

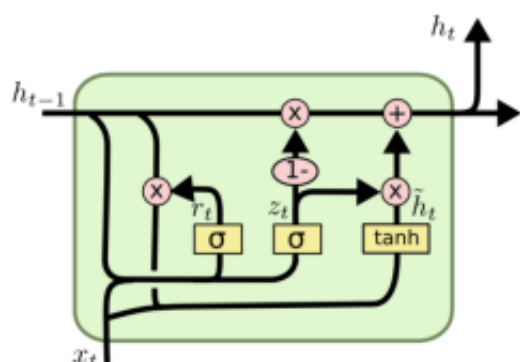
NLP

Word2Vec中skip-gram是什么,Negative Sampling怎么做

参考回答:

Word2Vec通过学习文本然后用词向量的方式表征词的语义信息,然后使得语义相似的单词在嵌入式空间中的距离很近。而在Word2Vec模型中有Skip-Gram和CBOW两种模式,Skip-Gram是给定输入单词来预测上下文,而CBOW与之相反,是给定上下文来预测输入单词。Negative Sampling是对于给定的词,并生成其负采样词集合的一种策略,已知有一个词,这个词可以看做一个正例,而它的上下文词集可以看做是负例,但是负例的样本太多,而在语料库中,各个词出现的频率是不一样的,所以在采样时可以要求高频词选中的概率较大,低频词选中的概率较小,这样就转化为一个带权采样问题,大幅度提高了模型的性能。

Lstm和Gru



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Gru由重置门和更新门组成,其输入为前一时刻隐藏层的输出和当前的输入,输出为下一时刻隐藏层的信息。重置门用来计算候选隐藏层的输出,其作用是控制保留多少前一时刻的隐藏层。更新门的作用是控制加入多少候选隐藏层的输出信息,从而得到当前隐藏层的输出。

Lstm由输入门,遗忘门,输出门和一个cell组成。第一步是决定从cell状态中丢弃什么信息,然后在决定有多少新的信息进入到cell状态中,最终基于目前的cell状态决定输出什么样的信息。

遗忘门:

$$f = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

输入门:

$$i = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

输出门:

$$o = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

LSTM与GRU区别：1) GRU和LSTM的性能在很多任务上不分伯仲。2) GRU 参数更少因此更容易收敛，但是数据集很大的情况下，LSTM表达性能更好。3) 从结构上来说，GRU只有两个门（update和reset），LSTM有三个门（forget, input, output），GRU直接将hidden state 传给下一个单元，而LSTM则用memory cell 把hidden state 包装起来。

RNN梯度消失问题,为什么LSTM和GRU可以解决此问题

RNN由于网络较深,后面层的输出误差很难影响到前面层的计算,RNN的某一单元主要受它附近单元的影响。

而LSTM因为可以通过阀门记忆一些长期的信息,相应的也就保留了更多的梯度。而GRU也可通过重置和更新两个阀门保留长期的记忆,也相对解决了梯度消失的问题。

梯度裁剪 (Clipping Gradient)

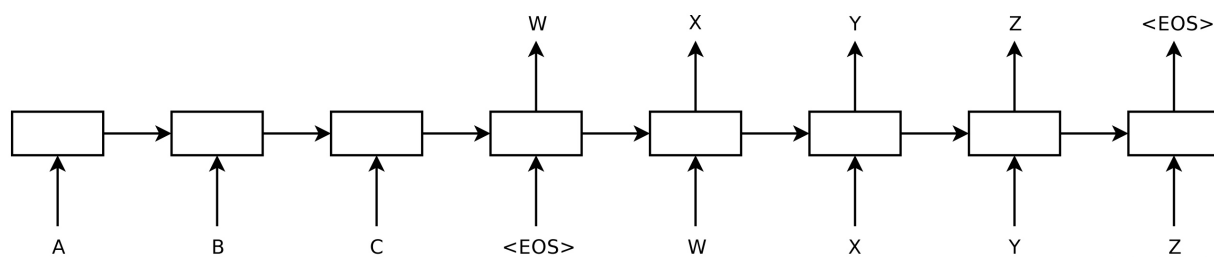
既然在BP过程中会产生梯度消失（就是偏导无限接近0，导致长时记忆无法更新），那么最简单粗暴的方法，设定阈值，当梯度小于阈值时，更新的梯度为阈值。

什么是seq2seq model

参考回答：

Seq2seq属于encoder-decoder结构的一种,利用两个RNN,一个作为encoder一个作为decoder。Encoder负责将输入序列压缩成指定长度的向量,这个向量可以看作这段序列的语义,而decoder负责根据语义向量生成指定的序列。

由于其灵活性，这个框架现在是自然语言生成任务的首选框架，其中不同的模型承担了编码器和解码器的角色。重要的是，解码器模型不仅可以解码一个序列，而且可以解码任意表征。例如，可以基于图像生成标题（Vinyals等,2015）（如下图所示）、基于表生成文本（Lebret等,2016）和基于应用程序中源代码更改描述（Loyola等,2017）。



attention机制

参考回答：

Attention简单理解就是权重分配，。以seq2seq中的attention公式作为讲解。就是对输入的每个词分配一个权重，权重的计算方式为与解码端的隐含层时刻作比较，得到的权重的意义就是权重越大，该词越重要。最终加权求和。sequence-to-sequence模型的主要瓶颈是需要将源序列的全部内容压缩为一个固定大小的向量decode hidden。注意力机制通过允许解码器回头查看源序列隐藏状态来缓解

这一问题，然后将其加权平均作为额外输入context vector提供给解码器。

$$u_t^i = v^T \tanh(W_1 h^i + W_2 H^t)$$

$$\alpha_t^i = \text{softmax}(u_t^i)$$

$$c^t = \sum_i \alpha_t^i h^i$$

RNN/CNN/Tranformer

RNN相关算法只能从左向右依次计算或者从右向左依次计算，这种机制带来了两个问题：时间片的计算依赖时刻的计算结果，这样限制了模型的并行能力；（训练时间很长）顺序计算的过程中信息会丢失，尽管LSTM等门机制的结构一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象,LSTM依旧无能为力。

CNN是层级结构，与循环网络建模的链结构相比，层次结构提供了一种较短的路径来捕获词之间远程的依赖关系，因此也可以更好地捕捉更复杂的关系。通过卷积的叠加可以精确地控制上下文的长度，因为卷积之间的叠加可以通过公式直接计算出感受野是多少，从而知道上下文的长度，RNN虽然理论上有长时记忆的功能，但是在实际的训练过程中，间隔较远的时候，很难学到这种词与词之间的联系。

- RNN

- O：位置/时序

- X：并行计算；内部关系/结构；长期依赖

- CNN

- O：并行计算；局部特征；

- X：位置/时序；内部关系/结构；长期依赖；

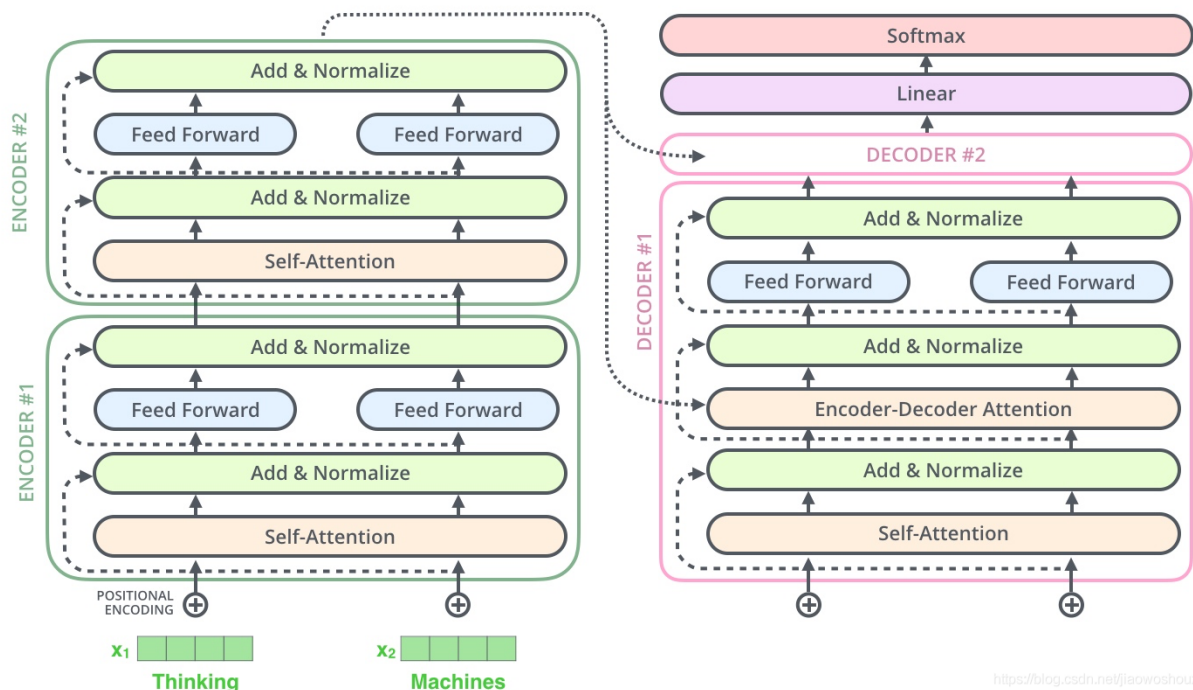
- Transformer

- O：并行计算；内部关系/结构；长期依赖

- X：位置/时序；局部特征

Transformer

对于encoder，包含两层，一个self-attention层和一个前馈神经网络，self-attention能帮助当前节点不仅仅只关注当前的词，从而能获取到上下文的语义。decoder也包含encoder提到的两层网络，但是在这两层中间还有一层attention层，帮助当前节点获取到当前需要关注的重点内容。



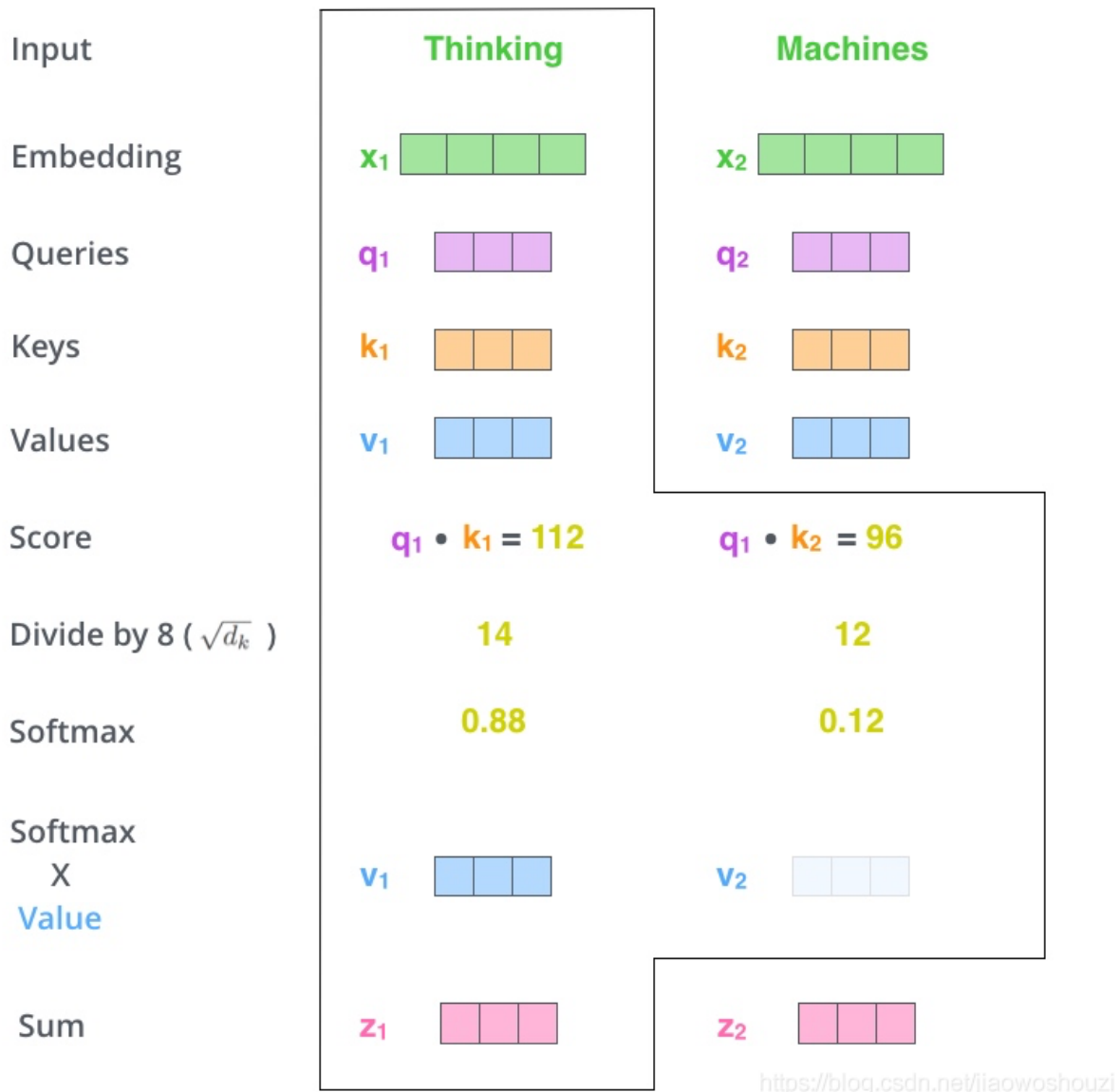
<https://blog.csdn.net/jieowoshouzi>

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向量与一个矩阵相乘得到的结果，这个矩阵是随机初始化的，维度为（64，512）注意第二个维度需要和embedding的维度一样，其值在BP的过程中会一直进行更新，得到的这三个向量的维度是64低于embedding维度的。

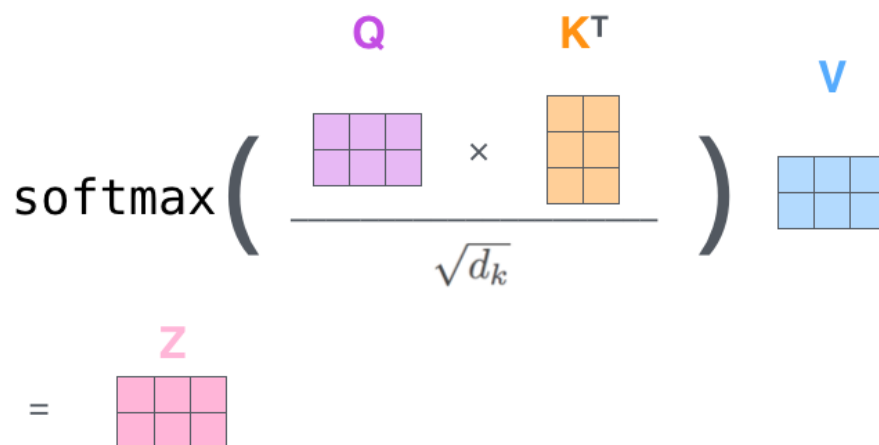
计算self-attention的分数值，该分数值决定了当我们在某个位置encode一个词时，对输入句子的其他部分的关注程度。这个分数值的计算方法是Query与Key做点乘，以下图为例，首先我们需要针对Thinking这个词，计算出其他词对于该词的一个分数值，首先是针对于自己本身即 $q_1 \cdot k_1$ ，然后是针对于第二个词即 $q_1 \cdot k_2$ 。

接下来，把点乘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-attention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。

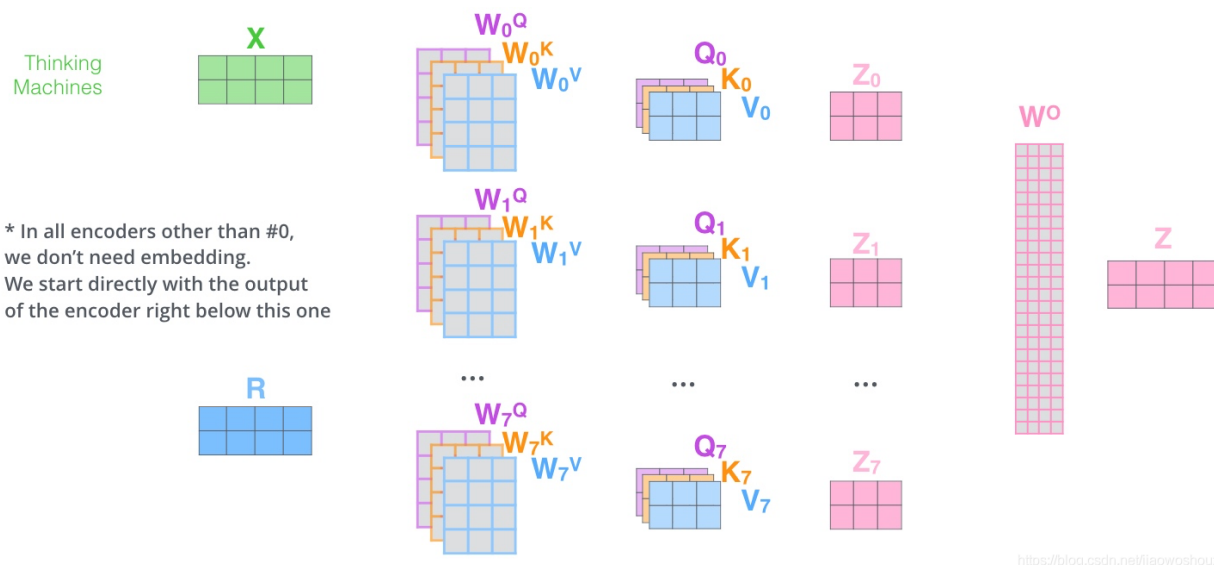
$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) \times V = Z$$


<https://blog.csdn.net/jiaowoshouzi>

Multi-Headed Attention

这篇论文更厉害的地方是给self-attention加入了另外一个机制，被称为“multi-headed” attention，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，transformer是使用了8组，所以最后得到的结果是8个矩阵。这给我们留下了一个小的挑战，前馈神经网络没法输入8个矩阵呀，这该怎么办呢？所以我们需要一种方式，把8个矩阵降为1个，首先，我们把8个矩阵连在一起，这样会得到一个大的矩阵，再随机初始化一个矩阵和这个组合好的矩阵相乘，最后得到一个最终的矩阵。

- 1) This is our input sentence*
- 2) We embed each word*
- 3) Split into 8 heads. We multiply X or R with weight matrices
- 4) Calculate attention using the resulting $Q/K/V$ matrices
- 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



Positional Encoding

到目前为止，transformer模型中还缺少一种解释输入序列中单词顺序的方法。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，维度和embedding的维度一样，这个向量采用了一种很独特的方法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

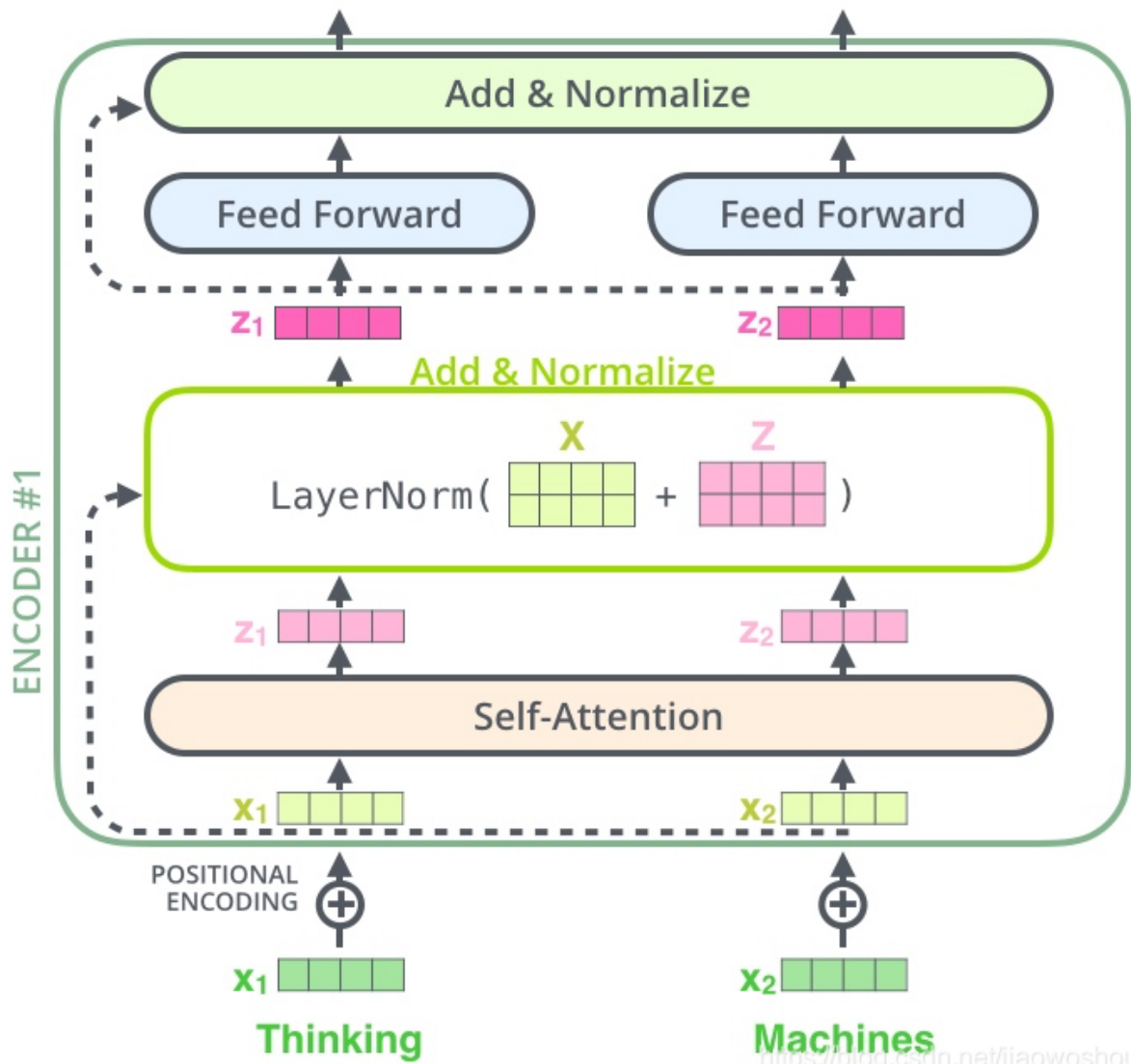
$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$$

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

其中pos是指当前词在句子中的位置，i是指向量中每个值的index，可以看出，在偶数位置，使用正弦编码，在奇数位置，使用余弦编码。最后把这个Positional Encoding与embedding的值相加，作为输入送到下一层。

Layer normalization

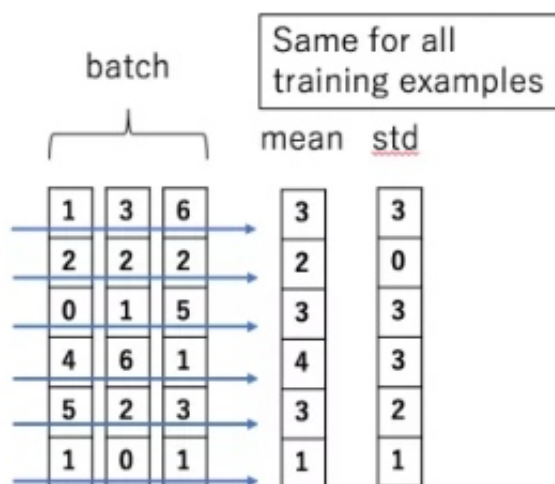
在transformer中，每一个子层（self-attention，FFN）之后都会接一个残差模块，并且有一个Layer normalization。



Normalization有很多种，但是它们都有一个共同的目的，那就是把输入转化成均值为0方差为1的数据来防止梯度消失或者梯度爆炸。

Batch Normalization的具体做法就是对每一小批数据，在batch这个方向上做标准化。

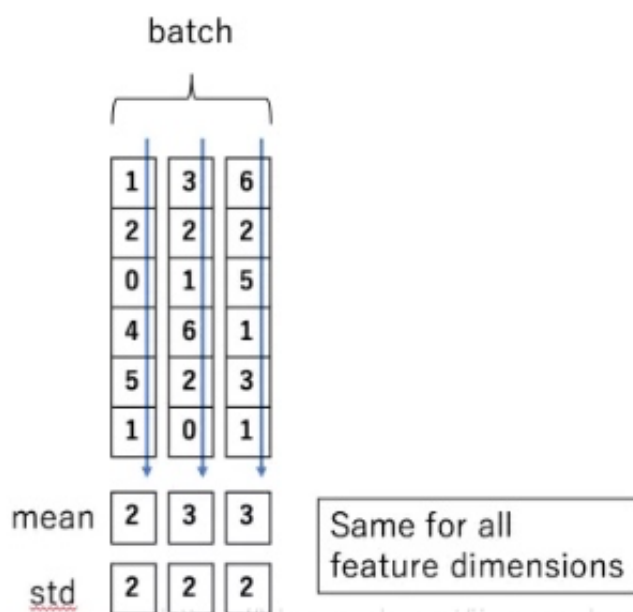
Batch Normalization



<https://blog.csdn.net/jiaowoshouzi>

那么什么是 Layer normalization 呢？它也是标准化数据的一种方式，不过 LN 是在每一个样本上计算均值和方差。

Layer Normalization



<https://blog.csdn.net/jiaowoshouzi>

那么我们为什么还要使用LN呢？因为NLP领域中，LN更为合适。

如果我们将一批文本组成一个batch，那么BN的操作方向是，对每句话的**第一个词**进行操作。但语言文本的复杂性是很高的，任何一个词都有可能放在初始位置，且词序可能并不影响我们对句子的理解。而BN是**针对每个位置**进行缩放，这不符合**NLP的规律**。

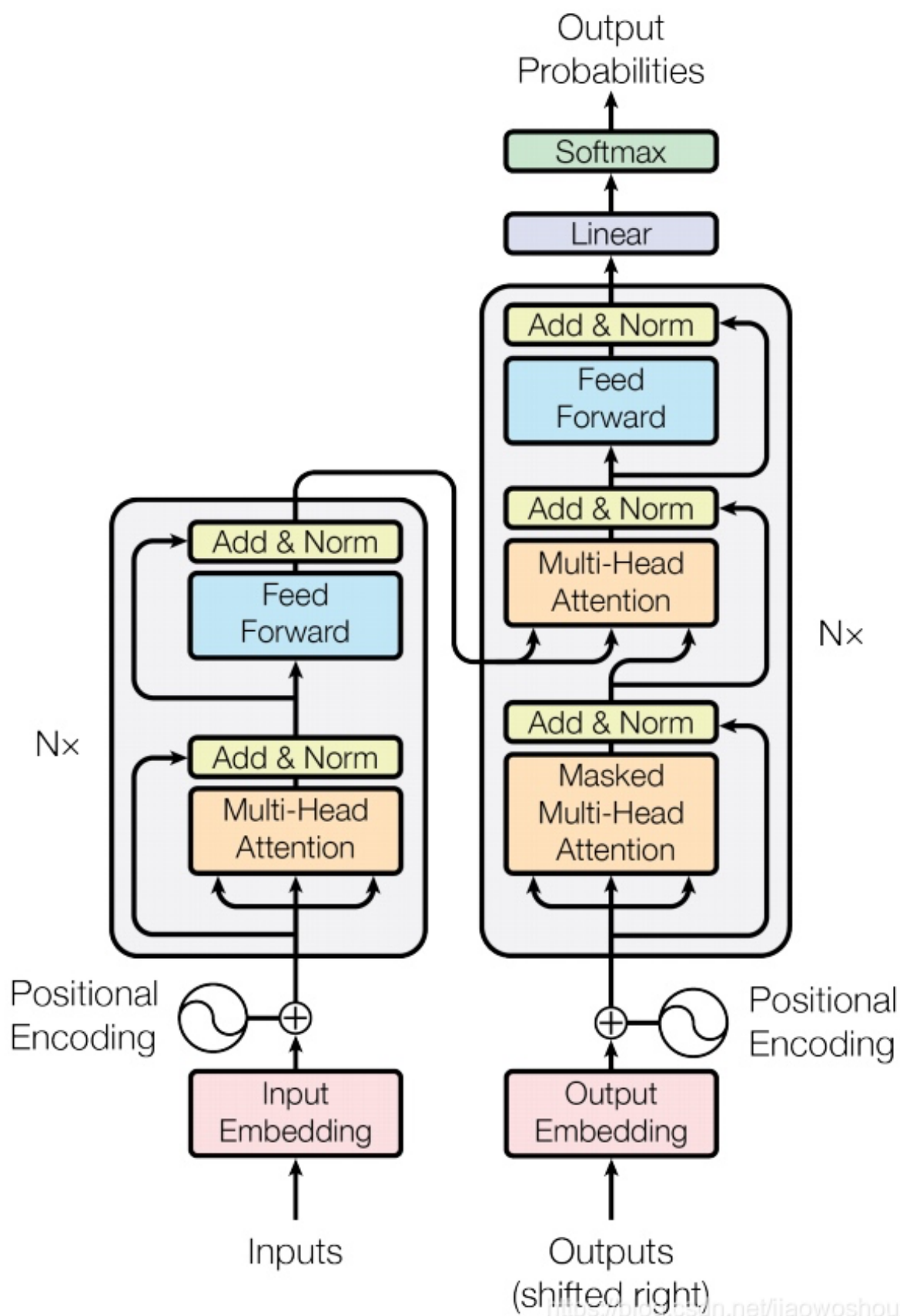
而LN则是针对一句话进行缩放的，且LN一般用在第三维度，如[batchsize, seq_len, dims]中的dims，一般为词向量的维度，或者是RNN的输出维度等等，这一维度各个特征的量纲应该相同。因此也不会遇到上面因为特征的量纲不同而导致的缩放问题。

Encoder-decoder过程

编码器通过处理输入序列启动。然后将顶部编码器的输出转换为一组注意向量k和v。每个解码器将在其“encoder-decoder attention”层中使用这些注意向量，这有助于解码器将注意力集中在输入序列中的适当位置。

完成编码阶段后，我们开始解码阶段。解码阶段的每个步骤从输出序列（本例中为英语翻译句）输出一个元素。以下步骤重复此过程，一直到达表示解码器已完成输出的符号。每一步的输出在下一个时间步被送入底部解码器，解码器像就像我们对编码器输入所做操作那样，我们将位置编码嵌入并添加到这些解码器输入中，以表示每个字的位置。

当decoder层全部执行完毕后，怎么把得到的向量映射为我们需要的词呢，很简单，只需要在结尾再添加一个全连接层和softmax层，假如我们的词典是1w个词，那最终softmax会输入1w个词的概率，概率值最大的对应的词就是我们最终的结果。

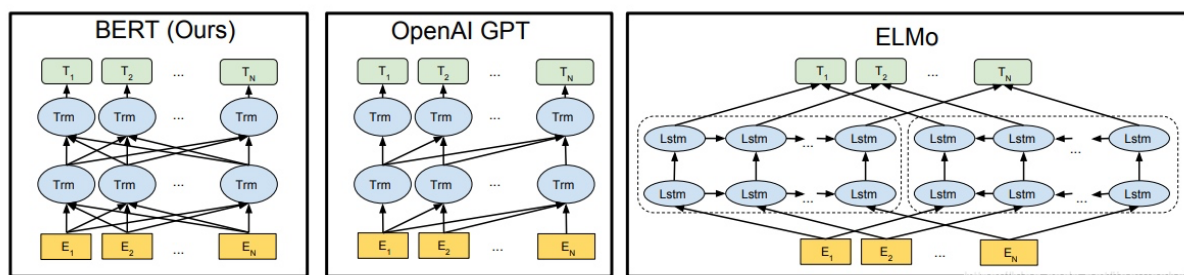


预训练语言模型

首先我们要了解一下什么是预训练模型，举个例子，假设我们有大量的维基百科数据，那么我们可以用这部分巨大的数据来训练一个泛化能力很强的模型，当我们需要在特定场景使用时，例如做文本相似度计算，那么，只需要简单的修改一些输出层，再用我们自己的数据进行一个增量训练，对权重进行一个轻微的调整。

预训练的好处在于在特定场景使用时不需要用大量的语料来进行训练，节约时间效率高效，bert就是这样的一个泛化能力较强的预训练模型。

从创新的角度来看，bert其实并没有过多的结构方面的创新点，其和GPT一样均是采用的transformer的结构，相对于GPT来说，其是双向结构的，而GPT是单向的，如下图所示



elmo：将上下文当作特征，但是无监督的语料和我们真实的语料还是有区别的，不一定的符合我们特定的任务，是一种双向的特征提取。

openai gpt就做了一个改进，也是通过transformer学习出来一个语言模型，不是固定的，通过任务 finetuning,用transformer代替elmo的lstm。openai gpt其实就是缺少了encoder的transformer。当然也没了encoder与decoder之间的attention。

openAI gpt虽然可以进行fine-tuning,但是有些特殊任务与pretraining输入有出入，单个句子与两个句子不一致的情况，很难解决，还有就是decoder只能看到前面的信息。其次bert在多方面的nlp任务变现来看效果都较好，具备较强的泛化能力，对于特定的任务只需要添加一个输出层来进行fine-tuning即可。

bert的内部结构，官网最开始提供了两个版本，L表示的是transformer的层数，H表示输出的维度，A表示mutil-head attention的个数。

$$BERT_{BASE} : L = 12, H = 768, A = 12, TotalParameters = 110M$$

$$BERT_{LARGE} : L = 24, H = 1024, A = 16, TotalParameters = 340M$$

BERT的预训练过程

接下来我们看看BERT的预训练过程，BERT的预训练阶段包括两个任务，一个是Masked Language Model，还有一个是Next Sentence Prediction。

Masked Language Model

MLM可以理解为完形填空，作者会随机mask每一个句子中15%的词，用其上下文来做预测，例如：

my dog is hairy → my dog is [MASK]

此处将hairy进行了mask处理，然后采用非监督学习的方法预测mask位置的词是什么，但是该方法有一个问题，因为是mask15%的词，其数量已经很高了，这样就会导致某些词在fine-tuning阶段从未见过，为了解决这个问题，作者做了如下的处理：

- 80%的时间是采用[mask], my dog is hairy → my dog is [MASK]
- 10%的时间是随机取一个词来代替mask的词, my dog is hairy -> my dog is apple
- 10%的时间保持不变, my dog is hairy -> my dog is hairy

这样做的好处是，BERT并不知道[MASK]替换的是这15%个Token中的哪一个词(注意：这里意思是输入的时候不知道[MASK]替换的是哪一个词，但是输出还是知道要预测哪个词的)，而且任何一个词都有可能是被替换掉的，比如它看到的apple可能是被替换的词。这样强迫模型在编码当前时刻的时候不能太依赖于当前的词，而要考虑它的上下文，甚至对其上下文进行“纠错”。比如上面的例子模型在编码apple是根据上下文my dog is应该把apple(部分)编码成hairy的语义而不是apple的语义。

Next Sentence Prediction

选择一些句子对A与B，其中50%的数据B是A的下一条句子，剩余50%的数据B是语料库中随机选择的，学习其中的相关性，添加这样的预训练的的目的是目前很多NLP的任务比如QA和NLI都需要理解两个句子之间的关系，从而能让预训练的模型更好的适应这样的任务。

损失函数

其中 θ 是 BERT 中 Encoder 部分的参数， θ_1 是 Mask-LM 任务中在 Encoder 上所接的输出层中的参数， θ_2 则是句子预测任务中在 Encoder 接上的分类器参数。

因此，在第一部分的损失函数中，如果被 mask 的词集合为 M ，因为它是一个词典大小 $|V|$ 上的多分类问题，

$$L1(\theta, \theta_1) = - \sum_{i=1}^M y_i * \log p(m = m_i | \theta, \theta_1), m_i \in [1, 2, \dots, |V|]$$

在句子预测任务中，也是一个分类问题的损失函数：

$$L2(\theta, \theta_2) = - \sum_{j=1}^N y_j * \log p(n = n_j | \theta, \theta_2), n_j \in [IsNext, NotNext]$$

激活函数

在神经网络的建模过程中，模型很重要的性质就是非线性，同时为了模型泛化能力，需要加入随机正则，例如dropout(随机置一些输出为0,其实也是一种变相的随机非线性激活)，而随机正则与非线性激活是分开的事情，而其实模型的输入是由非线性激活与随机正则两者共同决定的。

GELUs正是在激活中引入了随机正则的思想，是一种对神经元输入的概率描述，直观上更符合自然的认识，同时实验效果要比Relus与ELUs都要好。

GELUs其实是 dropout、zoneout、Relus的综合，GELUs对于输入乘以一个0,1组成的mask，而该mask的生成则是依概率随机的依赖于输入。假设输入为X, mask为m, 则m服从一个伯努利分布 $\Phi(x)$, $\Phi(x)=P(X \leq x)$, X服从标准正太分布)，这么选择是因为神经元的输入趋向于正太分布，这么设定使得当输入x减小的时候，输入会有一个更高的概率被dropout掉，这样的激活变换就会随机依赖于输入了。

$$GELU(x) = xP(X \leq x) = x\Phi(x)$$

对于假设为标准正太分布的GELU(x), 论文中提供了近似计算的数学公式，如下

$$GELU(x) = 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

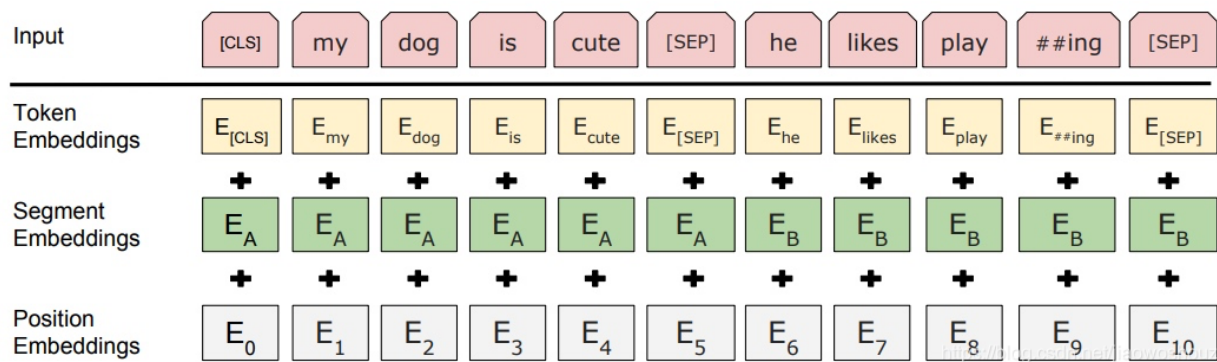
```
def gelu(input_tensor):
    cdf = 0.5 * (1.0 + tf.erf(input_tensor / tf.sqrt(2.0)))
    return input_tensor*cdf
```

输入

bert的输入可以是单一的一个句子或者是句子对，实际的输入值是segment embedding与position embedding相加，具体的操作流程可参考上面的transformer讲解。

BERT的输入词向量是三个向量的加和：（这里的相加是特征交叉）

Token Embedding: WordPiece tokenization subword词向量。Segment Embedding: 表明这个词属于哪个句子（NSP需要两个句子）。Position Embedding: 学习出来的embedding向量。这与Transformer不同，Transformer中是预先设定好的值。



输出

BERT预训练模型的输出结果，无非就是一个或多个向量。下游任务可以通过Finetune（改变预训练模型参数）或者Feature Extraction（不改变预训练模型参数，只是把预训练模型的输出作为特征输入到下游任务）两种方式进行使用。BERT原论文使用了精调方式，但也尝试了特征抽取方式的效果，比如在NER任务上，最好的特征抽取方式只比精调差一点点。但特征抽取方式的好处可以预先计算好所需的向量，存下来就可重复使用，极大提升下游任务模型训练的速度。

BERT优点

- Transformer Encoder因为有Self-attention机制，因此BERT自带双向功能
- 因为双向功能以及多层Self-attention机制的影响，使得BERT必须使用Cloze版的语言模型Masked-LM来完成token级别的预训练
- 为了获取比词更高级别的句子级别的语义表征，BERT加入了Next Sentence Prediction来和Masked-LM一起做联合训练
- 为了适配多任务下的迁移学习，BERT设计了更通用的输入层和输出层
- 微调成本小

BERT缺点

- task1的随机遮挡策略略显粗犷，推荐阅读《Data Nosing As Smoothing In Neural Network Language Models》
- [MASK]标记在实际预测中不会出现，训练时用过多[MASK]影响模型表现;

- 每个batch只有15%的token被预测，所以BERT收敛得比left-to-right模型要慢（它们会预测每个token）
- BERT对硬件资源的消耗巨大（大模型需要16个tpu，历时四天；更大的模型需要64个tpu，历时四天）。

BERT适用场景

- 第一，如果NLP任务偏向在语言本身就包含答案，而不特别依赖文本外的其它特征，往往应用Bert能够极大提升应用效果。典型的任务比如QA和阅读理解，正确答案更偏向对语言的理解程度，理解能力越强，解决得越好，不太依赖语言之外的一些判断因素，所以效果提升就特别明显。反过来说，对于某些任务，除了文本类特征外，其它特征也很关键，比如搜索的用户行为 / 链接分析 / 内容质量等也非常重要，所以Bert的优势可能就不太容易发挥出来。再比如，推荐系统也是类似的道理，Bert可能只能对于文本内容编码有帮助，其它的用户行为类特征，不太容易融入Bert中。
- 第二，Bert特别适合解决句子或者段落的匹配类任务。就是说，Bert特别适合用来解决判断句子关系类问题，这是相对单文本分类任务和序列标注等其它典型NLP任务来说的，很多实验结果表明了这一点。而其中的原因，我觉得很可能主要有两个，一个原因是：很可能是因为Bert在预训练阶段增加了Next Sentence Prediction任务，所以能够在预训练阶段学会一些句间关系的知识，而如果下游任务正好涉及到句间关系判断，就特别吻合Bert本身的长处，于是效果就特别明显。第二个可能的原因是：因为Self Attention机制自带句子A中单词和句子B中任意单词的Attention效果，而这种细粒度的匹配对于句子匹配类的任务尤其重要，所以Transformer的本质特性也决定了它特别适合解决这类任务。
- 第三，Bert的适用场景，与NLP任务对深层语义特征的需求程度有关。**感觉越是需要深层语义特征的任务，越适合利用Bert来解决**；而对有些NLP任务来说，浅层的特征即可解决问题，典型的浅层特征性任务比如分词，POS词性标注，NER，文本分类等任务，这种类型的任务，只需要较短的上下文，以及浅层的非语义的特征，貌似就可以较好地解决问题，所以Bert能够发挥作用的余地就不太大，有点杀鸡用牛刀，有力使不出来的感觉。
- 第四，Bert比较适合解决输入长度不太长的NLP任务，而输入比较长的任务，典型的比如文档级别的任务，Bert解决起来可能就不太好。主要原因在于：Transformer的self attention机制因为要对任意两个单词做attention计算，所以时间复杂度是 n^2 ， n 是输入的长度。如果输入长度比较长，Transformer的训练和推理速度掉得比较厉害，于是，这点约束了Bert的输入长度不能太长。所以对于输入长一些的文档级别的任务，Bert就不容易解决好。结论是：**Bert更适合解决句子级别或者段落级别的NLP任务。**