Bert 思路

准备知识：

深度学习做NLP的方法，基本上都是先将句子分词，然后每个词转化为对应的词向量序列。这样一来，每个句子都对应的是一个矩阵$\boldsymbol{X}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_t)$，其中$\boldsymbol{x}_i$都代表着第$i$个词的词向量（行向量），维度为$d$维，故$\boldsymbol{X}\in\mathbb{R}^{n\times d}$。这样的话，问题就变成了编码这些序列了。

第一个基本的思路是RNN层，RNN的方案很简单，递归式进行：

$$\boldsymbol{y}_t=f(\boldsymbol{y}_{t-1},\boldsymbol{x}_t)$$

不管是已经被广泛使用的LSTM、GRU还是最近的SRU，都并未脱离这个递归框架。RNN结构本身比较简单，也很适合序列建模，但RNN的明显缺点之一就是无法并行，因此速度较慢，这是递归的天然缺陷。另外我个人觉得RNN无法很好地学习到全局的结构信息，因为它本质是一个马尔科夫决策过程。

第二个思路是CNN层，其实CNN的方案也是很自然的，窗口式遍历，比如尺寸为3的卷积，就是

$$\boldsymbol{y}_t=f(\boldsymbol{x}_{t-1},\boldsymbol{x}_t,\boldsymbol{x}_{t+1})$$

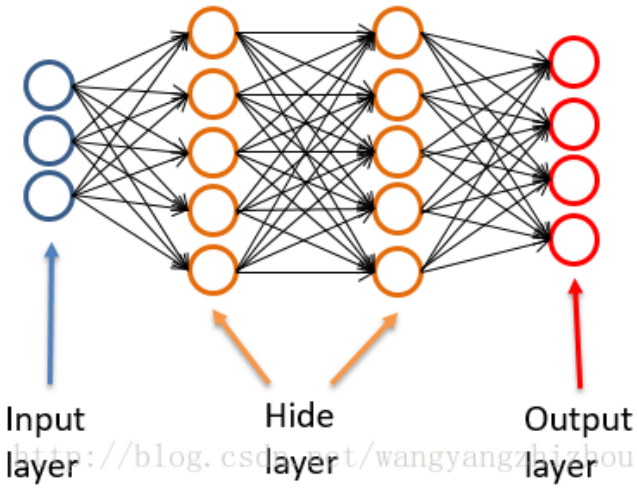
在FaceBook的论文中，纯粹使用卷积也完成了Seq2Seq的学习，是卷积的一个精致且极致的使用案例，热衷卷积的读者必须得好好读读这篇文论。CNN方便并行，而且容易捕捉到一些全局的结构信息，笔者本身是比较偏爱CNN的，在目前的工作或竞赛模型中，我都已经尽量用CNN来代替已有的RNN模型了，并形成了自己的一套使用经验，这部分我们以后再谈。

Google的大作提供了第三个思路：**纯Attention！单靠注意力就可以！**RNN要逐步递归才能获得全局信息，因此一般要双向RNN才比较好；CNN事实上只能获取局部信息，是通过层叠来增大感受野；Attention的思路最为粗暴，它一步到位获取了全局信息！它的解决方案是：

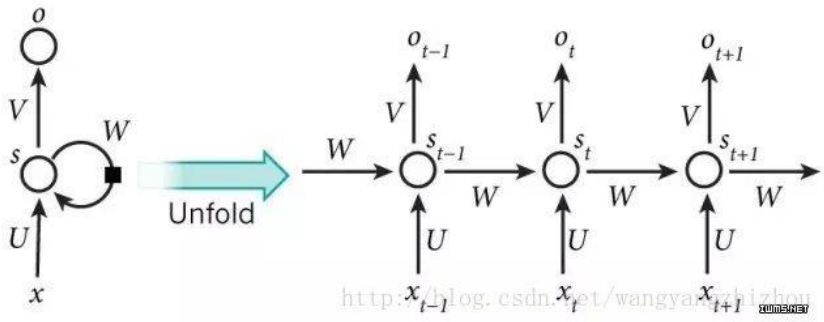
$$\boldsymbol{y}_t=f(\boldsymbol{x}_t,\boldsymbol{A},\boldsymbol{B})$$

其中$\boldsymbol{A},\boldsymbol{B}$是另外一个序列（矩阵）。如果都取$\boldsymbol{A}=\boldsymbol{B}=\boldsymbol{X}$，那么就称为Self Attention，它的意思是直接将$\boldsymbol{x}_t$与原来的每个词进行比较，最后算出$\boldsymbol{y}_t$！

双隐层神经网络结构



RNN 模型



LSTM 网络结构

LSTM模型是由 t 时刻的输入词 X_t ，细胞状态 C_t ，临时细胞状态 $\tilde{C}_t$ ，隐层状态 h_t ，遗忘门 f_t ，记忆门 i_t ，输出门 o_t 组成。LSTM的计算过程可以概括为，通过对细胞状态中信息遗忘和记忆新的信息使得对后续时刻计算有用的信息得以传递，而无用的信息被丢弃，并在每个时间步都会输出隐层状态 h_t ，其中遗忘，记忆与输出由通过上个时刻的隐层状态 h_{t-1} 和当前输入 X_t 计算出来的遗忘门 f_t ，记忆门 i_t ，输出门 o_t 来控制。

总体框架如图1所示。

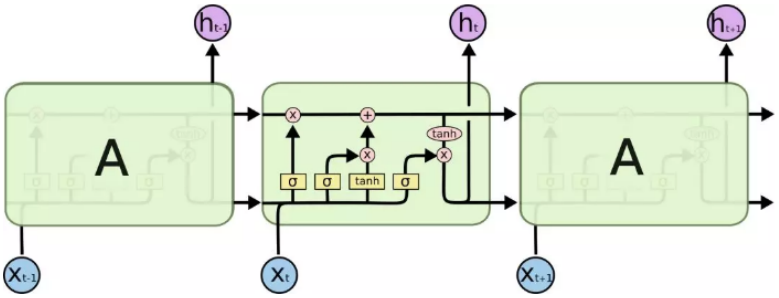
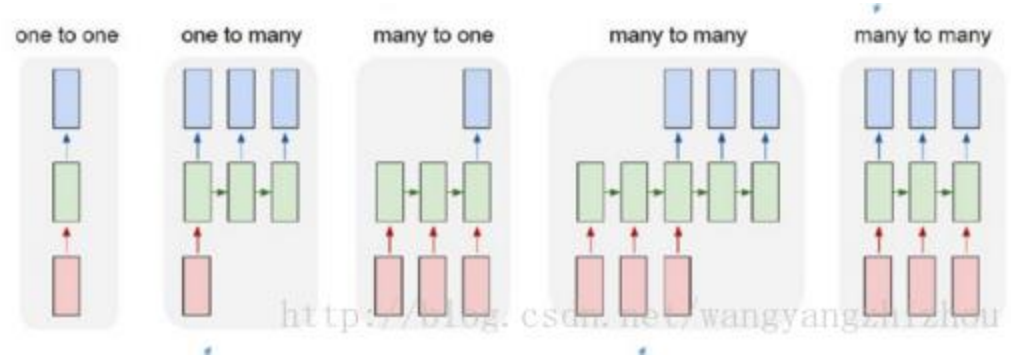


图1. LSTM总体框架

Seq2seq 网络结构

根据输出和输入序列不同数量 rnn 可以有多种不同的结构，不同结构自然就有不同的引用场合。如下图，



one to one 结构，仅仅只是简单的给一个输入得到一个输出，此处并未体现序列的特征，例如图像分类场景。

one to many 结构，给一个输入得到一系列输出，这种结构可用于生产图片描述的场景。

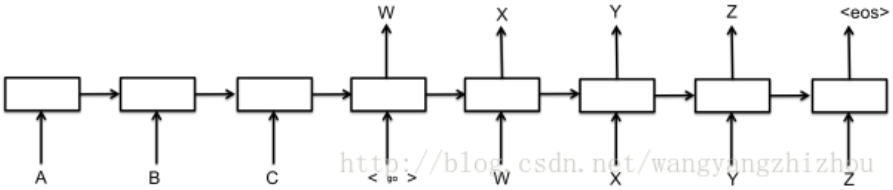
many to one 结构，给一系列输入得到一个输出，这种结构可用于文本情感分析，对一些列的文本输入进行分类，看是消极还是积极情感。

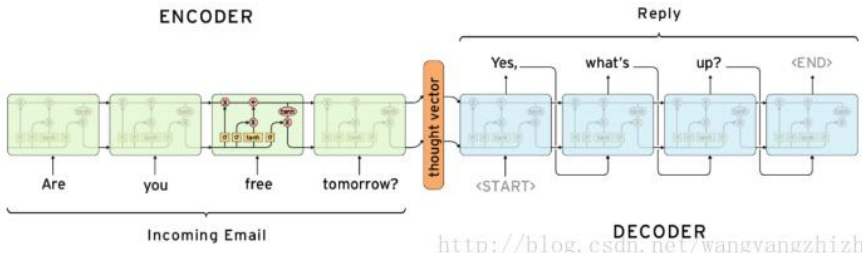
many to many 结构，给一些列输入得到一系列输出，这种结构可用于翻译或聊天对话场景，对输入的文本转换成另外一些列文本。

同步 many to many 结构，它是经典的 rnn 结构，前一输入的状态会带到下一个状态中，而且每个输入都会对应一个输出，我们最熟悉的就是用于字符预测了，同样也可以用于视频分类，对视频的帧打标签。

seq2seq

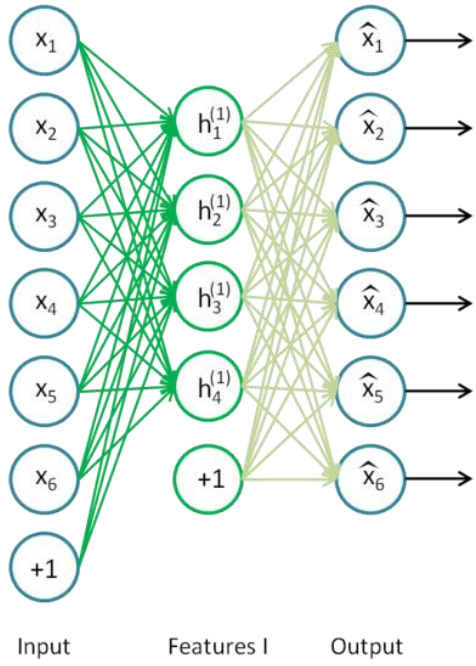
在 many to many 的两种模型中，上图可以看到第四和第五种是有差异的，经典的 rnn 结构的输入和输出序列必须要是等长，它的应用场景也比较有限。而第四种它可以是输入和输出序列不等长，这种模型便是 seq2seq 模型，即 Sequence to Sequence。它实现了从一个序列到另外一个序列的转换，比如 google 曾用 seq2seq 模型加 attention 模型来实现了翻译功能，类似的还可以实现聊天机器人对话模型。经典的 rnn 模型固定了输入序列和输出序列的大小，而 seq2seq 模型则突破了该限制。

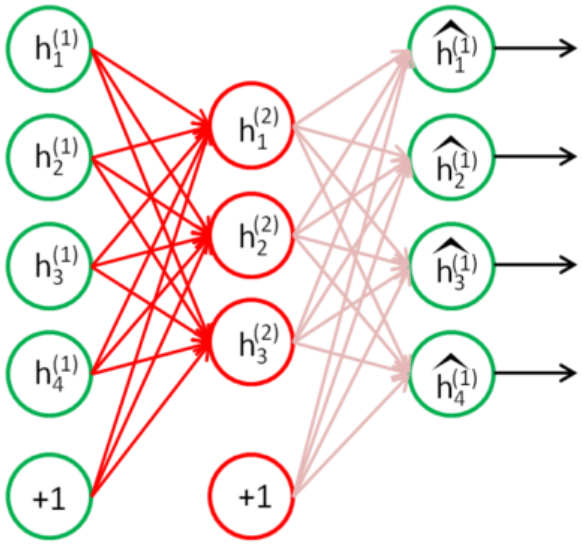


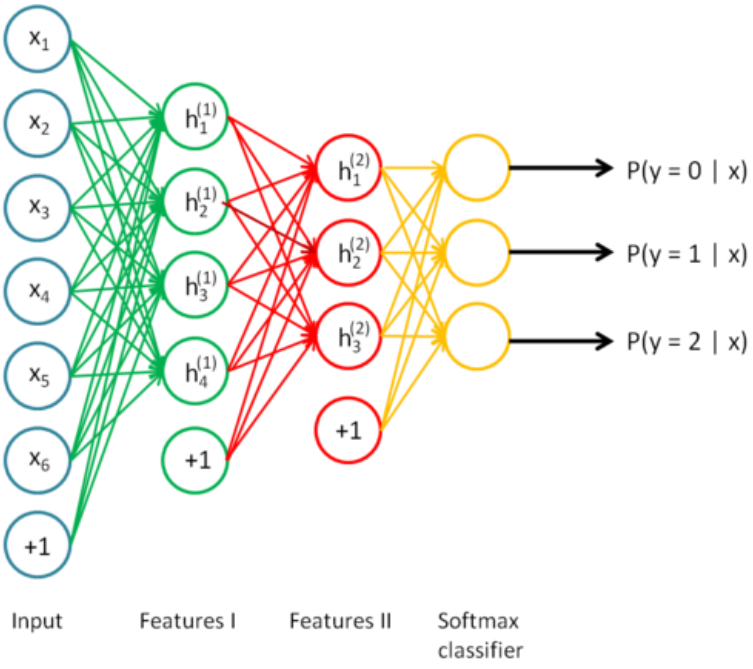


其实对于 seq2seq 的 decoder，它在训练阶段和预测阶段对 rnn 的输出的处理可能是不一样的，比如在训练阶段可能对 rnn 的输出不处理，直接用 target 的序列作为下时刻的输入，如上图一。而预测阶段会将 rnn 的输出当成是下一时刻的输入，因为此时已经没有 target 序列可以作为输入了，如上图

自编码器

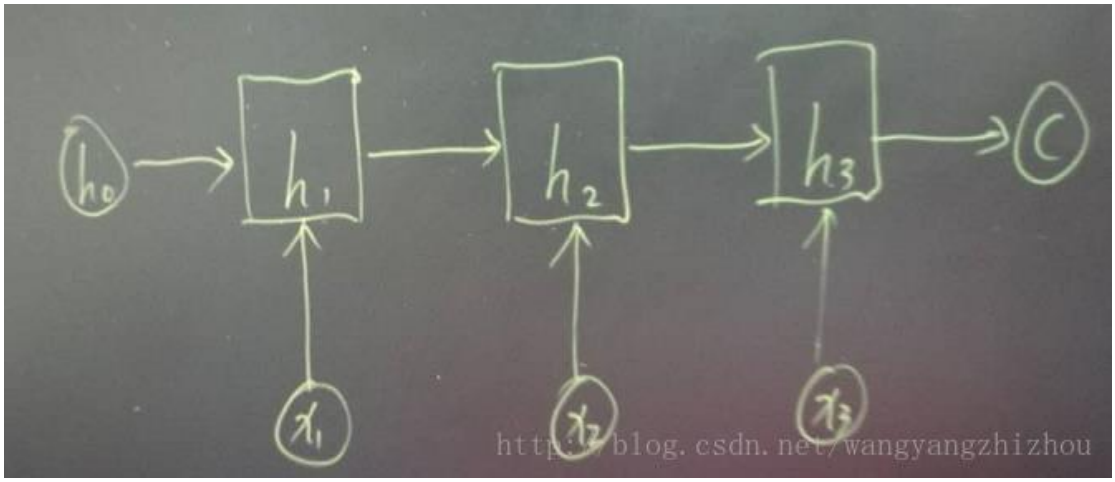




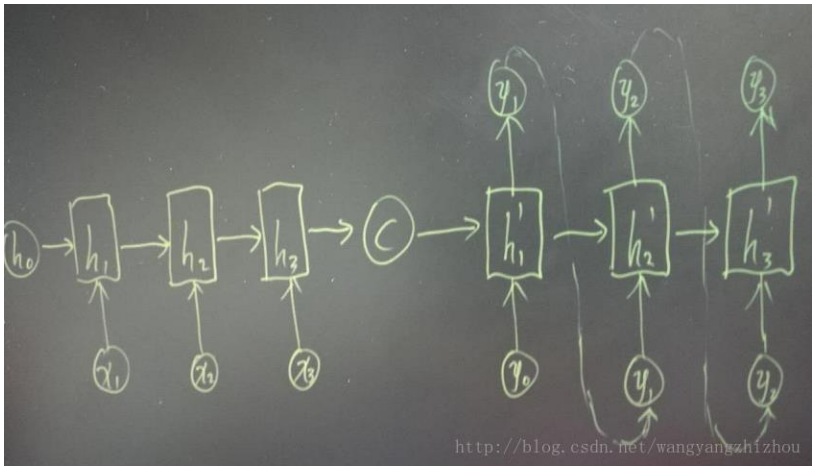


encoder-decoder 结构

seq2seq 属于 encoder-decoder 结构的一种，这里看看常见的 encoder-decoder 结构，基本思想就是利用两个 RNN，一个 RNN 作为 encoder，另一个 RNN 作为 decoder。encoder 负责将输入序列压缩成指定长度的向量，这个向量就可以看成是这个序列的语义，这个过程称为编码，如下图，获取语义向量最简单的方式就是直接将最后一个输入的隐状态作为语义向量 C。也可以对最后一个隐含状态做一个变换得到语义向量，还可以将输入序列的所有隐含状态做一个变换得到语义变量。



而 decoder 则负责根据语义向量生成指定的序列，这个过程也称为解码，如下图，最简单的方式是将 encoder 得到的语义变量作为初始状态输入到 decoder 的 rnn 中，得到输出序列。可以看到上一时刻的输出会作为当前时刻的输入，而且其中语义向量 C 只作为初始状态参与运算，后面的运算都与语义向量 C 无关。



decoder 处理方式还有另外一种，就是语义向量 C 参与了序列所有时刻的运算，如下图，上一时刻的输出仍然作为当前时刻的输入，但语义向量 C 会参与所有时刻的运算。

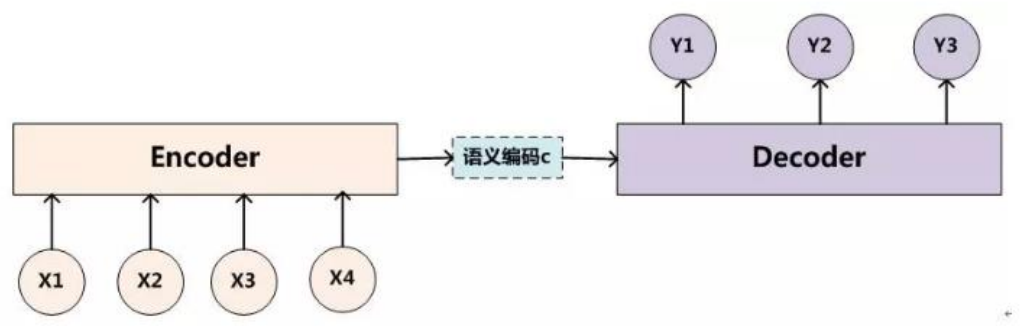


图1 抽象的Encoder-Decoder框架

Encoder-Decoder 框架可以这么直观地去理解：可以把它看作适合处理由一个句子（或篇章）生成另外一个句子（或篇章）的通用处理模型。对于句子对<X,Y>。 --------（思考：<X,Y>对很通用，X 是一个问句，Y 是答案；X 是一个句子，Y 是抽取的关系三元组；X 是汉语句子，Y 是汉语句子的英文翻译。等等），我们的目标是给定输入句子 X，期待通过 Encoder-Decoder 框架来生成目标句子 Y。X 和 Y 可以是同一种语言，也可以是两种不同的语言。而 X 和 Y 分别由各自的单词序列构成：

$$\mathbf{X} = \langle \mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_m \rangle$$
$$\mathbf{Y} = \langle \mathbf{y}_1, \mathbf{y}_2 \dots \mathbf{y}_n \rangle$$

Encoder 顾名思义就是对输入句子 X 进行编码，将输入句子通过非线性变换转化为中间语义表示 C

$$\mathbf{C} = \mathcal{F}(\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_m)$$

对于解码器 Decoder 来说，其任务是根据句子 X 的中间语义表示 C 和之前已经生成的历史信息 y1,y2….yi-1 来生成 i 时刻要生成的单词 yi ：

$$\mathbf{y}_i = \mathcal{G}(\mathbf{C}, \mathbf{y}_1, \mathbf{y}_2 \dots \mathbf{y}_{i-1})$$

每个 yi 都依次这么产生，那么看起来就是整个系统根据输入句子 X 生成了目标句子 Y。 ------（思考：其实这里的 Encoder-Decoder 是一个序列到序列的模型 seq2seq，这个模型是对顺序有依赖的。）

Encoder-Decoder 是个非常通用的计算框架，至于 Encoder 和 Decoder 具体使用什么模型都是由研究者自己定的，常见的比如 CNN/RNN/BiRNN/GRU/LSTM/Deep LSTM 等，这里的变化组合非常多。 ------（思考：人的学习过程包括输入、输出、外界评价。Encoder 模型类似于人的输入学习过程，Decoder 模型类似于人的输出学习过程，对输出的内容进行评价就类似于损失函数。英语老师给

我上了几堂英语课，我在不断的输入 Encoder；突然有一个随堂测试，我得做题输出 Decoder；最后英语老师改卷子，给我一个分数，不对的地方我得反思调整我对输入数据的加工方式。）-------（再思考：关于英语翻译。课本上的单词和课文是原始数据输入，相当于 X；我在大脑里加工这些数据，相当于 Encoder 模型，我的脑子里有很多加工后的数据，相当于 C；现在要让我翻译一个英语句子，这个任务相当于 Y，我不能翻课本，所以我只能借助我脑袋里加工的数据 C 去翻译这个句子，即我得动脑子，相当于 Decoder。　　学习的过程是什么都要学，要分类整理，要增加线索，并不知道未来的某天能用到什么，所以 Encoder-Decoder 是一个泛泛学习的框架）

以上介绍的 Encoder-Decoder 模型是没有体现出“注意力模型”的，所以可以把它看作是注意力不集中的分心模型。为什么说它注意力不集中呢？请观察下目标句子 Y 中每个单词的生成过程如下：

$$y_1 = f(C)$$
$$y_2 = f(C, y_1)$$
$$y_3 = f(C, y_1, y_2)$$

其中 **f 是 decoder 的非线性变换函数**。从这里可以看出，在生成目标句子的单词时，不论生成哪个单词，是 y1,y2 也好，还是 y3 也好，他们使用的句子 X 的语义编码 C 都是一样的，没有任何区别。而语义编码 C 是由句子 X 的每个单词经过 Encoder 编码产生的，这意味着不论是生成哪个单词，y1,y2 还是 y3，其实**句子 X 中任意单词对生成某个目标单词 yi 来说影响力都是相同的，没有任何区别**（其实如果 Encoder 是 RNN 的话，理论上越是后输入的单词影响越大，并非等权的，估计这也是为何 Google 提出 Sequence to Sequence 模型时发现把输入句子逆序输入做翻译效果会更好的小 Trick 的原因）。这就是为何说这个模型**没有体现出注意力**的缘由。

引入 AM 模型，以翻译一个英语句子举例：输入 X：Tom chase Jerry。　　理想输出：汤姆追逐杰瑞。

应该在翻译“杰瑞”的时候，体现出英文单词对于翻译当前中文单词不同的影响程度，比如给出类似下面一个概率分布值：

（Tom,0.3）（Chase,0.2）（Jerry,0.5）

每个英文单词的概率代表了翻译当前单词“杰瑞”时，注意力分配模型分配给不同英文单词的注意力大小。这对于正确翻译目标语单词肯定是有帮助的，因为引入了新的信息。同理，目标句子中的每个单词都应该学会其对应的源语句子中单词的注意力分配概率信息。这意味着在生成每个单词 Yi 的时候，**原先都是相同的中间语义表示 C 会替换成根据当前生成单词而不断变化的 Ci**。理解 AM 模型的

关键就是这里，即由固定的中间语义表示 C 换成了根据当前输出单词来调整成加入注意力模型的变化的 Ci。

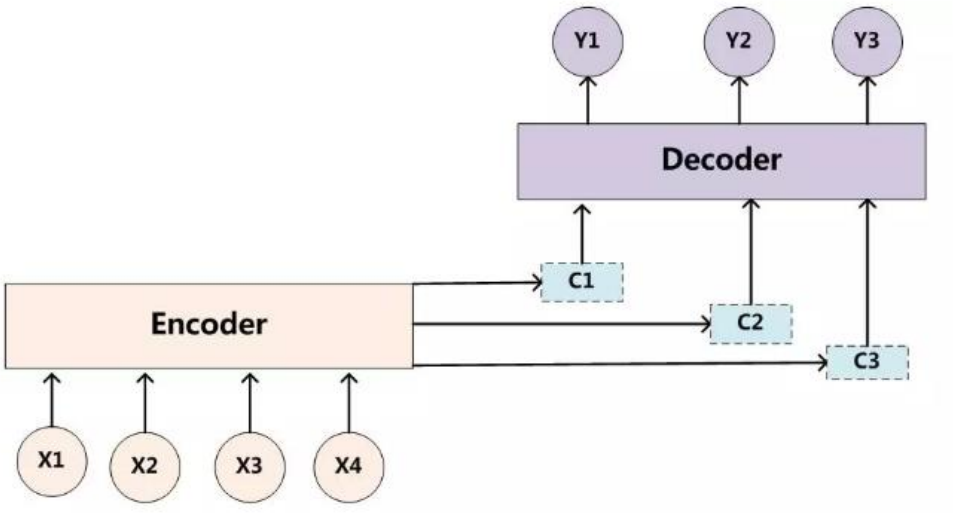


图2 引入AM模型的Encoder-Decoder框架

即生成目标句子单词的过程成了下面的形式：

$$y_1 = f1(C_1)$$
$$y_2 = f1(C_2, y_1)$$
$$y_3 = f1(C_3, y_1, y_2)$$

而**每个Ci可能对应着不同的源语句子单词的注意力分配概率分布**，比如对于上面的英汉翻译来说，其对应的信息可能如下：

$$C_{汤姆} = g(0.6 * f2("Tom"), 0.2 * f2(Chase), 0.2 * f2("Jerry"))$$

$$C_{追逐} = g(0.2 * f2("Tom"), 0.7 * f2(Chase), 0.1 * f2("Jerry"))$$

$$C_{杰瑞} = g(0.3 * f2("Tom"), 0.2 * f2(Chase), 0.5 * f2("Jerry"))$$

其中，**f2函数代表Encoder对输入英文单词的某种变换函数**，比如如果Encoder是用的RNN模型的话，这个f2函数的结果往往是某个时刻输入xi后隐层节点的状态值；**g代表Encoder根据单词的中间表示合成整个句子中间语义表示的变换函数**，一般的做法中，**g函数就是对构成元素加权求和**，也就是常常在论文里看到的下列公式：

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

假设Ci中那个i就是上面的“汤姆”，那么**Tx就是3，代表输入句子的长度**，h1=f(“Tom”)，h2=f(“Chase”),h3=f(“Jerry”)，对应的注意力模型权值分别是0.6,0.2,0.2，**所以g函数就是个加权求和函数**。如果形象表示的话，翻译中文单词“汤姆”的时候，数学公式对应的**中间语义表示Ci的形成过程**类似下图：

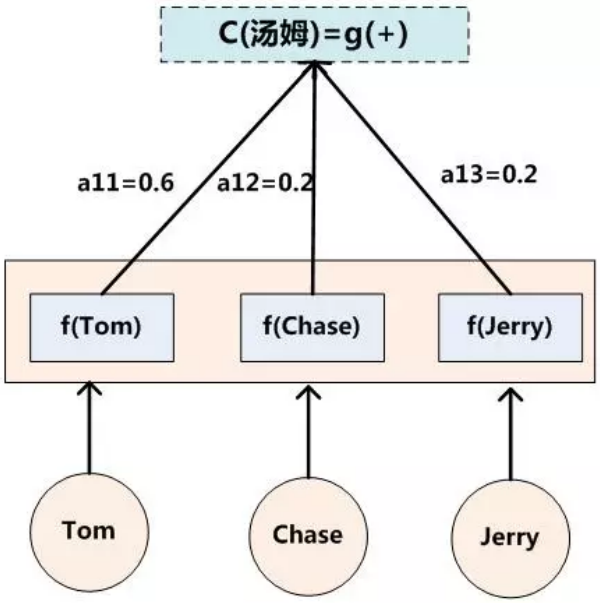


图3 Ci的形成过程

这里还有一个问题：生成目标句子某个单词，比如“汤姆”的时候，**你怎么知道AM模型所需要的输入句子单词注意力分配概率分布值呢？**就是说“汤姆”对应的概率分布：

划重点(注意力权重获取的过程)（Tom,0.3）（Chase,0.2）（Jerry,0.5）是如何得到的呢？

为了便于说明，我们假设对图1的非AM模型的Encoder-Decoder框架进行细化，Encoder采用RNN模型，Decoder也采用RNN模型，这是比较常见的一种模型配置，则图1的图转换为下图：

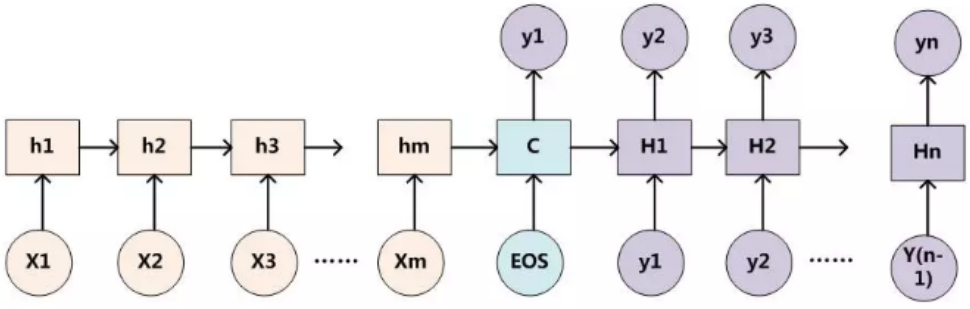


图4 RNN作为具体模型的Encoder-Decoder框架

注意力分配概率分布值的通用计算过程：

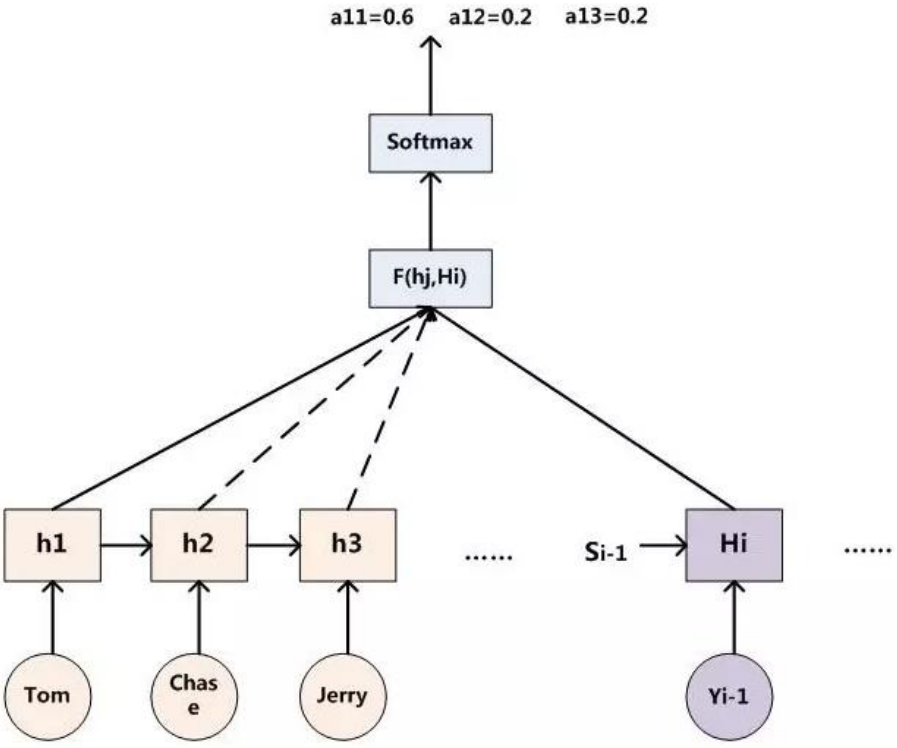
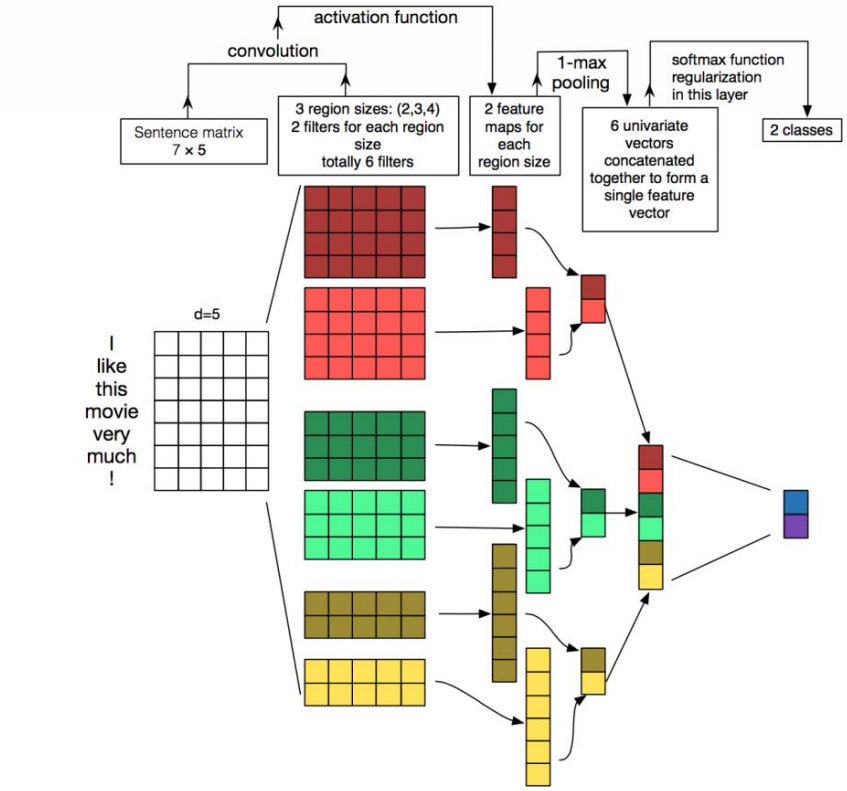


图5 AM注意力分配概率计算

讲解 CNN



LSTM 缺陷
从模型上看，Attention 一般用在 CNN 和 LSTM 上，也可以直接进行纯 Attention 计算。

1）CNN+Attention

CNN 的卷积操作可以提取重要特征，我觉得这也算是 Attention 的思想，但是 CNN 的卷积感受视野是局部的，需要通过叠加多层卷积区去扩大视野。另外，Max Pooling 直接提取数值最大的特征，也像是 hard attention 的思想，直接选中某个特征。
CNN 上加 Attention 可以加在这几方面：

a. 在卷积操作前做 attention，比如 Attention-Based BCNN-1，这个任务是文本蕴含任务需要处理两段文本，同时对两段输入的序列向量进行 attention，计算出特征向量，再拼接到原始向量中，作为卷积层的输入。

b. 在卷积操作后做 attention，比如 Attention-Based BCNN-2，对两段文本的卷积层的输出做 attention，作为 pooling 层的输入。

c. 在 pooling 层做 attention，代替 max pooling。比如 Attention pooling，首先我们用 LSTM 学到一个比较好的句向量，作为 query，然后用 CNN 先学习到一个特征矩阵作为 key，再用 query 对 key 产生权重，进行 attention，得到最后的句向量。

2）LSTM+Attention

LSTM 内部有 Gate 机制，其中 input gate 选择哪些当前信息进行输入，forget gate 选择遗忘哪些过去信息，我觉得这算是一定程度的 Attention 了，而且号称可以解决长期依赖问题，实际上 LSTM 需要一步一步去捕捉序列信息，在长文本上的表现是会随着 step 增加而慢慢衰减，难以保留全部的有用信息。

LSTM 通常需要得到一个向量，再去做任务，常用方式有：

a. 直接使用最后的 hidden state（可能会损失一定的前文信息，难以表达全文）

b. 对所有 step 下的 hidden state 进行等权平均（对所有 step 一视同仁）。

c. Attention 机制，对所有 step 的 hidden state 进行加权，把注意力集中到整段文本中比较重要的 hidden state 信息。性能比前面两种要好一点，而方便可视化观察哪些 step 是重要的，但是要小心过拟合，而且也增加了计算量。

长文本任务，document 级别，因为长文本本身所携带的信息量比较大，可能会带来信息过载问题，很多任务可能只需要用到其中一些关键信息（比如文本分类），所以 Attention 机制用在这里正适合 capture 这些关键信息。

一　ELMo--动态词向量

ELMO(Embeddings from Language Models），被称为时下最好的通用词和句子嵌入方法，来自于语言模型的词向量表示，也是利用了深度上下文单词表征，该模型的优势：
（1）能够处理单词用法中的复杂特性（比如句法和语义）
（2）这些用法在不同的语言上下文中如何变化（比如为词的多义性建模）

ELMo 与 word2vec 最大的不同：
即词向量不是一成不变的，而是**根据上下文而随时变化**，这与 word2vec 具有很大的区别

举个例子：针对某一词多义的词汇 w="苹果"
文本序列 1="我 买了 六斤 苹果。"
文本序列 2="我 买了一个 苹果 7。"
上面两个文本序列中都出现了“苹果”这个词汇，但是在不同的句子中，它们我的含义显示是不同的，一个属于水果领域，一个属于电子产品呢领域，如果针对“苹果”这个词汇同时训练两个词向量来分别刻画不同领域的信息呢？答案就是使用 ELMo。

ELMo 是双向语言模型 biLM 的多层表示的组合，基于大量文本，ELMo 模型是从深层的双向语言模型（deep bidirectional language model）中的内部状态(internal state)学习而来的，而这些词向量很容易加入到 QA、文本对齐、文本分类等模型中，后面会展示一下 ELMo 词向量在各个任务上的表现。

双向语言模型
考虑给定的 N 个词组，计算这句话出现的概率：:
(1) forward language model(前向语言模型)

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, \ldots, t_{k-1}).$$

Forward language model 采用的是利用前面的信息估计后面的信息的方式，具体如下：
每一个 t(k)出现的概率都依赖于前面的所有词语。如果采用 lstm（RNN）模型的话，我们定义如下符号：表示第 k 个单词在第 j 层的输出（注意箭头的方向，LM 表示 language model），如果不理解层的意思，那么请去参考有关 RNN 的资料。

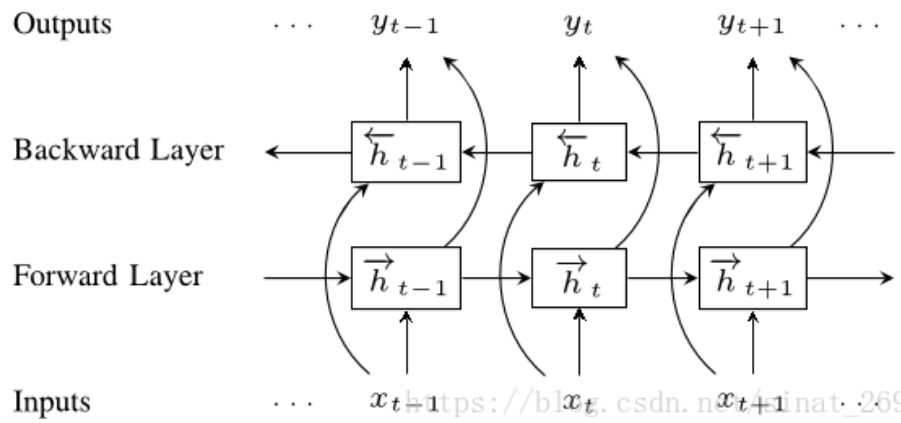
(2) backword language model

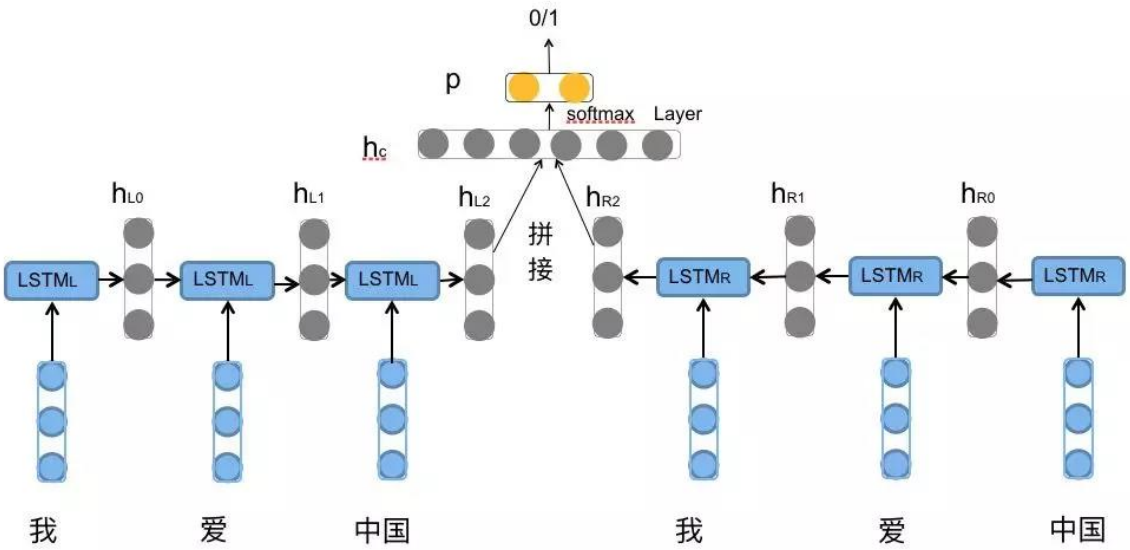
$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, \ldots, t_N).$$

Backward language model 采用的是利用后面的信息估计前面的信息的方式，具体如下（注意和 forward LM 比较）：

每一个 t(k)chu 出现的概率都依赖于后面的所有词语。如果采用 lstm(RNN)模型的话，我们定义如下符号：表示第 k 个单词在第 j 层的输出（注意箭头的方向，代表 forward 或者 backword）。

（3）biLM(bidirectional language model)





1. biLM 则是整合了上面的两种语言模型，目标函数为最大化下面的 log 似然函数：

$$\sum_{k=1}^{N} \left(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \right).$$

2. 从图中可以看出，前半部分是 forward language model，后半部分是 backward language model。

线性组合

（1）哪些是需要线性组合的

对于循环神经网络，每一层都是有输出的（上文的 h 向量）。我们要线性组合是就是上述的所有 h 向量。

（2）为什么要线性组合

文中提到：高层的 h 更加容易捕捉独立与文章的词义信息（the higher level LSTM states capture context-depend aspects of word meaning）,底层的 h 更容易捕捉语法信息（while lower level states model aspects of syntax）。也就是说，不同层次的输出所对应的特征不一样，组合起来更能表达整个意思。

（3）如何线性组合。

$$\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM} .$$

不理解地方：

（1）sj 和 r 的求解方法

$$\begin{cases} p(y_i|y_1, y_2, , , , y_{i-1}, X) = g(y_{i-1}, s_i, c_i) \\ 1)\text{隐状态} \longrightarrow s_i = f(s_{i-1}, y_{i-1}, c_i) \\ 2)\text{注意力向量} \longrightarrow c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \\ 3)\text{权重} \longrightarrow \alpha_{ij} = \frac{e^{e_{ij}}}{\sum_{k=1}^{T_x} e^{e_{ik}}} \\ 4)\text{匹配度} \longrightarrow e_{ij} = a(s_{i-1}, h_j)) \end{cases}$$

让我们仔细地研究下Decoder其中的公式关系。

当预测某个词时：

$$p(y_i|y_1,y_2,\ldots,y_{i-1},X)=g(y_{i-1},s_i,c_i)$$

依赖于三个变量，前刻预测词y_{i-1}，当前的隐状态s_i，当前的上下文向量c_i。

而当前的隐状态：

$$s_i=f(s_{i-1},y_{i-1},c_i)$$

，则依赖于前刻的隐状态s_{i-1}，前刻的预测词y_{i-1}，当前的上下文向量c_i。**不同于常见的RNN结构里面的隐状态，增加对上下文c_i的依赖。**

上下文向量c_i涵盖了当前待预测词y_i需要关注的source信息总和，$c_i=\sum_{j=1}^{T_x}\alpha_{ij}h_j$。到底需要关注哪些source信息呢？对输入序列的对应定义**解释变量**h_j，该变量描述了输入序列位置j处及附近的信息。α_{ij}则表示当前待预测词y_i与source词x_j附近信息的**相关性程度(对每个x_j的关注度)。**

疑问1：为什么不直接model x_j与y_i的关系呢？后文后有解释。

疑问2：如何具体表示α_{ij}呢？

$\alpha_{ij}=\frac{exp(e_{ij})}{\sum_{k=1}^{T_x}exp(e_{ik})}$，根据所有输入词对$y_i$的相关值$e_{ij}$作normalize，得到归一化的概率权重，其中$e_{ij}=a(s_{i-1},h_j)$

疑问3：如何理解e_{ij}能够描述source词x_j(隐藏着$[h_j]$)与y_i相关性。

e_{ij}表达的是输入j附近与输出位置i的相关性评估，e_{ij}反映了h_j在考虑前刻隐状态s_{i-1}在决定当前隐状态s_i和生成预测值y_i时的某种重要性。

还是那个问题，**为什么不直接modelx_j和y_i的关系？**一方面，会跳过解释变量h，没法跟当前的RNN模型结合在一起；另一方面，在预测时，不能直接得到y_i的变量来计算（因为还没预测到），必须要借助前刻的信息s_{i-1}，而s_{i-1}也刚好包括前刻的必备信息。

疑问4：关联函数a怎么表达呢？

这里用了个前向网络$e_{ij}=v_a^Ttanh[W_as_{i-1}+U_ah_j])$来表达（concate的形式，后面会介绍有好几种alignment形式），与模型一块训练。

疑问5：如何在翻译模式中体现h_j呢？

输入序列的解释变量h如何实现的？与s是什么结构关系？解码器: Bidirection GRU for $h_j=[\overrightarrow{h_j};\overleftarrow{h_j}]$，可以得知$h$是BRNN的隐状态，那么$s$又是什么呢？Encode时和Decode时的隐状态名称不同，前者称为h，后者称为s，这下明白了吧。

Transformer

Position Embedding

然而，只要稍微思考一下就会发现，这样的模型**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。

Google 的这个作品中，它的 Position Embedding 有几点区别：

1、以前在 RNN、CNN 模型中其实都出现过 Position Embedding，但在那些模型中，Position Embedding 是锦上添花的辅助手段，也就是“有它会更好、没它也就差一点点”的情况，因为 RNN、CNN 本身就能捕捉到位置信息。但是在这个纯 Attention 模型中，Position Embedding 是位置信息的唯一来源，因此它是模型的核心成分之一，并非仅仅是简单的辅助手段。

2、在以往的 Position Embedding 中，基本都是根据任务训练出来的向量。而 Google 直接给出了一个构造 Position Embedding 的公式：

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

这里的意思是将 id 为 p 的位置映射为一个 d_{pos} 维的位置向量，这个向量的第 i 个元素的数值就是 $PE_i(p)$。Google 在论文中说到他们比较过直接训练出来的位置向量和上述公式计算出来的位置向量，效果是接近的。因此显然我们更乐意使用公式构造的 Position Embedding 了。

3、Position Embedding 本身是一个绝对位置的信息，但在语言中，相对位置也很重要，Google 选择前述的位置向量公式的一个重要原因是：由于我们

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

以及

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

这表明位置 p+k 的向量可以表明位置 p 的向量的线性变换，这提供了表达相对位置信息的可能性。

Bert

ELMO 的设置其实是最符合直觉的预训练套路，两个方向的语言模型刚好可以用来预训练一个 BiLSTM，非常容易理解。那如何用 transformer 在无标注数据行来做一个预训练模型呢？BERT 用了两个反直觉的手段来找到了一个“更好”的方式:

(1) 用比语言模型更简单的任务来做预训练。直觉上要做更深的模型，需要设置一个比语言模型更难的任务，而 BERT 则选择了两个看起来更简单的任务：完形填空和句对预测。

(2) 完形填空任务在直观上很难作为其它任务的预训练任务。在完形填空任务中，需要 mask 掉一些词，这样预训练出来的模型是有缺陷的，因为在其它任务中不能 mask 掉这些词。

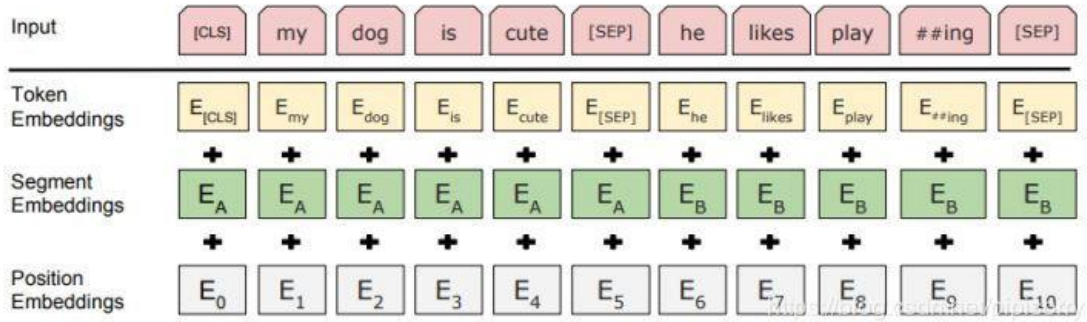
而 BERT 通过随机的方式来解决了这个缺陷：80%加 Mask，10%用其它词随机替换，10%保留原词。这样模型就具备了迁移能力。

Bert 的输入

BERT 的输入的编码向量（d_model=512）是 3 个嵌入特征的单位和，这三个词嵌入特征是：

1. WordPiece 嵌入：WordPiece 是指将单词划分成一组有限的公共子词单元，能在单词的有效性和字符的灵活性之间取得一个折中的平衡。例如示例中‘playing’被拆分成了‘play’和‘ing’。
2. 位置嵌入（Position Embedding）：位置嵌入是指将单词的位置信息编码成特征向量，位置嵌入是向模型中引入单词位置关系的至关重要的一环。和之前文章中的 Transformer 不一样，它不是三角函数，而是学习出来的。
3. 分割嵌入（Segment Embedding）：用于区分两个句子，例如 B 是否是 A 的下文（对话场景，问答场景等）。对于句子对，第一个句子的特征值是 0，第二个句子的特征值是 1。

Note: 图中的两个特殊符号[CLS]和[SEP]，其中[CLS]表示该特征用于分类模型，对非分类模型，该符合可以省去。[SEP]表示分句符号，用于断开输入语料中的两个句子。



BERT 的输入特征。特征是 token 嵌入，位置嵌入和分割嵌入的单位和

Note: 语料的选取很关键，要选用 document-level 的而不是 sentence-level 的，这样可以具备抽象连续长序列特征的能力。

Task #1： Masked Language Model/完形填空

Masked Language Model（MLM）和核心思想取自 Wilson Taylor 在 1953 年发表的一篇论文[cloze procedure: A new tool for measuring readability.]。所谓 MLM 是指在训练的时候随即从输入预料上 mask 掉一些单词，然后通过的上下文预测该单词，该任务非常像我们在中学时期经常做的完形填空。正如传统的语言模型算法和 RNN 匹配那样，MLM 的这个性质和 Transformer 的结构是非常匹配的。

在 BERT 的实验中，15%的 WordPiece Token 会被随机 Mask 掉（而不是把像 cbow 一样把每个词都预测一遍），假如有 1 万篇文章，每篇文章平均有 100 个词汇，随机遮盖 15% 的词汇，模型的任务是正确地预测这 15 万个被遮盖的词汇。在训练模型时，一个句子会被多次喂到模型中用于参数学习，但是 Google 并没有在每次都 mask 掉这些单词，而是在确定要 Mask 掉的单词之后，80%的时候会直接替换为[Mask]，10%的时候将其替换为其它任意单词，10%的时候会保留原始 Token。

80%： my dog is hairy -> my dog is [mask]
10%： my dog is hairy -> my dog is apple
10%： my dog is hairy -> my dog is hairy

这么做的原因是如果句子中的某个 Token100%都会被 mask 掉，那么在 fine-tuning 的时候模型就会有一些没有见过的单词。加入随机 Token 的原因是因为 Transformer 要保持对每个输入 token 的分布式表征，否则模型就会记住这个[mask]是 token ’ hairy ‘，随机词替换会给模型增加一点点噪声，但是因为此时模型不知道哪个词是被随机换了(不像[MASK]，给模型[MASK]则模型知道此处词的是被挖了，他需要预测这个位置是啥)，所以就迫使他去更好地保留每个词的词义，为下游任务提供方便。至于单词带来的负面影响，因为一个单词被随机替换掉的概率只有 15%*10% =1.5%，这个负面影响其实是可以忽略不计的。

另外文章指出每次只预测 15%的单词，因此模型收敛的比较慢。

Note: 为什么要使用 MaskLM 的方式来训练语言模型？

传统的语言模型是单向的（数学上已经定义了）。而且往往都很浅（想象一下 LSTM 堆三层就 train 不动了，就要上各种 trick 了），比如 ELMo。与之前使用的单向语言模型进行预训练不同，BERT 使用遮蔽语言模型来实现预训练的深度双向表示。举个例子：

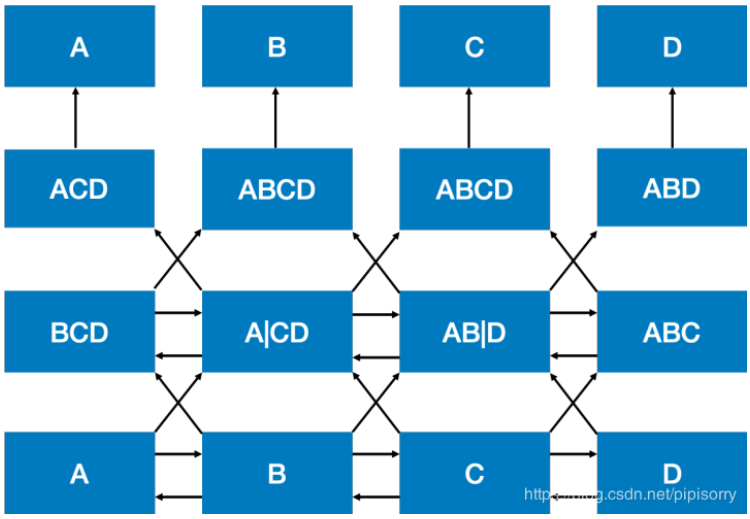
> 今天 天气 不错， 我们 去 公园 玩 吧。

这句话，单向语言模型在学习的时候是从左向右进行学习的，先给模型看到“今天 天气”两个词，然后告诉模型下一个要填的词是“不错”。然而单向语言模型有一个欠缺，就是模型学习的时候总是按照句子的一个方向去学的，因此模型学习每个词的时候只看到了上文，并没有看到下文。更加合理的方式应该是让模型同时通过上下文去学习，这个过程有点类似于完形填空题。例如：

>今天 天气 {}， 我们 去 公园 玩 吧。

通过这样的学习，模型能够更好地把握“不错”这个词所出现的上下文语境。

此前其实也有一些研究在语言模型这个任务上使用了双向的方法，例如在 ELMo 中是通过双向的两层 RNN 结构对两个方向进行建模，但两个方向的 loss 计算相互独立。而 BERT 的作者指出这种两个方向相互独立或只有单层的双向编码可能没有发挥最好的效果，我们可能不仅需要双向编码，还应该要加深网络的层数。但加深双向编码网络却会引入一个问题，导致模型最终可以间接地“窥探”到需要预测的词。这个“窥探”的过程可以用下面的图来表示：



从图中可以看到经过两层的双向操作，每个位置上的输出就已经带有了原本这个位置上的词的信息了。这样的“窥探”会导致模型预测词的任务变得失去意义，因为模型已经看到每个位置上是什么词了。

ref: 《Semi-supervised sequence tagging with bidirectional language models》.

Task #2: Next Sentence Prediction

Next Sentence Prediction（NSP）的任务是判断句子 B 是否是句子 A 的下文。如果是的话输出’IsNext‘，否则输出’NotNext‘。训练数据的生成方式是从平行语料中随机抽取的连续两句话，其中 50%保留抽取的两句话，它们符合 IsNext 关系，另外 50%的第二句话是随机从预料中提取的，它们的关系是 NotNext 的。这个关系保存在[CLS]符号中。

对比：word2vec 的一个精髓是引入了一个优雅的负采样任务来学习词向量（word-level representation），BERT 使用句子级负采样任务学到句子表示。同时在句子表示上，BERT 这里并没有像下游监督任务中的普遍做法一样，在 encoding 的基础上再搞个全局池化之类的，它首先在每个 sequence（对于句子对任务来说是两个拼起来的句子，对于其他任务来说是一个句子）前面加了一个特殊的 token[CLS]。然后让 encoder 对[CLS]进行深度 encoding，深度 encoding 的最高隐层即为整个句子/句对的表示。这个做法乍一看有点费解，不过 Transformer 是可以无视空间和距离的把全局信息 encoding 进每个位置的，而[CLS]作为句子/句对的表示是直接跟分类器的输出层连接的，因此其作为梯度反传路径上的“关卡”，当然会想办法学习到分类相关的上层特征啦。为了让模型能够区分里面的每个词是属于“左句子”还是“右句子”，作者这里引入了“segment embedding”的概念来区分句子。