2022-08

Summary

1. Transformer 基于注意力机制，兼具 RNN 和 CNN 的特点：它既可以像 RNN 那样处理变长的时序数据，其注意力机制又类似于卷积，多头注意力就类似于多个卷积核产生的多个 channel；
2. 除了网络结构，Transformer 还有大量有意思的细节，挺有启发的，比如 subword 分词算法，label smothing，drop out (几乎每一个 sub-layer 后都会进行 drop out，包括 embedding 层，不过 drop out 的概率都不高，只有 0.1)，weight tying，model average 等，其中，可以将 label smothing 引入到 fastText2；
3. BERT 就是一个预训练的特征提取器，模型结构很简单，只用到了 Transformer 的 encoder 部分，关键是它定义的两个预训练任务，通过预训练 + 微调的方式，效果非常好，将 CV 的这样一套流程引入了 NLP；
4. Transformer 除了用在 NLP，还可以用于 CV，也就是 Vision Transformer，似乎 GoldFish 中图像和URL等的处理可以统一用 Transformer 来做，这样都在同样的语义空间，效果可能会更好。

Attention

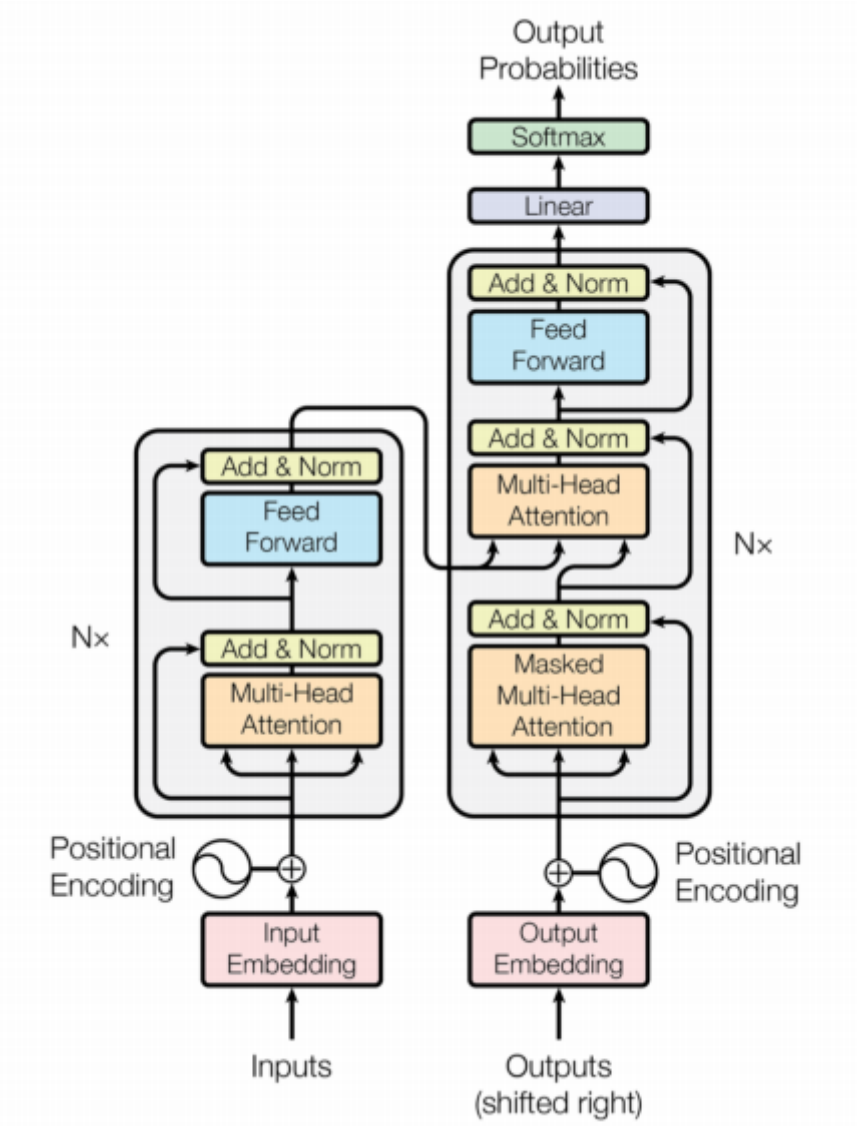
注意力一般分为两种：

1. **自上而下**的有意识的注意力，称为聚焦式注意力（Focus Attention）．聚焦式注意力也常称为选择性注意力（Selective Attention）．聚焦式注意力是指有预定目的、依赖任务的，主动有意识地聚焦于某一对象的注意力．
2. **自下而上**的无意识的注意力，称为基于显著性的注意力（Saliency Based Attention）．基于显著性的注意力是由外界刺激驱动的注意，不需要主动干预，也和任务无关．如果一个对象的刺激信息不同于其周围信息，一种无意识的“赢者通吃”（Winner-Take-All，比如最大汇聚）或者门控（Gating）机制就可以把注意力转向这个对象．不管这些注意力是有意还是无意，大部分的人脑活动都需要依赖注意力，比如记忆信息、阅读或思考等．

可见，**Transformer 使用的是第一种注意力机制，LSTM 使用的就是第二种注意力机制。**

Transformer

网络结构



左边的 encoder 部分和右边的 decoder 部分均堆叠了 6 个模块，即 $N = 6$.

对于 encoder 部分，一个模块的主要组件是 2 个 sub-layers：Multi-Head Attention 和 Feed Forward。接着，We employ a residual connection around each of the two sub-layers, followed by layer normalization，也就是图中的 `Add & Norm` 组件. 此外，We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized，也就是说，**对 sub-layer 的输出，先进行 dropout，再残差连接，最后进行层归一化**:

```
LayerNorm(x+Dropout(Sublayer(x))) # Sublayer 是 Multi-Head Attention 或 Feed Forward
```

代码中的实现是

```
x+Dropout(Sublayer(LayerNorm(x)))
```

其顺序也是正确的，代码中的 `x` 实际与前面的 `x` 含义不同，指的是残差连接的输出。Transformer 中 dropout 的概率设置为 `0.1`.

另外，对于层归一化 layer normalization，并不是对这一层的值减去均值，除以标准差就可以了，**还有缩放和平移两个待学习的参数向量 (注意， Layer Normalization applies per-element scale and bias.)**，也就是代码中的 `self.a_2, self.b_2`.

> 关于 layer normalization，可参考 LayerNorm — PyTorch 1.12 documentation，解释得很清楚，或者 Layer Normalization (labml.ai)。一般我们做 layer normalization，对 NLP 数据 `[batch, sentence_length, embedding_dim]`，会 Normalize over `embedding_dim`，也就是对第 i 个样本的第 j 个 token 向量 (长为 `embedding_dim`) 进行 normalization，不同 i, j 的向量 (共 `batch X sentence_length` 个) normalization 时，计算均值和方差时相互独立，在各自向量内部计算，但缩放和平移的参数向量则是共享的，也就是说待学习的参数共有 `2 X embedding_dim` 个，而不是 `2 X batch X sentence_length X embedding_dim` 个；对 Image `[N, C, H, W]`，会 Normalize over the last three dimensions (i.e. the channel and spatial dimensions)。

下面先介绍 encoder 部分，再介绍 decoder 部分，最后介绍 embedding + positional encoding 部分，背景是 the Multi30k German-English Translation task 这样一个**序列到序列**任务。

Encoder

和 RNN 不同，encoder 部分输入序列不是依次读入处理的，encoder 一下子可以看到所有输入。source 和 target language token 的 embedding dim 相同 (一个重要原因是Transformer让 source 和 target language共用一个 vocabulary)，记为 d_{model}，论文中取 $d_{model} = 512$。

首先介绍 encoder 和 decoder 中都会用到的多头 (h 头) 注意力：

$$
\begin{aligned}
&\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \cdots, \mathrm{head}_\mathrm{h}) W^O \\
&\text{where } \mathrm{head}_\mathrm{i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V) \\
&\qquad\qquad = \mathrm{softmax}\left(\frac{(QW_i^Q)(KW_i^K)^\mathrm{T}}{\sqrt{d_k}}\right)(VW_i^V)
\end{aligned}
$$

$\mathrm{MultiHead}(Q, K, V)$ 的 Q, K, V 由用户提供是已知的，它们分别经过 W^Q, W^K, W^V 变换后得到注意力机制的查询 (query)、键 (key)、值 (value) 向量组成的矩阵，d_k 表示 query 向量的维度 (query 向量与 key 向量的维数相同，毕竟它们要做点积的)。 多头注意力经过拼接后还要经过 W^O 的变换，因此，多头注意力中需要学习的参数有 W_i^Q, W_i^K, W_i^V $(i = 1, \cdots, h)$ 和 W^O. 而注意力函数 $\mathrm{Attention}(Q, K, V)$，它是没有可学习的参数的，用户提供 Q, K, V 后它返回相应的计算结果。可以看到，`MultiHead` 和 `Attention` 的 Q, K, V 含义并不完全相同。

Transformer **保持每个 sub-layer (Multi-Head Attention 和 Feed Forward) 输入和输出的shape相同**，即若输入 $X : T \times d_{model}$ 由维度为 d_{model} 的 T 个序列组成，则输出的也是 T 个维度为 d_{model} 的向量。

encoder 中底层模块的输出是上层模块的输入，以最底层模块为例，它接受样本 X，此时 $Q = K = V = X : T \times d_{model}$，$T$ 为该样本序列的长度，也就是所含token的个数，d_{model} 为 embedding 的维度，注意 X 表示一个样本而不是一个 batch，则

$$
\begin{array}{ll}
W_i^Q : d_{model} \times d_k & QW_i^Q : T \times d_k \\
W_i^K : d_{model} \times d_k & KW_i^K : T \times d_k \\
W_i^V : d_{model} \times d_v & VW_i^V : T \times d_v \\
\mathrm{head}_\mathrm{i} : T \times d_v & \mathrm{Concat}(\mathrm{head}_1, \cdots, \mathrm{head}_\mathrm{h}) : T \times hd_v \\
W^O : hd_v \times d_{model} & \mathrm{MultiHead} : T \times d_{model}
\end{array}
$$

Transformer 取 $h=8$ 头注意力，并令 $d_k=d_v=d_{model}/h=64$。`softmax` 的输入输出均为 $T\times T$ 的矩阵，softmax 对每一行进行：每一行给出了一个注意力分布，第 t 行给出了第 t 个位置的输出向量是被如何被加权得到的，参与加权的向量共有 T 个，维度为 d_v，即 $VW_i^V:T\times d_v$。拼接 `Concat` 按列进行，即相同位置的行向量拼接成一个更长的行向量，拼接后的矩阵的行数不变。`MultiHead` 层在拼接完后还乘以了 W^O 进行线性映射，注意，这一步线性映射不是我们所说的 Fully connected sub-layer(即文章中提到的 Position-wise Feed-Forward Network)。最终，$\mathrm{MultiHead}$ 层的输出的shape保持为 $T\times d_{model}$。

The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately. 经过多头自注意力sub-layer后，接下来的一个 sub-layer 就是 a fully connected feed-forward network, which is applied to each position (指 $t=1,\cdots,T$) separately and identically. This consists of two linear transformations with a ReLU activation in between. The dimensionality of input and output is $d_{model}=512$, and the inner-layer has dimensionality $d_{ff}=2048$.

也就是说 Fully connected sub-layer 接受一个 d_{model} 维向量，输出一个 d_{model} 维向量，只含有一个隐层，隐层神经元个数为 $d_{ff}=2048$；对于 `MultiHead` 层输出的 T 个 d_{model} 向量 (不妨记第 t 个 position 的行向量为 x_t)，它们都经过同样的前馈神经网络处理：

$$\mathrm{FFN}(x_t)=\max(0,x_tW_1+b_1)W_2+b_2$$

$W_1:d_{model}\times d_{ff},W_2:d_{ff}\times d_{model},b_1,b_2$ 为这些向量所共享的网络参数。**事实上，根据代码实现，relu 后还加了一层 dropout**：

```
self.w_2(self.dropout(self.w_1(x).relu()))
```

可以看到:

- 和 RNN 类似，Transformer 参数的多少与处理的序列长度 T 无关，它可以处理任意长度的序列；
- 和 CNN 类似，Transformer 又可以一次性看到全部的 input；
- 关于并行计算：各个 position 的信息经 attention 层处理时是相互依赖的，并行不了；但是在前馈全连接 sub-layer 的计算是相互独立的，可以并行。Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Decoder

decoder 模块在结构上在自注意力 sub-layer 和前馈全连接 sub-layer 中间多加了一层由 encoder 和 decoder 共同参与的多头注意力 $\mathrm{MultiHead}(Q,K,V)$，其中 Q 由 decoder 提供，K,V 则由最顶层的 encoder 的 output 提供，代码中的实现是：

```
self.src_attn(x, m, m, src_mask)
# x 由 decoder 提供
# m 是 encoder 最终的输出，代码中也记为 memory
```

Transformer 的结构图中，encoder 到 decoder 有两个箭头，这两个箭头是说多头注意力的 key、value 矩阵基于 encoder 的 output 矩阵产生，encoder 只提供一个 output 矩阵，从 output 到 key、value 还需要分别乘以参数矩阵 W^K,W^Q，进行线性变换，而这两类矩阵的学习是在 decoder 中进行的，不要误以为 encoder 直接提供了 key 和 value 矩阵。

---

decoder 和 LSTM 类似，generates one word at a time, from left to right. 不同的是，基于注意力机制，decoder 可以看到这个时刻及之前的所有 decoder 的输入 (也就是架构图decoder部分最下面的 `Outputs (shifted right)`)，和 encoder 最终的输出。encoder 最终的输出如前所述是一个 $T \times d_{model}$ 的矩阵，在解码过程中不再变化，而 decoder 的输入:

- 预测时，decoder 某一时刻的输入是**之前所有时刻** decoder 所产生的输出。因此随着时间推移，decoder 接受的输入序列的长度是不断变长的；作为对比，LSTM decoder 的输入**只是上一时刻** decoder 的输出。此外，和 LSTM 等类似，除了 greedy search，还可以使用 beam search 扩展预测时的搜索空间 (论文中 used beam search with a beam size of 4 and length penalty $\alpha = 0.6$，length penalty 是作用于句子长度的一个正则化超参数)；
- 训练时，为了加速收敛，和 LSTM 类似，可以将 target 序列直接作为 decoder 的输入，比如长为 N 的 target 序列 $\text{token}_1, \text{token}_2, \cdots, \text{token}_N$，则为了预测 token_n，decoder 接受的输入应该是 $\langle s \rangle, \text{token}_1, \text{token}_2, \cdots, \text{token}_{n-1}$，其中 $\langle s \rangle$ 为标识符，是预测或训练时 decoder 接收到的第一个输入，这也是 `shifted right` 说法的原因。

---

decoder 还有一个非常重要的细节，论文中一笔带过：

> We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.
>
> ---
>
> Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections.

decoder 中的自注意力 sub-layer 是 **Masked** Multi-Head Attention. 比 encoder 的自注意力多了个 mask。最开始误以为是针对训练，在代码实现时先提供全部的 target 序列 $\langle s \rangle, \text{token}_1, \text{token}_2, \cdots, \text{token}_{n-1}, \text{token}_n, \cdots, \text{token}_N, \langle /s \rangle$，再 mask 掉当前时刻不需要的后半部分序列 $\text{token}_n, \cdots, \text{token}_N, \langle /s \rangle$。事实上，不论是训练还是预测，decoder 的自注意力均会用到 mask 机制。`Attention` **的 mask 机制总结而言就是，给定 Q, K, V，最终输出的第 i 个注意力向量，只基于查询向量 $Q[i, :]$ 和 $K[: i, :], V[: i, :]$ 计算得到，即 $\text{Attention}(Q[i, :], K[: i, :], V[: i, :])$，而 $K[i+1 :, :], V[i+1 :, :]$ 并不会参与计算。可以看到，该层第 i 个position的输出 (也是下一层第 i 个position的输入) 只用到前 i 个position的输入信息，信息的利用是单向的，只用到左边position的信息，而encoder部分左右 position 的信息都能用到，信息的利用是双向的，这也是 BERT (Bidirectional Encoder Representations from Transformers) 名称的由来，因为 BERT 只基于 Transformer 的 encoder 部分。**

下面详细考察 decoder 模块的两个注意力 sub-layer：

记 encoder 最终的输出为矩阵 $M: T \times d_{model}$，decoder 在 N 时刻的输入 $X: N \times d_{model}$，注意，这里 N 表示当前 decoder 接受的输入序列的长度，其数值随时间不断增大，而不是像上面那样表示训练时 target 序列的总长。

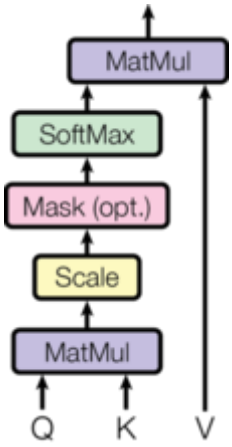
1. 首先是 `Masked Multi-Head Attention` sub-layer。

这是一个多头自注意层，$\mathrm{MultiHead}(Q,K,V)$ 的 Q,K,V 均是 X，第 i 头注意力：

$$
\begin{aligned}
W_i^Q &: d_{model} \times d_k \quad XW_i^Q : N \times d_k \\
W_i^K &: d_{model} \times d_k \quad XW_i^K : N \times d_k \\
W_i^V &: d_{model} \times d_k \quad XW_i^V : N \times d_k
\end{aligned}
$$

根据矩阵乘法，XW 的第 n 行由 $X[n,:]W$ 给出，因此，多头注意力对 X 进行投影，各 position 的信息，也就是 X 的行向量间并不会交互，否则后面 mask 的让信息是单向交互的机制就毫无意义了。不妨记 XW_i^Q, XW_i^K, XW_i^V 为 Q_i, K_i, W_i $(i=1,\cdots,8)$，下面考察 $\mathrm{Attention}(Q_i,K_i,V_i)$ 的计算。

`Attention` 的完整实现如下图所示：



其中，`Scale` 表示除以 $\sqrt{d_k}$；`Mask (opt.)` 就是 mask 机制，只在 decoder 模块的 `Masked Multi-Head Attention` 中起作用，因此是可选项。具体而言：

$$
\begin{aligned}
\mathrm{head_i} &= \mathrm{Attention}(Q_i,K_i,V_i) \\
&= \mathrm{softmax}\left(\mathrm{mask}\left(\frac{Q_i(K_i)^{\mathrm{T}}}{\sqrt{d_k}}\right)\right)V_i
\end{aligned}
$$

其中，

$$
\begin{aligned}
S_i \triangleq \frac{Q_i(K_i)^{\mathrm{T}}}{\sqrt{d_k}} &: N \times N \\
V_i &: N \times d_k \\
\mathrm{head_i} &: N \times d_k
\end{aligned}
$$

为了简便起见，下面忽略下标 i。**矩阵 S 的第 n 行 $S[n,:] = Q[n,:]K^T$ 给出了 `softmax` 的 score，经过 `softmax` 处理后的概率分布作为权重以对 V 的行向量进行加权平均，加权平均的结果即为第 n 个查询向量或者说 position 的输出。为了实现信息的单向传递，也就是对于第 n 个查询向量 $Q[n,:]$，对应的 key, value 在计算时只有 $K[:n,:], V[:n,:]$。为此，在代码实现时，只需将 S 矩阵的主对线之上的元素全部置为 $-\infty$ (代码中是 $-1e^9$)，而主对角线及以下的元素保持不变即可，这样经过 `softmax` 后 $-\infty$ 处的概率就为 0，$K[n+1:,:], V[n+1:,:]$ 对输出结果不再有影响，这就是 `mask` 的过程。**

多头注意力按 position 经过 concat 并经过 W^O 变换后输出的矩阵大小为 $N \times d_{model}$，同样地，这两个操作也不涉及不同 position 向量的交互。接下来经过 `Add & Norm` 模块，输出大小不变，不同 position 的向量间也并不交互 (这进一步说明 layer normalization 是对每个行向量分别进行，而不是对整个矩阵进行，否则不同 position 的信息就会有交互，mask 机制也就没有意义了)。为了方便表述，我们仍然记输出为 $X: N \times d_{model}$，接下来进入与 encoder 一起进行的多头注意力 sub-layer.

2. `Multi-Head Attention` sub-layer

此时，$\mathrm{MultiHead}(Q,K,V)$ 的查询向量 Q 由 decoder 提供，为 $X:N\times d_{model}$，K,V 则由 encoder 提供，为 $M:T\times d_{model}$，即 $\mathrm{MultiHead}(X,M,M)$。也就是说，对任意 position 的查询向量，它可以看到 encoder 端输出的所有信息。类似地，第 i 头注意力：

$$
\begin{aligned}
W_i^Q:d_{model}\times d_k \quad XW_i^Q:N\times d_k\\
W_i^K:d_{model}\times d_k \quad MW_i^K:T\times d_k\\
W_i^V:d_{model}\times d_k \quad MW_i^V:T\times d_k
\end{aligned}
$$

此时，

$$
\begin{aligned}
S_i\triangleq\frac{Q_i(K_i)^{\mathrm{T}}}{\sqrt{d_k}}:N\times T\\
V_i:T\times d_k\\
\mathrm{head_i}:N\times d_k
\end{aligned}
$$

接下来的 concat、W^O 线性变换等步骤不再赘述。可以看到，因为 K,V 已知，且不是自注意力，因此与 $Q=X$ 无关，而是事先由 encoder 给定 (即 M)，则此 sub-layer **各 position 查询向量对应输出的计算相互独立，可以并行进行。**sub-layer 最终输出的大小为 $N\times d_{model}$。

经过 6 个 decoder stacks，输出的是一个 $N\times d_{model}$ 的矩阵，不妨仍然记为 X，其中第 n 行向量能看到前面 $n-1$ 行向量的信息，而不能看到 $n+1\sim N$ 行的信息，但它们都能看到 encoder 端输出的信息。

decoder 最终要输出的是第 N 个时间步词表中各个词的概率，这通过网络结构图中最顶部的 `Linear` + `Softmax` 两个模块实现 (代码中记为 `generator`)，它们的输入为 `X[N,:]`，是一个长为 d_{model} 的向量，即只用了最后一个 position 的信息，这保证了网络宽度是固定的而不是变长的。代码中对应的是

```
prob = model.generator(out[:, -1])
```

Embedding & Positional Encoding

Transformer 和 BERT 用的分词算法分别是 **byte-pair encoding (BPE)** 和 **WordPiece**，它们都属于 subword 算法。subword 算法不再以空格进行分词，分词的颗粒度更细，比如 `low lower` 分词的结果可能是 `low` 和 `er` 两个 token 而不是 `low` 和 `lower` 两个 token。这种思想就类似于 `fastText` 进行 embedding 时通过使用单词的 n-gram 来考虑word的形态信息 (the morphology of words)，也就是 Subword Information。

BPE 从单个字符开始，自顶向上地合并出现频率最高的相邻字节对，直到达到预设的词表大小或相邻字节对出现的频率最大为 1。构建过程有点类似于 Huffman Tree 的构建，具体可以参考 [BPE算法详解 - mathor (wmathor.com)](https://wmathor.com/index.php/archives/1696/)。WordPiece 生成词表的方式和 BPE 非常相近，都是用合并 token 的方式来生成新的 token，最大的区别在于选择合并哪两个 token：BPE 选择频率最高的相邻字节对进行合并，而 WordPiece 的做法是选择合并前后最能提升句子概率的相邻字节对，可见 [NLP中的subword算法及实现 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/191648692)。

还有一个重要细节是，对 the Multi30k German-English 这样一个翻译任务，**source language 和 target language 共用一个词表**！这确实是可以实现的，毕竟德语也使用的是拉丁字母。这也是 source 和 target language token 的 embedding dim 相同，均取 d_{model} 的原因。

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. 也就是说，只有 embedding 是不够的，我们还希望能让模型看到各 token 的位置信息。为此，我们在 embedding 上加上一个同样长为 d_{model} 的 positional encoding 后才作为输入提供给模型 (BERT 还会再加这个token属于句子对中哪一个句子的信息)。**positional encoding 可以像 embedding 一样通过学习得到，也可以直接事先指定，二者效果差不多，因此 Transformer 选择了事先给定的方案，其 positional encoding 基于三角函数生成，函数形式见原论文。**

另一个细节是：In addition, we apply **dropout** to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$. 也就是说，embedding + positional encoding 后会接一层 dropout 再提供给 encoder 和 decoder。

还有一个非常重要的细节，论文里没有提，但官方给的notation里面有：

> Shared Embeddings: When using BPE with shared vocabulary we can share the same weight vectors between the source / target / generator. See the Using the Output Embedding to Improve Language Models (arxiv.org) for details.

看了下它给的参考文献，意思是说 source language 和 target language 会共享一个词表 embedding，即对应一个矩阵 $W_i : C \times d_{model}$ （C 表示词表大小）。此外，在 decoder 最顶层 `Linear` + `Softmax` 输出概率时，`Linear` 模块将 d_{model} 维向量映射到 C 维向量所用的输出矩阵 $W_o : d_{model} \times C$ 直接就用 W_i 的转置，即 $W_o = W_i^{\mathrm{T}}$。参考文献中提到的类似例子还有强制令 word2vec 的输出矩阵与输入矩阵的转置相等。

这样，encoder input embedding、decoder input embedding 以及 decoder output embedding (也就是 `Softmax` 前的 `Linear` 层) 三者的权重共享 (weight tying)，文中称之为 three-way weight tying (TWWT)。

其他

- label smothing：假设训练数据集中有一些样本的标签是被错误标注的，那么最小化这些样本上的损失函数会导致过拟合，一种改善的正则化方法是标签平滑（Label Smoothing），即在输出标签中添加噪声来避免模型过拟合。标签平滑可以避免模型的输出过拟合到硬目标上，并且通常不会损害其分类能力。
- model average：For the base models, we used a single model obtained by averaging the last 5 checkpoints, which were written at 10-minute intervals. For the big models, we averaged the last 20 checkpoints.
- 优化算法：Adam + warm up steps.
- 论文中 perplexity (PPL) 和 BLEU 是评估序列到序列模型好坏的两个指标.

> 困惑度（Perplexity）是信息论中的一个概念，可以用来衡量一个分布的不确定性，也可以用来衡量两个分布之间差异，困惑度越低则两个分布越接近。因此，模型分布与样本经验分布之间的困惑度越低，模型越好。
>
> BLEU（BiLingual Evaluation Understudy）算法是一种衡量模型生成序列和参考序列之间的 N 元词组（N-Gram）重合度的算法，最早用来评价机器翻译模型的质量，目前也广泛应用在各种序列生成任务中。BLEU 越大表明生成的质量越好，但是 BLEU 算法只计算精度，而不关心召回率。
>
> --《NNDL》15.4 评价方法

BERT

BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) :

- BERT 实际上只用了 Transformer 的 encoder 模块，严格来说它根本不是一个完整的 Transformer 网络；
- 此外，Bidirectional (双向) 也并不是类似于从 LSTM 到双向 LSTM 那样的改进，它是 Transformer encoder 本身就有的性质，即任意 position 都可以看到两侧所有的信息 (Transformer decoder 则是单向的，只能看到左侧的信息)；
- BERT 在网络结构上并没有什么创新，它的创新点在于构建了两个预训练任务，而 BERT 本身则作为一个特征抽取器，在大规模数据上进行预训练后，再针对不同的下游任务，在 BERT 输出的特征上搭建相应的模块 (比如，分类模块) 进行微调，这一套做法与计算机视觉中的做法相似，效果也非常好。

论文中介绍了两个模型：**BERT BASE** 和 **BERT LARGE**，就是模型大小不同而已。

参考文献

1. Attention Is All You Need (arxiv.org)：Transformer 的原始论文；
2. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (arxiv.org)：BERT 的原始论文；
3. The Annotated Transformer (harvard.edu)：Transformer 的官方notation，含有实例代码；
4. The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io)：比较好的讲 Transformer 的 blog；
5. 《NNDL》邱锡鹏.