Transformer-recognition 算法介绍

目录

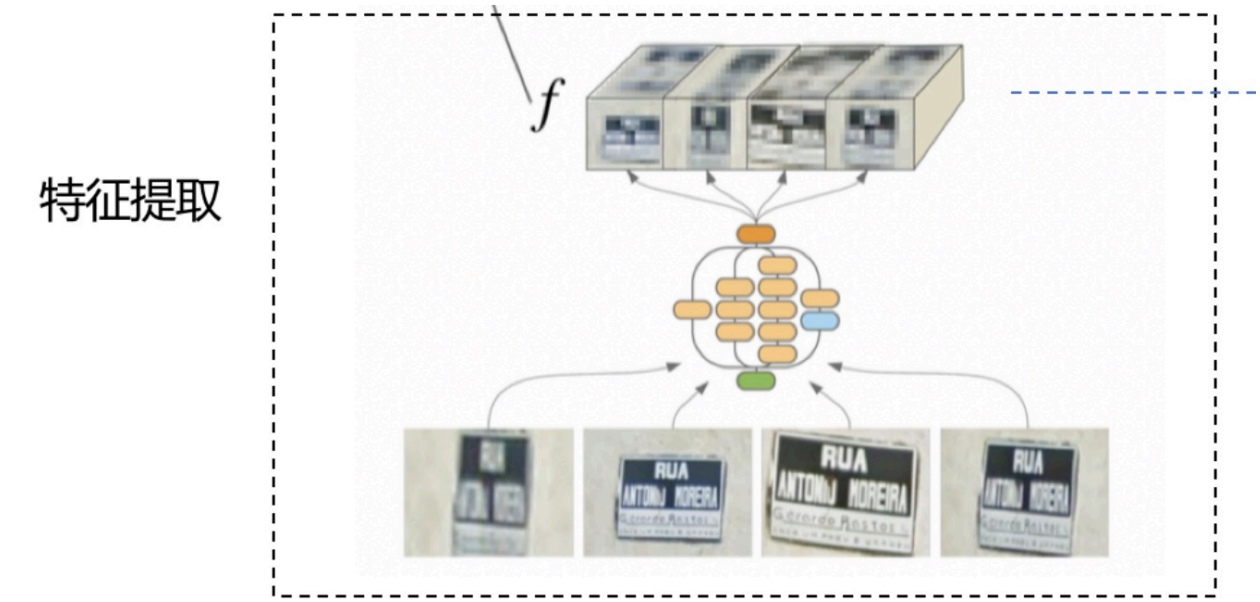
1. 基于seq2seq的Text Recognition

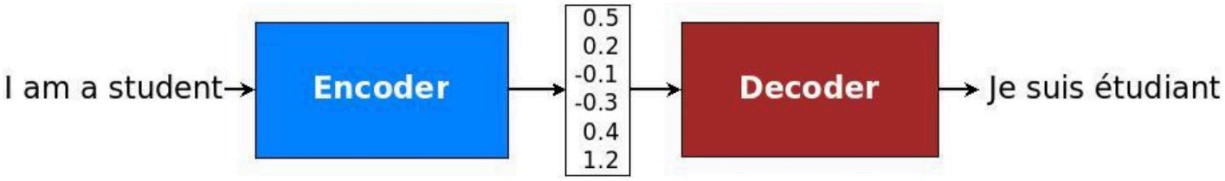
Seq2seq(Sequence to sequence)是NLP领域最常用的一类模型。它用于将序列输入映射为序列输出，典型的应用场景就是**机器翻译**，**语音识别**。

在Text Recognition中，输入时一个图片，输出是一个序列(句子)。一般分为两个步骤: 1) 使用一个CNN Backbone对图片提取时序特征; 2)采用某种方法将这个特征映射为句子。步骤2一般可以采用CTC，最近的一些工作步骤2一般采用seq2seq框架。

输入为 $Img \in R^{H \times W \times 3}$(**所有参数的batch_size维度都省略**),

1. 使用Backbone 提取时序特征: $F_{img} = Backbone(Img) \in R^{T \times C}$
2. 使用Seq2seq框架学习映射: $Y = Seq2seq(F_{img}) \in R^{T_{label} \times Classes}$





Backbone一般采用经典的网络如ResNet50, DenseNet等，对Img提取的特征一般为$F_{net} \in R^{H \times W \times C_{net}}$,为了得到上面的时序特征$F_{img} = \in R^{T \times C}$, 效果比较好的处理方法有两种:

- 将高(H)这个维度reshape到channel(C)这个维度中，即$F_{net} \in R^{H \times W \times C_{net}} \rightarrow F_{img} \in R^{W \times (H \times C_{net})} = R^{T \times C'}$, 一般$C'$维度较高，可以通过一个全连接层将$C'$维度降低为期待的特征维度$C$. 比如图片输入$32 \times 240$采用ResNet50中$C' = 4 * 2048 = 9162$,通过全连接层将其降维为$C = 512$，最终特征$F_{img} \in R^{30 \times 512}$.
- 将高(H)和(W)这两个维度reshape为时间维度T中，即$H \times W \rightarrow T$. 这种做法可能对于不规则字符的图片比较有效，因为他保留了横向和纵向的信息。

Backbone提取的时序特征即为seq2seq框架的输入

下面首先简要介绍一下seq2seq框架和引入attention的seq2seq，最后介绍一下Transformer的结构。

1.1 Seq2seq

Seq2seq 可以分为Encoder和Decoder两个部分。如下图所示，蓝色的是Encoder,红色的是Decoder。

1. Encoder对图片特征F_{img}进一步提取，主要是学习时序t之间的关系，学习到的特征为

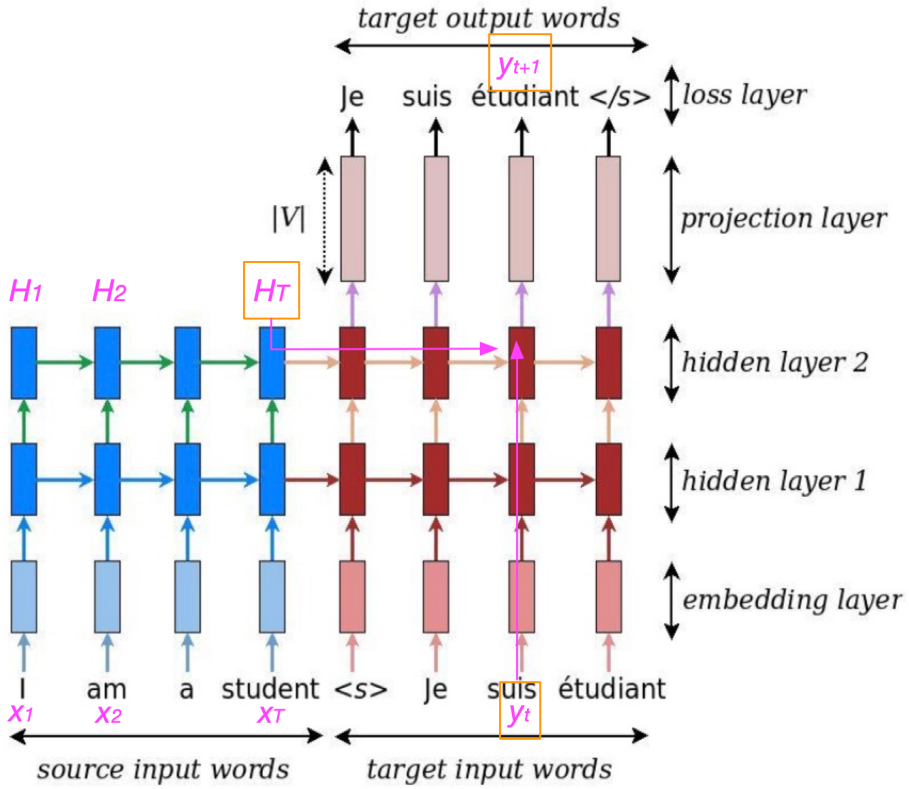
$$H_{1:T} = Encoder(F_{img}) \in R^{T \times C_e}$$

其中 $H = H_{1:T} = \{H_1, H_2, \ldots, H_T\}$, $1:T$代表从t=1到t=T的特征,下同。

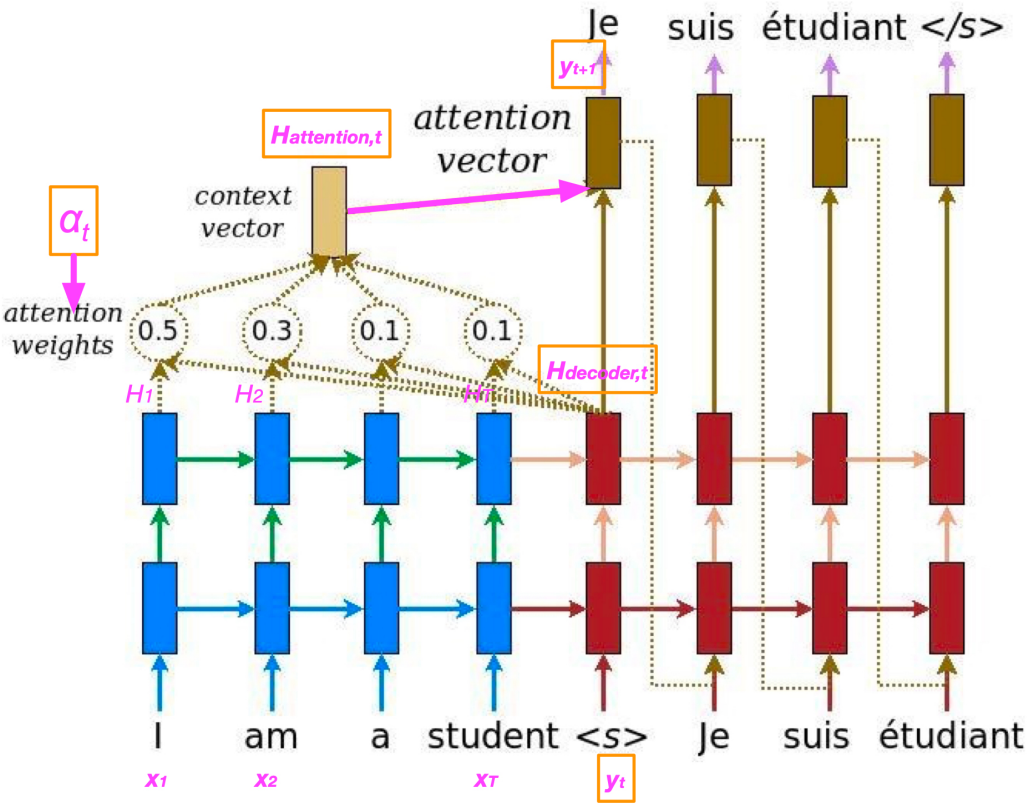
2. Decoder是对H进行解码，得到目标label,在ocr里就是需要识别的句子。它的输入有两个，一个是Encoder提取的特征，另外一个之前预测的label。 解码过程类似于RNN计算过程，第t+1个字符的计算依赖于第t个字符，解码开始于预定义标签[SOS]，终止于[EOS](字典中额外添加这两个标签和一个[PAD]标签). 解码过程:

$$y_{t+1} = Decoder([H_T, y_t])$$

训练时采用有师学习的方法，即对$y_{1:T}$的最开始补一个[SOS]作为Decoder的输入，$y_{1:T}$最后补一个[EOS]作为Decoder的输出，通过cross_entropy作为loss训练。预测时则一个一个的解码直到出现[EOS]. 详细介绍可以参考google nmt tutorial



1.2 Attention-based seq2seq



在1.1中的seq2seq，解码时在不同时间t均依赖于同一个Encoder的特征H_T，**直觉上的改进方法在不同的时间t应该采用不同的Encoder的特征。这就是Attention机制。**如上图所示，针对不同的时间t，利用$H_{1:T}$生成不同的特征$H_{attention,t}$，然后进行解码:

$$y_{t+1} = Decoder([H_{attention,t}, y_t])$$

$H_{attention,t}$是通过$H_{1:T}$和$H_{decoder,t}$(Decoder的隐层时间t处的输出特征)计算出来的，也即根据$H_{decoder,t}$计算出时间t的字符与$H_{1:T}$的每个时间的相关程度(注意力矩阵) $\alpha \in R^T$, 这通常是通过一个注意力矩阵W计算得到，如Luong attention计算方式:

$$\alpha_t = Softmax((H_{decoder,t})WH)$$

$$H_{attention,t} = reduce\_sum(\alpha_t \odot H)$$

现在一般采用多层的attention机制，也即对Decoder的多个隐层的输出$H^0_{decoder}, H^1_{decoder}, \ldots, H^l_{decoder}$相对于Encoder的输出特征$H_{1:T}$进行attention。
Decoder的第i层输出:

$$H^i_{decoder,1:T} = H^i_{decoder,1:T} + H^i_{attention,1:T}$$

Encoder和Decoder的实现各有不同，但基本可以分为几类:

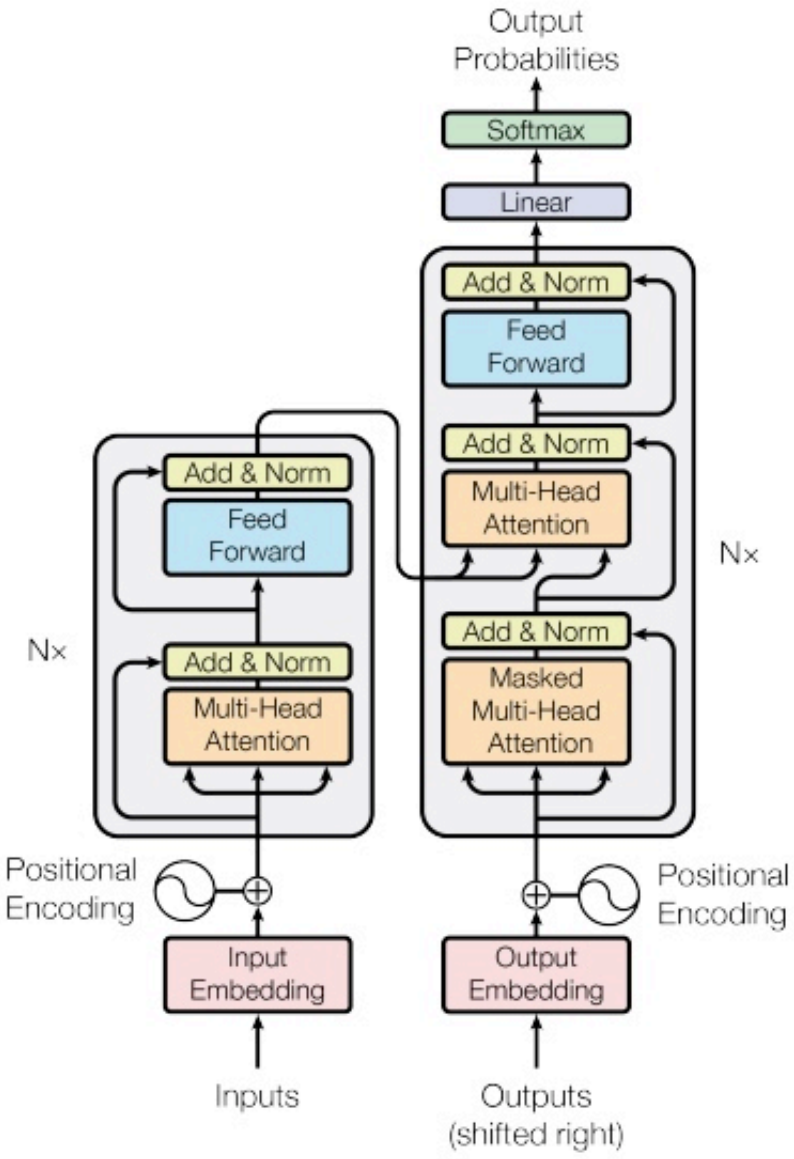
1. 用RNN实现(传统Seq2seq)
2. 以Conv1D结构为核心实现(Convolutional Seq2seq, Facebook FAIR, ICML 2017)

3. 用Dense+self-attention实现 (下面会讲到的Transformer, Google BRAIN, NIPS 2017)
4. 以及之上的方法的各种变体

2. Transformer

Seq2seq框架主流的结构演变流程: RNN-seq2seq -> Attention-based RNN-seq2seq -> Attention-based Conv-seq2seq -> Self&Vanilla Attention Densely-connected Seq2seq(Transformer)

Transformer是Google NIPS2017的工作，一经提出就在NLP领域被广泛采用，被奉为NLP领域3大特征提取神奇之一(另外两大忘记了...)。它的结构如图所示:



它采用分层Attention结构: Encoder和Decoder都有N层,每一层的每一个子层(attention层，Feed-Forward层)都采用了残差结构($x = x + layer(x)$)和Layer normalization。

1. Encoder: 对该层的输入x进行Self-Attention，然后经过Feed-Forward作为该层的输出，这个输出会作为Encoder下一层的输入和Decoder同层Attention的输入(Query和Key)。
2. Decoder: 第0层的输入为Embedding后的句子。对每一层的输入进行一次Self-Attention，然后进行一次Encoder-Decoder Attention, 最后Feed-Forward输出作为下一层的输入。

下面会详细讲解上面提到的Self-Attention, Encoder-Decoder Attention, Feed-Forward, 以及图中的Positional Encoding, Multi-Head Attention.

2.1 Multi-Head Attention

Multi-Head Attention本质上并不是Attention,而是一个AttentionWrapper。它的思想就是对输入x的channel维度split为多个头(head)，每个头都做一个Attention,最后将这些Attention的输出concat回来。Multi-Head Attention的引入可以让模型可以在同一层同时学习到不同位置的不同表达子空间，通俗的讲就是同时进行多个Attention可以学习到更多样的表达。具体操作过程下面公式很容易理解:

$$\begin{aligned}\text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,...,\text{head}_\text{h})W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

2.2 Self-Attention& Encoder-Decoder Attention

2.2.1 Transformer中的Attention

这两个Attention在形式上和参数上都是一样的，唯一的区别是输入不同。**Transformer这篇文章关于Attention的定义非常容易理解，强烈推荐仔细琢磨**。其它文章中Attention实质上就是该文的Encoder-Decoder Attention，只是描述上不一致，但该文的描述更易理解。

该文将Attention进行了一个抽象。任何Attention机制都可以描述为通过一个query 将一个 key-value对 映射为输出, query、keys、values都是向量。输出是对value的加权和，权重是 query对key的相关程度。公式化描述更容易理解:

给定$Q \in R^{T_o \times C_k}, K \in R^{T_i \times C_k}, V \in R^{T_i \times C_v}$，那么

$$Score = \frac{QK^T}{\sqrt{C_k}} \in R^{T_o \times T_i}$$
$$Attention(Q, K, V) = Softmax(Score)V \in R^{T_o \times C_v}$$

$\sqrt{C_k}$为归一化系数。$Score$的每一个行向量$Score_t \in R^{T_i}$即为Q_t关于$K_{1:T_i}$的每一个时间的特征(列向量)的权重，$Softmax(Score)V$即为$V_{1:T_i}$的每一个时间特征的加权和,其中Softmax用于对权重归一化。

- **在Self-Attention中，Q=K=V均为上一层的输出。** Self-Attention学的是时间维度上的每一个特征与其它时间特征的相关性，并按照相关程度对所有特征加权和加到这个特征上来。
- **在Encoder-Decoder Attention 中， Q为Decoder上一层的输出，K=V为Encoder的同层输出。** Encoder-Decoder Attention学习的是Decoder每一个时间维度上的特征与Encoder的特征的相关性，并把Encoder的特征加权和以后加到Decoder的这个特征上来。

2.2.2 与其它论文中的Attention的一致性

注意到公式中没有可训练的参数W之类的。实际上，它是通过全连接层分别对Q, K, V进行学习，这个全连接层的参数就是Attention的参数，也是Multi-Head Attention中的W_i^Q, W_i^K, W_i^V等:

$$Q = QW_q, W_q \in R^{C_k, C_k}$$
$$K = KW_k, V = VW_v$$

那么$Score = Softmax(\frac{QK^T}{\sqrt{C_k}}) = Softmax(\frac{QW_qW_k^TK^T}{\sqrt{C_k}}) = Softmax(QWK^T)$, 其中$W = W_qW_k^T/\sqrt{(C_k)}$, **这个形式与常用的乘性Attention(Dot Attention) Luong Attention 一致的**:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases}$$

Bahdanau Attention 是加性Attention,但是在Transformer中选择了乘性Attention。

2.2.3 Why Self-Attention

Transformer只用了Dense层，而别人的seq2seq框架用了Conv、RNN等高科技，为什么Transformer效果最好呢？这主要归功于Self-Attention的强大特征提取能力。
比如，一个序列的时间长度为n, 使用RNN需要传递n步才能学习到t=1和t=n的两个特征之间的关系，使用CNN需要传递$log_k(n)$步才能传递过去。这中间还存在着信息丢失和错误增加情况。而

对于Self-Attention，却只需要一步操作即可学习到这两个特征之间的关系!因为他直接对这两个特征向量进行矩阵乘进行计算。在堆叠相同层数的情况下，Self-Attention显然具有更宽视野的表达能力，而且不存在梯度消失和爆炸问题。具体详细分析请见原论文。

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

2.3 Feed-Forward

Feed-Forward层其实就是两个全连接层加上一个残差连接。第一个全连接层对特征维度进行升高，第二个全连接层将特征维度还原:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

参数设置示例： 特征维度$d(x) = 512$, 第一个全连接层神经元个数设为2048，第二个全连接层神经元个数设为512。

2.4 Positional Encoding

在没采用RNN的seq2seq框架中，由于Dense层不会保留位置信息(时序信息)，因此需要位置编码(positional encoding)来告知**每个特征他们自己的绝对位置和相对位置**。甚至在基于RNN的seq2seq框架中，合适的位置编码也会带来性能的提升。

$$x = x + PositionalEncoding(x)$$

文章没有对位置编码进行过多的研究，但是进行了经验尝试，最终选择的位置编码如下:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

pos为当前的位置，$1 \leq pos \leq T$。 $2i, 2i+1$为特征维度($[1, C]$)

为什么需要位置编码?

对于由Dense层提取的两个特征x_i和x_j(列向量，$\in C \times 1$), 特征本身是对位置信息没有任何感知的，因为Dense只进行特征内的矩阵乘，换个位置结果一样。

在进行self-attention时，两个特征相关性为：

$$coef_{ij} = x_i^T x_j = x_j^T x_i$$

即使x_i和x_j调换位置，self-attention也不会产生变化，但是单词在句子中出现的位置是有极其重要的影响的。
因此需要加入位置编码告诉self-attention位置的信息.相关性变为:

$$coef_{ij} = (x_i + pe_i)^T (x_j + pe_j) \neq (x_j + pe_i)^T (x_i + pe_j)$$

式中最后一项为调换x_i和x_j位置后的相关性$coef_{ij}$

3 Transformer Recognition

这一节主要对Transformer Recognition的参数和训练进行介绍。

3.1 模型细节

代码仓库在code.

- Backbone: 模型采用ResNet50作为Backbone, 不要最后的pooling层和fc层，ResNet输出特征维度为[H/8, W/8, 2048];接着，一个卷积核数量为`hidden_size=512`的1x1卷积层、ReLu激活层对特征进行降维为[H/8, W/8, hidden_size]; 然后将高这个维度reshape到channel中: [W/8, H/8*hidden_size]; 最后连一个ReLu激活的全连接层将特征降维`feature_len=512`为: [W/8, feature_len]。
- Transformer: 将上面的特征连到Transformer中。大部分参数与原论文推荐一致，但是将Attention块堆叠层数从6层减少为3层。其它参数如下所示:

```
# training
label_smoothing: 0.1    #perform label smoothing to the labels.

# model
hidden_size: 512        # hidden_size
# backbone
feature_len: 512        # length of backbone feature
# transformer
initializer_gain: 1.0    # Used in trainable variable initialization.
```

```
num_hidden_layers: 3    # Number of layers in the encoder and decoder stacks
.
num_heads: 8    # Number of heads to use in multi-headed attention.
filter_size: 1024    # Inner layer dimension in the feedforward network.
# dropout values (only used when training)
layer_postprocess_dropout: 0.1
attention_dropout: 0.1
relu_dropout: 0.1

# decode
beam_size: 4
extra_decode_length: 10
alpha: 0.6
allow_ffn_pad: True
```

3.2 训练

3.2.1 输入尺寸

训练时不同数据集由于图片大小不同采用不同尺寸的图片，仅需要在模型参数中指定即可。

```
# input
# 手写体 hw_freechars
img_size: [32, 300]

# 庆杰生成的 printing_freechars
# 由于图片较长，为了保存图片比例，降低高度为24减小计算量。([有待证实]增大尺寸理论上会带来性能提升)
# img_size: [24, 600]
img_ext: "jpg"
```

模型的Backbone初始化为在Imagenet上训练的权重，其余参数随机初始化。([有待证实]对于较大的数据集，ImageNet初始化是否会有性能提升？)

3.2.2学习率

这里采用Adam优化器，学习率参数:

```
# optimizer
learning_rate: 0.3
learning_rate_decay_rate: 0.8
learning_rate_warmup_steps: 2000
learning_rate_decay_steps: 32000    # = 2 epoches
```

```
learning_rate_decay_staircase: True
optimizer_adam_beta1: 0.9
optimizer_adam_beta2: 0.997
optimizer_adam_epsilon: 1e-09
```

学习率调度方法是采用warmup和exp decay。即最开始 `learning_rate_warmup_steps` 个step学习率线性增加到最大，然后每隔 `learning_rate_decay_steps` 个step学习率乘以 `learning_rate_decay_rate` . `learning_rate_decay_steps` 设置为2个epoch的步数, 即每2个epoch学习率衰减一次。

最大学习率为$max_lr = \frac{learning_rate}{\sqrt{hidden_size} \times \sqrt{learning_rate_warmup_steps}}$

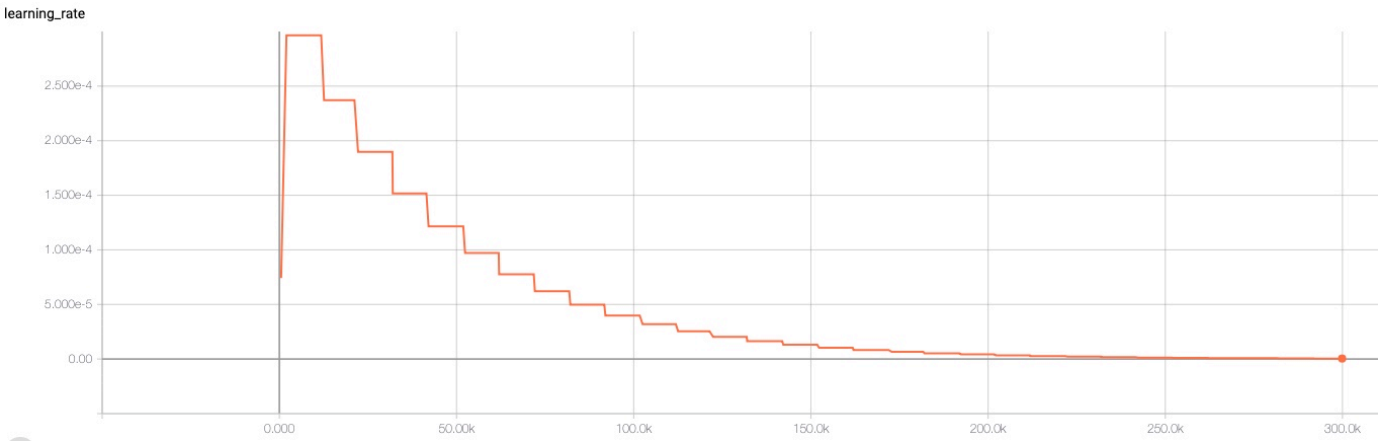
```
# step 为 当前step
warmup_lr = lambda: max_lr * tf.minimum(1.0, step / warmup_steps)
exp_decay_lr = tf.train.exponential_decay(max_lr,
                    step - learning_rate_warmup_steps,
                    decay_steps=learning_rate_decay_steps,
                    staircase=learning_rate_decay_staircase)
learning_rate = tf.cond(step < learning_rate_warmup_steps,
                    warmup_lr,
                    exp_decay_lr)
```

这里没有对学习率进行精细的调参，仔细调参可能会带来1~2%的提升。

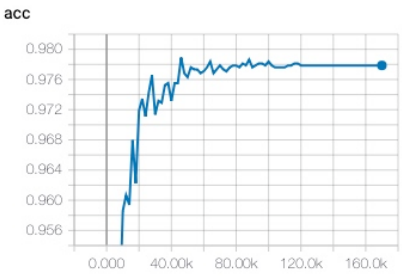
3.2.3 一些训练曲线

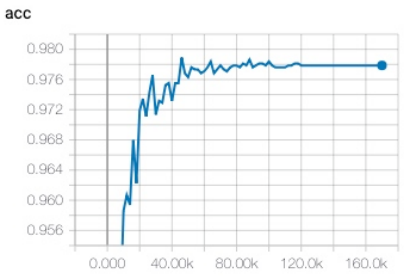
由于采用了exp decay的学习率调度方法，训练过程会非常平稳。

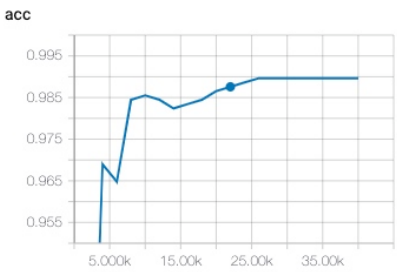
- 学习率调度

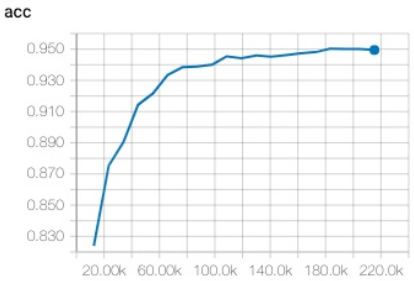


- 一些验证集上的准确率曲线(每个epoch测试一次)









- 一些数据集训练集loss