Attention Is All You Need

Transformer的结构如下图，分为Encoder和Decoder部分，每个部分有N次重复的子结构（论文中N=6）。
后面将描述每个部分的功能，代码部分参考的是Kyubyong的transformer实现。

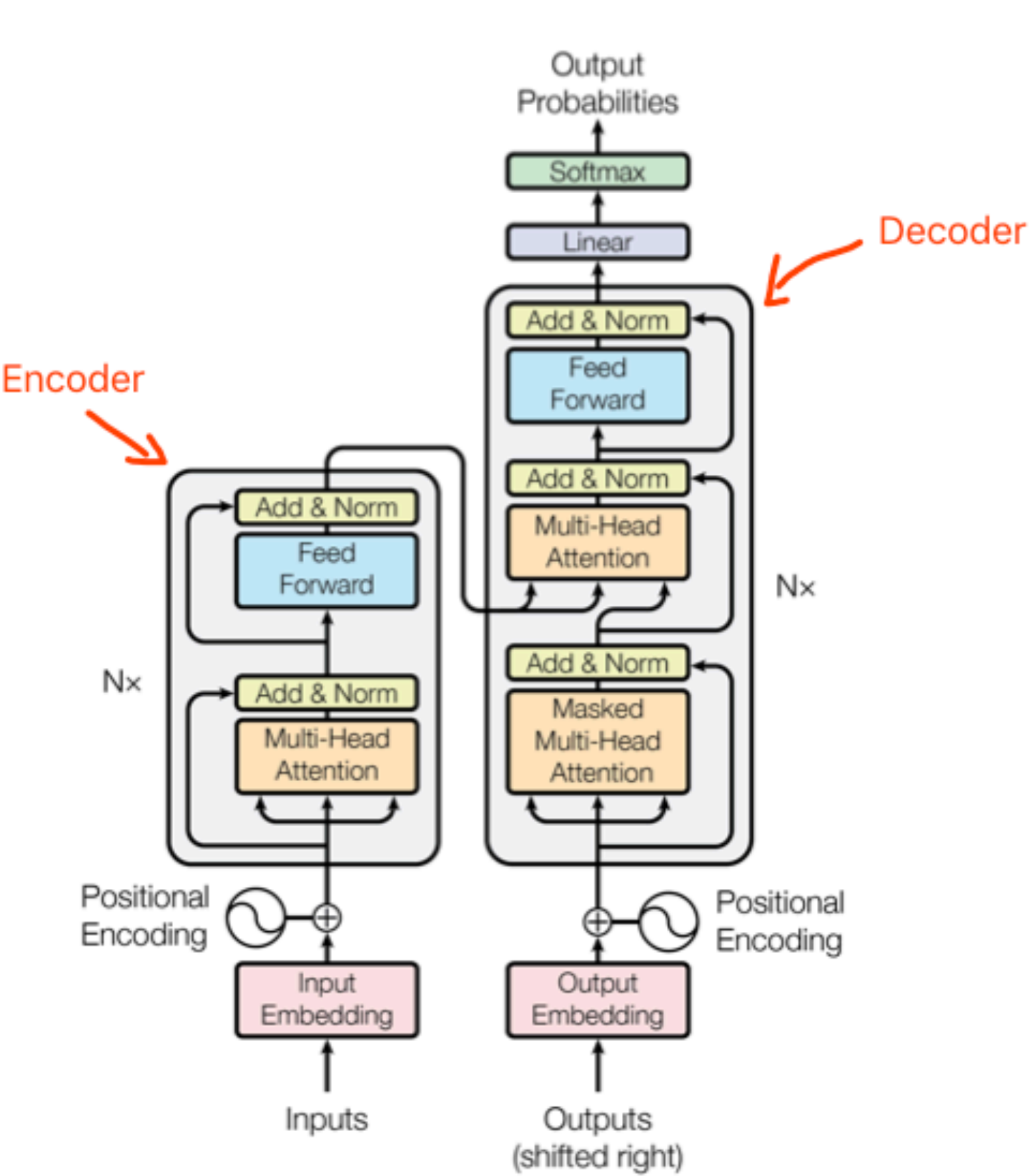


Figure 1: The Transformer - model architecture.

Encoder部分

Input Embedding

就是普通的word embedding，将词转换成词向量。

```
def embedding(inputs,
              vocab_size,
              num_units,
              zero_pad=True,
              scale=True,
              scope="embedding",
              reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        lookup_table = tf.get_variable('lookup_table',
                                       dtype=tf.float32,
                                       shape=[vocab_size,
num_units],

initializer=tf.contrib.layers.xavier_initializer())
        if zero_pad: #解决文本长度不均使用0进行了pad，这里是处理0行的词向
量
            lookup_table = tf.concat((tf.zeros(shape=[1,
num_units]),
                                      lookup_table[1:, :]), 0)
        outputs = tf.nn.embedding_lookup(lookup_table, inputs)

        if scale:
            outputs = outputs * (num_units ** 0.5)

    return outputs
```

Position Encoding

Attention模型并不能捕捉序列的顺序！换句话说，如果将K,V按行打乱顺序（相当于句子中的词序打乱），那么Attention的结果还是一样的。这就表明了，到目前为止，Attention模型顶多是一个非常精妙的“词袋模型”而已。

这问题就比较严重了，大家知道，对于时间序列来说，尤其是对于NLP中的任务来说，顺序是很重要的信息，它代表着局部甚至是全局的结构，学习不到顺序信息，那么效果将会大打折扣（比如机器翻译中，有可能只把每个词都翻译出来了，但是不能组织成合理的句子）。

于是Google再祭出了一招——Position Embedding，也就是“位置向量”，将每个位置编号，然后每个编号对应一个向量，通过结合位置向量和词向量，就给每个词都引入了一定的位置信息，这样Attention就可以分辨出不同位置的词了。然而也有说Attention无法对位置信息进行很好地建模，这是硬伤。尽管可以引入Position Embedding，但这只是一个缓解方案，并没有根本解决问题。

论文中提到了两种方式，方法二就是根据每个词的位置进行编号（0–squence_len），然后跑一下embedding。

```
## Positional Encoding
if hp.sinusoid: #方法一 sinusoidal version
    self.enc += positional_encoding(self.x,
                      num_units=hp.hidden_units,
                      zero_pad=False,
                      scale=False,
                      scope="enc_pe")
else: #方法二
    self.enc +=
embedding(tf.tile(tf.expand_dims(tf.range(tf.shape(self.x)[1]),
0), [tf.shape(self.x)[0], 1]),
                      vocab_size=hp.maxlen,
                      num_units=hp.hidden_units,
                      zero_pad=False,
                      scale=False,
                      scope="enc_pe")
```

而方法一(sinusoidal version)则是Google直接给出了一个构造Position Embedding的公式:

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

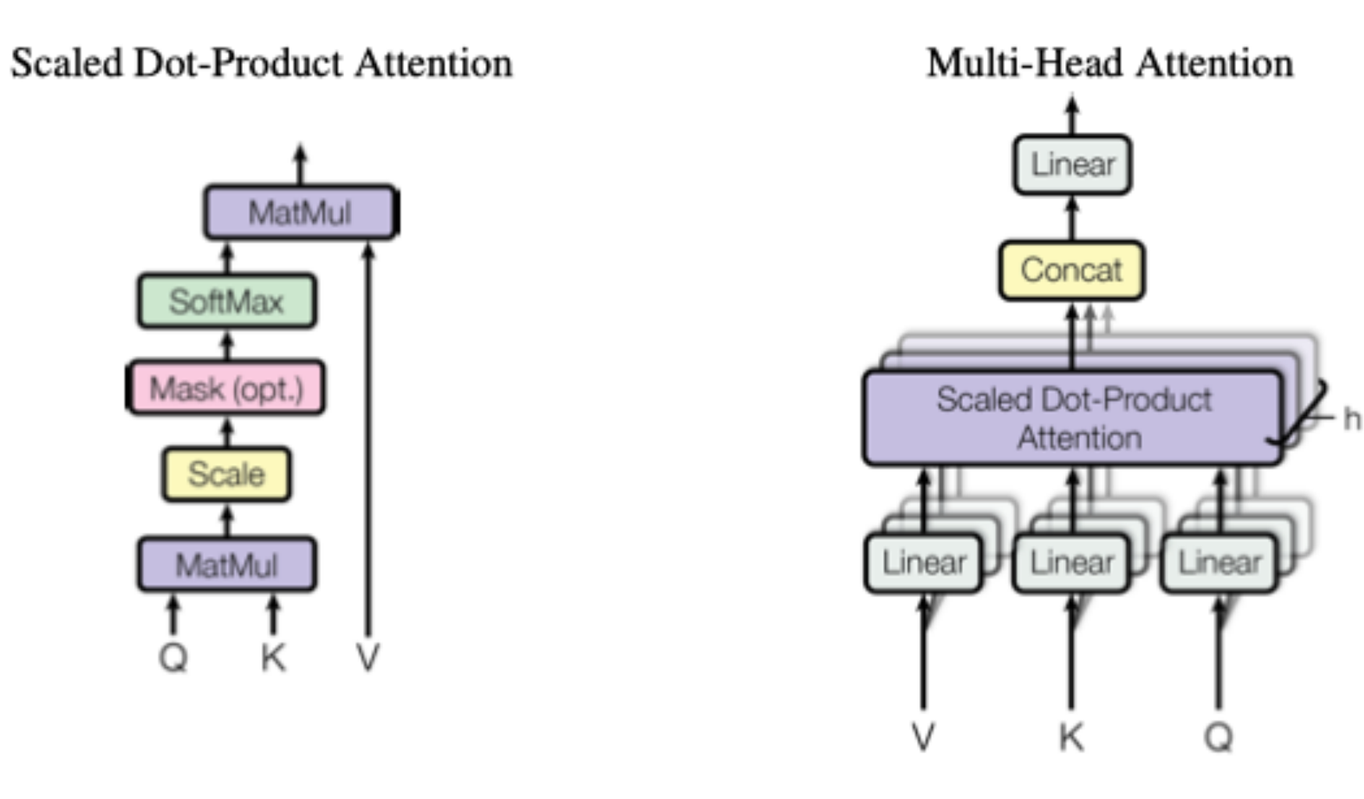
pos代表word在序列中的位置，而i表示的是word的embedding中的位置，d_{model} 表示的是embedding的长度。

```
def positional_encoding(inputs,
                        num_units,
                        zero_pad=True,
                        scale=True,
                        scope="positional_encoding",
                        reuse=None):
    N, T = inputs.get_shape().as_list()
    with tf.variable_scope(scope, reuse=reuse):
        position_ind = tf.tile(tf.expand_dims(tf.range(T), 0), [N,
1])

        # First part of the PE function: sin and cos argument
        position_enc = np.array([
            [pos / np.power(10000, 2.*i/num_units) for i in
range(num_units)]
            for pos in range(T)])

        # Second part, apply the cosine to even columns and sin to
odds.
        #对应于上面的公式
        position_enc[:, 0::2] = np.sin(position_enc[:, 0::2])  #
dim 2i
        position_enc[:, 1::2] = np.cos(position_enc[:, 1::2])  #
dim 2i+1

        # Convert to a tensor
        lookup_table = tf.convert_to_tensor(position_enc)

        if zero_pad:
            lookup_table = tf.concat((tf.zeros(shape=[1,
num_units]),
                                      lookup_table[1:, :]), 0)
        outputs = tf.nn.embedding_lookup(lookup_table,
position_ind)

        if scale:
            outputs = outputs * num_units**0.5

        return outputs
```

论文中提到方法一相对于方法二能承受更长的序列，而且效果来说差不多。

> 1、We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.
> 2、In row (E) we replace our sinusoidal positional encoding with learned positional embeddings [9], and observe nearly identical results to the base model.

Attention



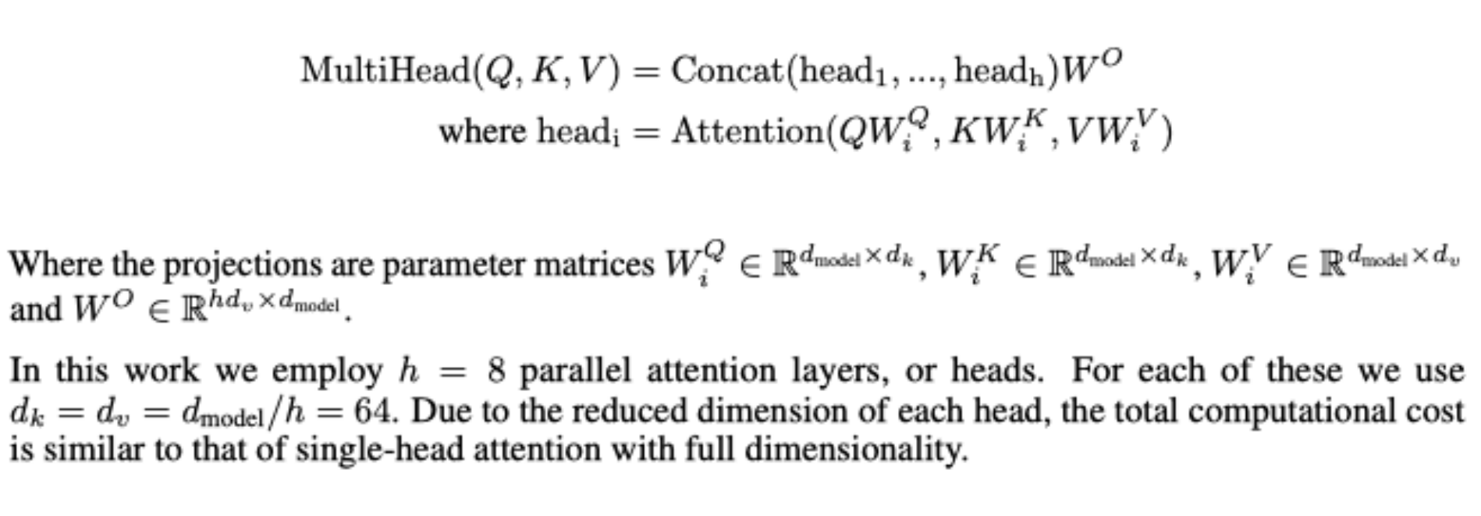
Scaled Dot-Product Attention

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

其中的mask是为了忽略填充部分的影响。一般的Mask是将填充部分置零，但Attention中的Mask是要在softmax之前，把填充部分减去一个大整数（这样softmax之后就非常接近0了）。

```
# Key Masking
#0代表原先的keys第三维度所有值都为0，反之则为1，我们要mask的就是这些为0的key
key_masks = tf.sign(tf.abs(tf.reduce_sum(keys, axis=-1))) # (N,
T_k)
key_masks = tf.tile(key_masks, [num_heads, 1]) # (h*N, T_k)
key_masks = tf.tile(tf.expand_dims(key_masks, 1), [1,
tf.shape(queries)[1], 1]) # (h*N, T_q, T_k)

paddings = tf.ones_like(outputs)*(-2**32+1)
#将为0的换成极小值
outputs = tf.where(tf.equal(key_masks, 0), paddings, outputs) #
(h*N, T_q, T_k)
```

Multi-Head Attention



$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},...,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

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.

Attention虽然跟CNN没有直接联系，但事实上充分借鉴了CNN的思想，比如Multi-Head Attention就是Attention做多次然后拼接，这跟CNN中的多个卷积核的思想是一致的；还有论文用到了残差结构，这也源于CNN网络。
论文中每个attention维度大小为原来的d_{model}/h，所以计算量与维度为d_{model}的single-head差不多。

Position-wise Feed-Forward Networks

就是窗口为1的卷积，用的relu激活。

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

```
def feedforward(inputs,
                num_units=[2048, 512],
                scope="multihead_attention",
                reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        # Inner layer
        params = {"inputs": inputs, "filters": num_units[0],
"kernel_size": 1,
                  "activation": tf.nn.relu, "use_bias": True}
        outputs = tf.layers.conv1d(**params)

        # Readout layer
        params = {"inputs": outputs, "filters": num_units[1],
"kernel_size": 1,
                  "activation": None, "use_bias": True}
        outputs = tf.layers.conv1d(**params)

        # Residual connection
        outputs += inputs

        # Normalize
        outputs = normalize(outputs)

    return outputs
```

Decoder部分

Decoder部分大部分和encoder的差不多，只有几个部分需要注意。

outputs(shifted right)

self.y来初始化解码器的输入。decoder_inputs和self.y相比，去掉了最后一个句子结束符，而在每句话最前面加了一个初始化为2的id，即<S>，代表开始。

```
 if is_training:
    self.x, self.y, self.num_batch = get_batch_data() # (N, T)
else: # inference
    self.x = tf.placeholder(tf.int32, shape=(None, hp.maxlen))
    self.y = tf.placeholder(tf.int32, shape=(None, hp.maxlen))

# define decoder inputs
self.decoder_inputs = tf.concat((tf.ones_like(self.y[:, :1])*2,
self.y[:, :-1]), -1) # 2:<S>
```

Masked Multi-Head Attention

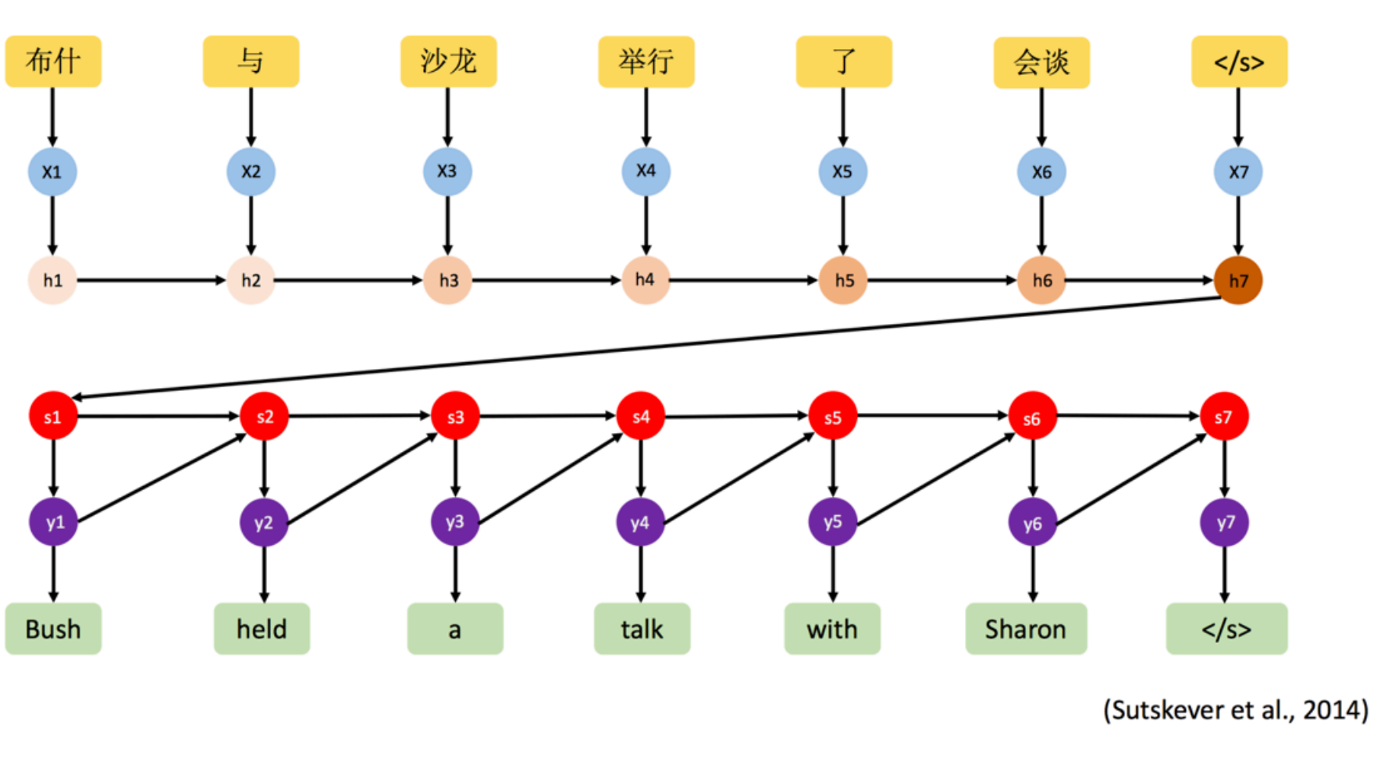
Masked:是否屏蔽未来序列的信息（解码器self attention的时候不能看到自己之后的那些信息），这里即causality为True时的屏蔽操作。

```
# Causality = Future blinding
if causality:
    diag_vals = tf.ones_like(outputs[0, :, :]) # (T_q, T_k)
    #三角阵中，对于每一个T_q,凡是那些大于它角标的T_k值全都为0，这样作为mask
就可以让query只取它之前的key（self attention中query即key）
    tril =
tf.contrib.linalg.LinearOperatorTriL(diag_vals).to_dense() # (T_q,
T_k)
    masks = tf.tile(tf.expand_dims(tril, 0), [tf.shape(outputs)
[0], 1, 1]) # (h*N, T_q, T_k)

    paddings = tf.ones_like(masks)*(-2**32+1)
    outputs = tf.where(tf.equal(masks, 0), paddings, outputs) #
(h*N, T_q, T_k)
```

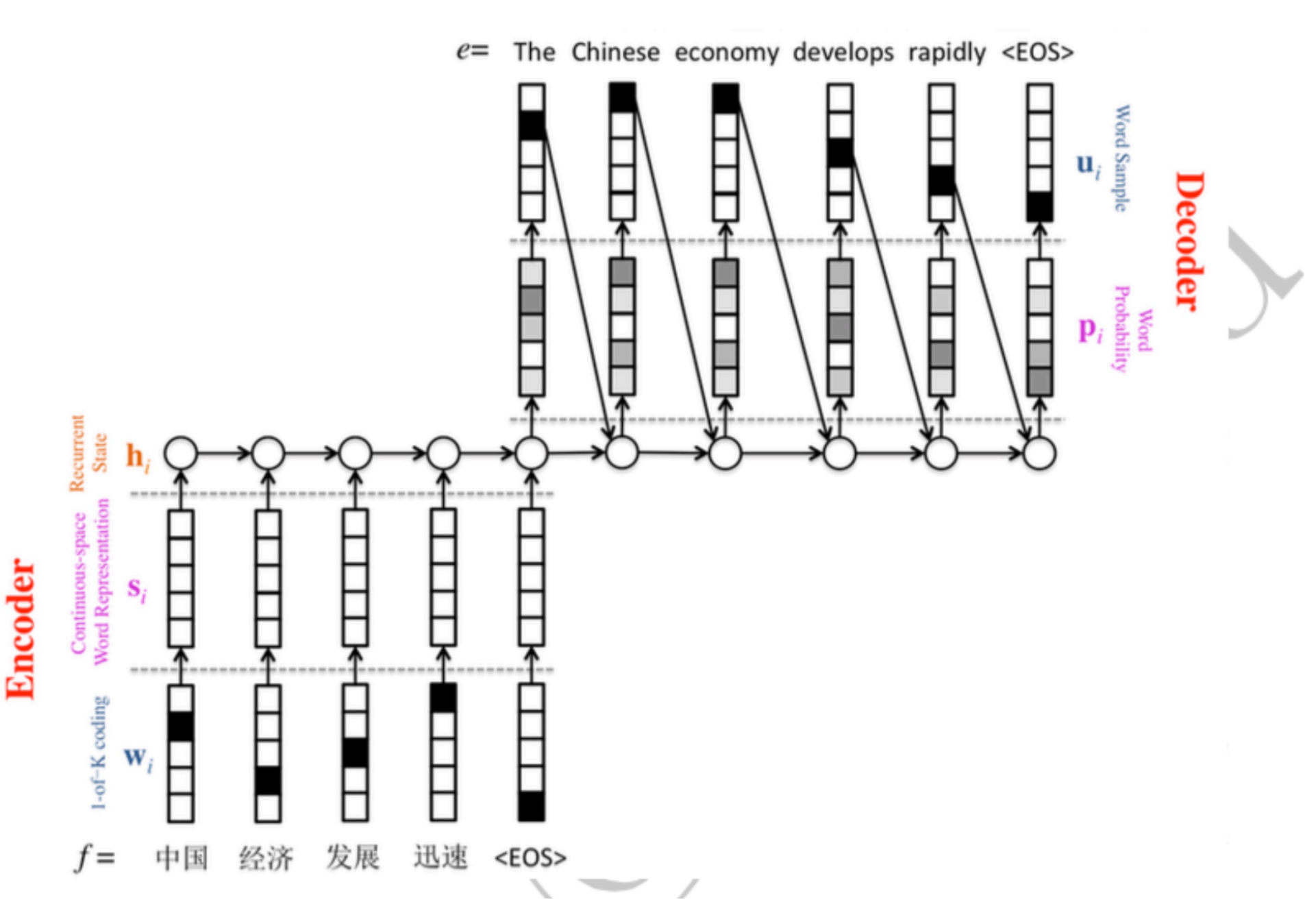
Eval部分

```
### Autoregressive inference
preds = np.zeros((hp.batch_size, hp.maxlen), np.int32)
for j in range(hp.maxlen):
    _preds = sess.run(g.preds, {g.x: x, g.y: preds})
    preds[:, j] = _preds[:, j]
```

之前一直不是特别明白这个迭代的过程，后来知道其decoder是一个词一个词去dencoder的，每个词需要其之前的word的embedding。这也和训练阶段的Masked Multi-Head Attention对应上了。



(Sutskever et al., 2014)



参考

《Attention is All You Need》论文

《Attention is All You Need》浅读（简介+代码）

机器翻译模型Transformer代码详细解析

Kyubyong的transformer实现

论文笔记：Attention is all you need