.pth.tar' % (epoch))
```

loss曲线:



10. 模型测试生成

测试生成部分利用的model.decode()函数，采样方式使用的贪婪算法，部分代码:

```
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len - 1):
        out = model.decode(memory, src_mask, Variable(ys), Variable(subsequent_mask(ys.size(1)).type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
```

```
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys
```

输出结果：

```
src: tensor([[  28,  306,    9,    3,  876, 2159,    4]])
trg: tensor([[  2,  25,   8,  47,   6,  63, 175,  24,   9,   3,   1,   1]])
src: tensor([[  28,  306,    9,    3,  876, 2159,    4]]) torch.Size([1, 7])
ys: tensor([[2]]) torch.Size([1, 1])
out: torch.Size([1, 60])
out: tensor([[  2,  25,   8,  47,   6,  63, 175,  24,   9,   3,  24,   9,   3,  24,
           9,   3,  24,   9,   3,  24,   9,   3,  24,   9,   3,  24,   9,   3,
          24,   9,   3,  24,   9,   3,  24,   9,   3,  24,   9,   3,  24,   9,
           3,  24,   9,   3,  24,   9,   3,  24,   9,   3,  24,   9,   3,  24,
           9,   3,  24,   9]])
Source      :   pensa che Tom possa trovarlo ?
Target      :   Do you think Tom can find it ?
Translation :   Do you think Tom can find it ?
```

12.注意力分布可视化

先随机对valid验证集中某个句子进行翻译

```
for i, batch in enumerate(valid_iter):
    if i == 2:
        ........
        out = greedy_decode(model, src, src_mask, max_len=70, start_symbol=TGT.vocab.stoi["<s>"])
        source = ""
        print("Source      :", end="\t")
        for i in range(1, batch.src.size(0)):
            sym = SRC.vocab.itos[src[0, i]]
            if sym == "</s>" or sym == '<blank>': break
            print(sym, end=" ")
            source += sym + " "
        print("Target      :", end="\t")
        for i in range(1, batch.trg.size(0)):
            sym = TGT.vocab.itos[trg[0, i]]
            if sym == "</s>": break
            print(sym, end=" ")
        trans = ""
        print("Translation :", end="\t")
        for i in range(1, out.size(1)):
            sym = TGT.vocab.itos[out[0, i]]
            if sym == "</s>": break
            print(sym, end=" ")
            trans += sym + " "
        print()
```

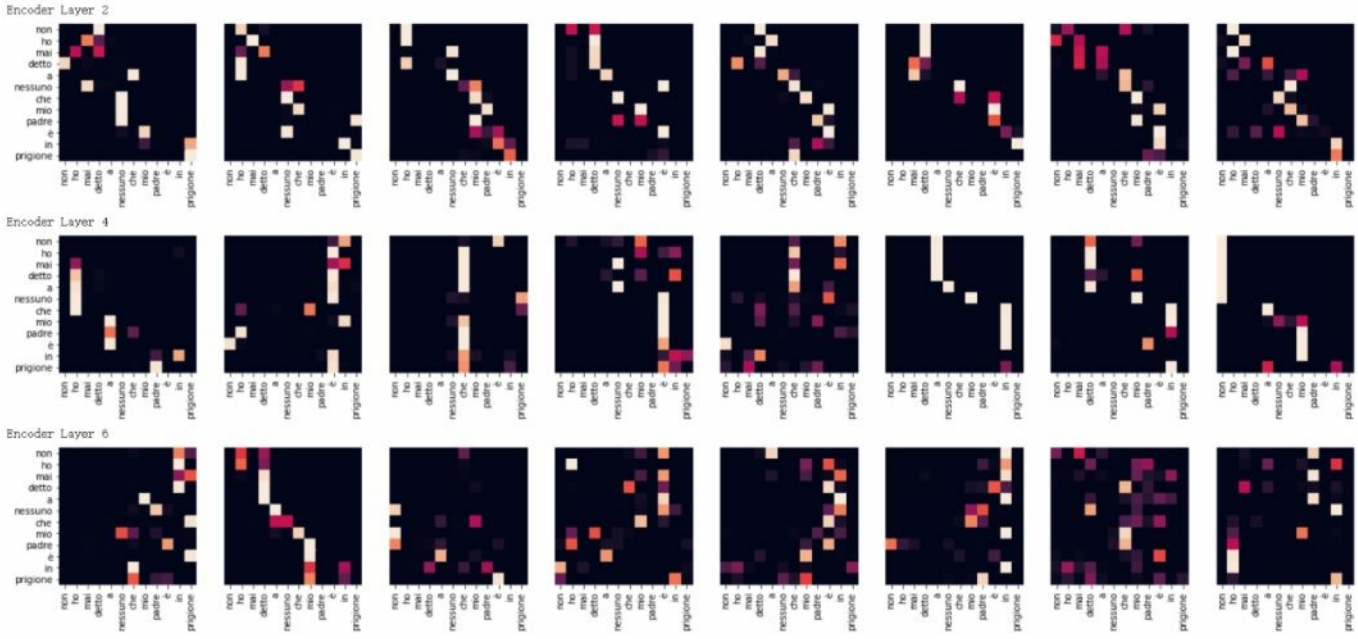
Source : non ho mai detto a nessuno che mio padre è in prigione .

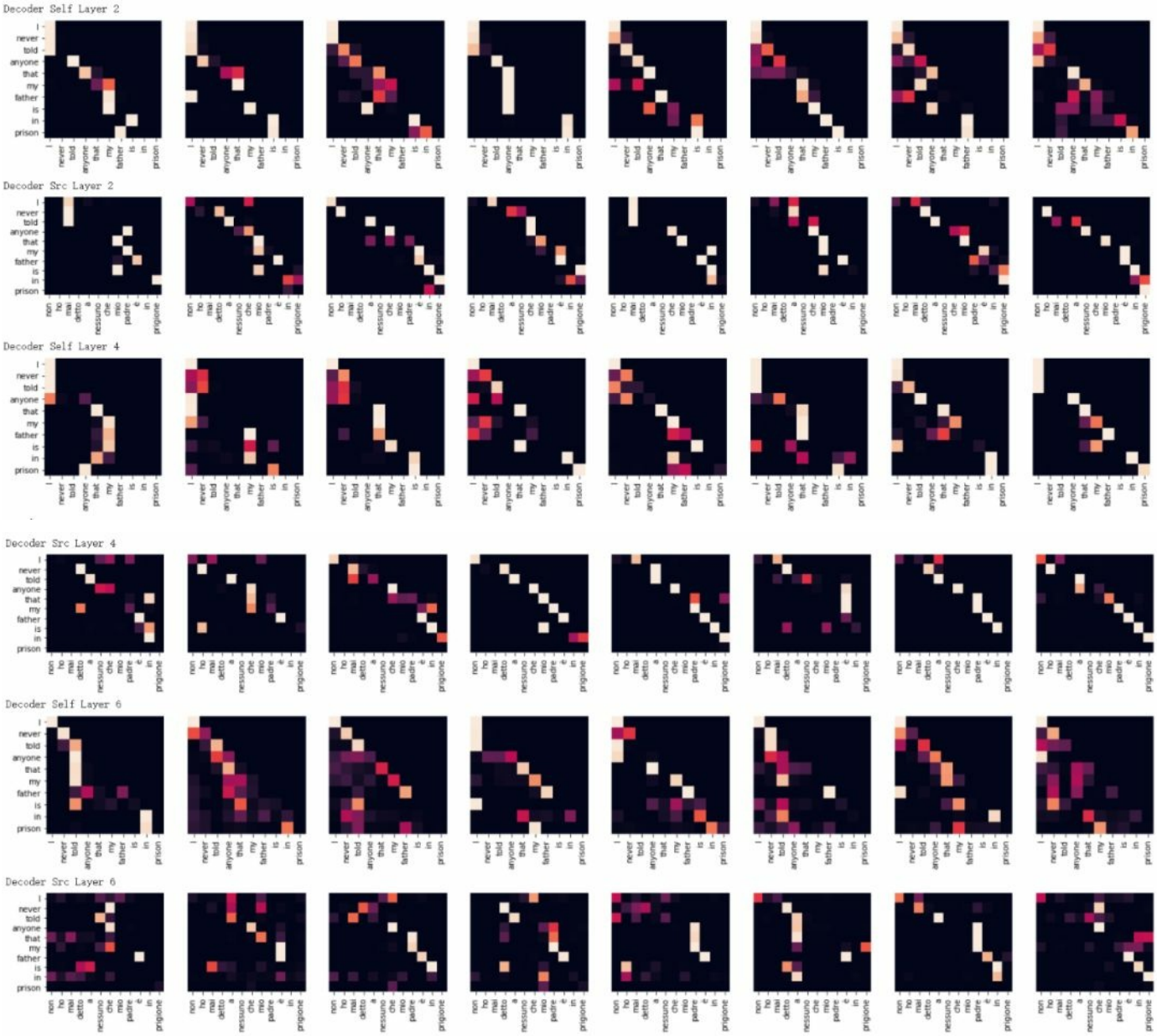
Target : I 've never told anyone that my father is in prison .

Translation : I never told anyone that my father is in prison .

可视化：

```
tgt_sent = trans.split()  # 翻译数据
sent = source.split()  # 源数据src
def draw(data, x, y, ax):
    seaborn.heatmap(data, xticklabels=x, square=True, yticklabels=y, vmin=0.0, vmax=1.0, cbar=False, ax=ax)
for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1, 8, figsize=(25, 15))
    print("Encoder Layer", layer + 1)
    for h in range(8):
        draw(model.encoder.layers[layer].self_attn.attn[0, h].data[:12, :12],
             sent, sent if h == 0 else [], ax=axs[h])
    plt.show()
for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1, 8, figsize=(25, 15))
    print("Decoder Self Layer", layer + 1)
    for h in range(8):
        draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(tgt_sent)],
             tgt_sent, tgt_sent if h == 0 else [], ax=axs[h])
    plt.show()
    print("Decoder Src Layer", layer + 1)
    fig, axs = plt.subplots(1, 8, figsize=(25, 15))
    for h in range(8):
        draw(model.decoder.layers[layer].src_attn.attn[0, h].data[:len(tgt_sent), :len(sent)],
             sent, tgt_sent if h == 0 else [], ax=axs[h])
    plt.show()
```





可以看到8个头在不同注意力层里分布情况，实际上在不同的子空间学习到了不同的信息。

13.数学原理解释Transformer和RNN本质区别

至此，大家应该可以感受到Transformer之所以横扫碾压RNN，其实是多个机制大力出奇迹的成果，并不单单是attention的应用。

但我们深一步思考下，Transformer可以把这么多机制组合在一起而性能没有下降是为什么呢，我个人觉得还是attention的应用大大提升了模型并行运算，但是只是用attention精度可能并不如人意，所以attention省下的空间和时间可以把其他能提高精度的模块(比如position encoding、residual、mask、multi-head等等)一起添加进来。所以从这个角度来讲，attention还是transformer最核心的部分，这个大家应该没有异议的。

再深入一下，不管是attention还是传统的rnn，其实都是为了在计算序列的hidden，RNN使用gate(sigmoid)的概念，计算hidden的权重；而attention使用softmax来计算hidden的权重，无论RNN还是attention他们计算完权重都是为了共同的目标——求得上下文context。

先看下RNN的数学公式。

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$

$$\Gamma_u = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$

$$\Gamma_f = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$

$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \times c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \times \tilde{c}^{\langle t \rangle}$$

$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \times \tanh(c^{\langle t \rangle})$$

再看下attention 相关数学公式：

$$softmax(\frac{QK^T}{\sqrt{d_k}})V$$

大家有没有发现呢？RNN求得各种门是不是很像softmax求得的权重分布？这里RNN里c<t-1>和c^(t)可以类比attention公式中的V，他们都是hidden的含义。

那么我们再仔细看下RNN的门和attention的权重，是不是也能很像，都是对(query,key)使用了非线性激活函数，前者使用了sigmoid，后者使用了softmax，不管使用哪个激活函数activation，其实目的都是再寻找（query,key）之间的相似度，RNN使用了加性运算（W(a,x)+b），而Transformer使用的是乘性运算（QK^T）。

至此是不是恍然大悟呢，这两个经典模型追踪溯源竟然只是sigmoid和softmax的区别。那么我们再回顾下这两个函数：

sigmod计算是标量，而transformer计算的是向量，my god，这不正好符合rnn和transformer的特性么？

rnn使用的是step by step顺序算法，每次都是 计算当前input(query)和上一个cell传来的hidden(key)的关系，由于一次只能喂入一个，所以自然使用sigmoid的标量属性；但是softmax不同，他针对的是一个向量，transformer里的key可不就是一个序列所有的值么，他们的和为1，计算这个序列向量的权重分布，也就是所谓的并行计算。

网上很多人探讨两者的区别，但总让我有种隔靴搔痒的感觉，花了点时间从数学原理的角度感知了两者底层的本质，让我对（QUERY,KEY,VALUE）模式有了更深的理解，希望对大家也有所帮助~

END