

# Efficient Estimation of Word Representations in Vector Space

Word2Vec原文

提出了两种新的模型结构，用于计算非常大数据集中单词的连续矢量表示。这些表示的质量是在一个词相似性任务中测量的，并将结果与以前基于不同类型神经网络的最佳表现技术进行比较。

1. Word2Vec两个算法模型的原理是什么，网络结构怎么画？
2. 网络的输入输出是什么？隐藏层的激活函数是什么？输出层的激活函数是什么？
3. Loss？
4. Word2Vec如何获取词向量？
5. 参数如何更新？
6. 分层的softmax怎么做的？二叉树，左右两个概率
7. Word2Vec的参数有哪些？0
8. 局限性？

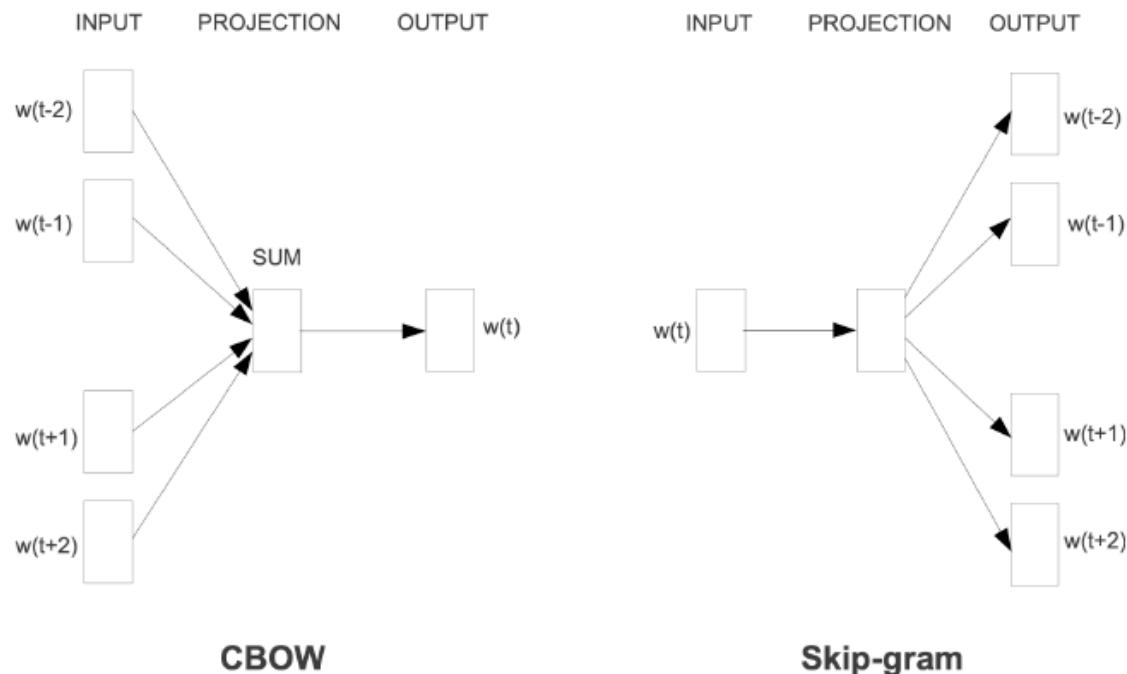
## 编码方式

### One-Hot

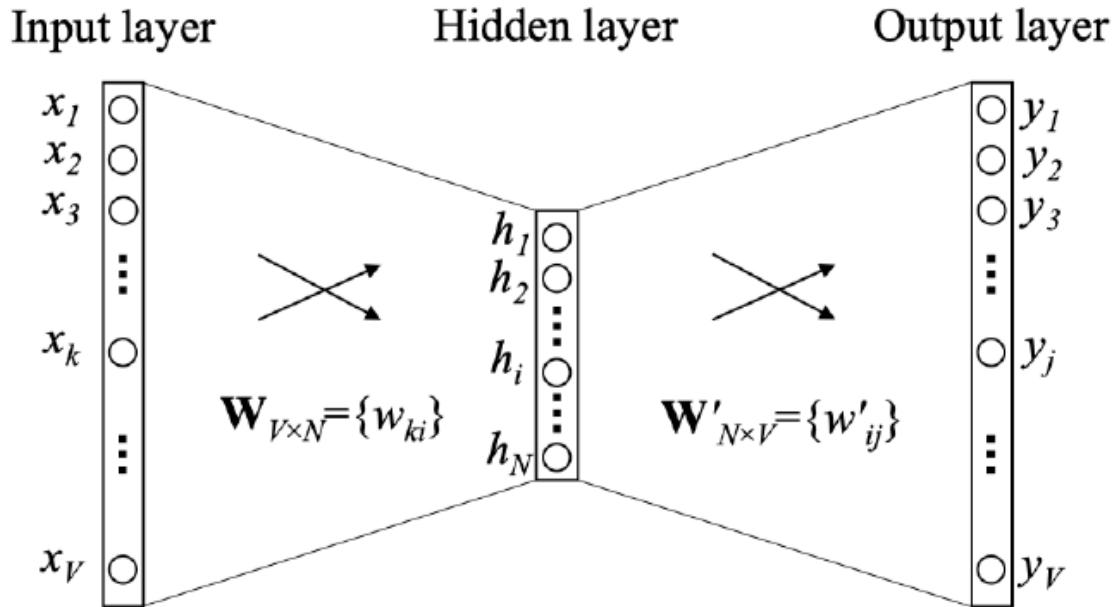
以字符变量的种类作为向量长度，向量中仅有一个元素为1，其余为0，数据稀疏，不适合作为网络的输入，且不能显示词与词之间的关系

### 分布式编码

把字符变量映射到固定长度的向量中，向量空间中的表示字符，且字符间的距离是有意义的，越相似，越相近



COBOW用上下文预测当前单词，Skip-Gram用当前次预测上下文



网络的输入是One-Hot向量 $w_k$ , 隐藏层没有激活函数, 输出层有Softmax函数, 输出的是概率分布, 预测目标为One-Hot向量 $w_j$ 。层与层之间是全连接的

输入层到隐藏层的映射矩阵为  $W_{V \times N}$ , 隐藏层到输出层的映射矩阵为  $W'_{N \times V}$ , 也就是说对于任意的单词  $w_k$  我们都可以有两种表示向量:

$$v_{w_j} = X_k W^T \quad v'_{w_j} = X_k W'$$

其中,  $X_k$  为单词 k 的 One-Hot 编码, 大小为  $(1, N)$ 。这个操作的本质是把 W 的第 k 行复制给 v。举个例子:

$$\begin{bmatrix} 0 & 0 & 0 & \boxed{1} & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ \boxed{10} & \boxed{12} & \boxed{19} \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

为方便起见, 我们将  $v_{w_j}$  成为输入向量, 将  $v'_{w_j}$  成为输出向量。

输出层的计算方式采用 Softmax:

$$p(w_j|w_k) = \frac{\exp(v'_{w_j}^T v_{w_k})}{\sum_{i=1}^V \exp(v'_{w_i}^T v_{w_k})}$$

我们目的是想让  $y_j$  的第 j 个位置的值越大越好, 其他位置的值越小越好。

$$\begin{aligned} \max p(w_j|w_k) &= \max \log(p(w_j|w_k)) \\ &= v_{w_j}^T v_{w_k} - \log\left(\sum_{i=1}^V \exp(v_{w_i}^T v_{w_k})\right) := -E \end{aligned}$$

所以损失函数为：  $E = -\log(p(w_j|w_k))$ 。

我们利用反向传播来更新参数，首先输出向量求偏导：

$$\frac{\partial E}{\partial w_{i,j}'} = \frac{\partial E}{\partial u_j'} \frac{\partial u_j'}{\partial w_{i,j}'} = (y_j - p_j) \cdot h_i$$

其中： $u_j' = v_{w_j}^T v_{w_j}$ ,  $y_j$  为 One-Hot 向量第  $j$  个位置的值,  $p_j$  为预测的向量中第  $j$  个位置的值。

输出向量的参数更新为：

$$w_{ij}'^{(new)} = w_{ij}'^{(old)} - \eta \cdot (y_j - p_j) \cdot h_i$$

值得注意的是，这个更新方程意味着我们需要遍历所有的单词（计算损失函数）。

输入向量的参数更新为：

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i} \frac{\partial h_i}{\partial w_{ki}} = \sum_{j=1}^V (y_j - p_j) \cdot w_{ij}' \cdot x_k$$

其中， $h_i = \sum_{k=1}^V x_k \cdot w_{ki}$ ,  $x_k$  为输入的词向量中第  $k$  维的数值。

则输入向量的参数更新为：

$$w_{ki}^{(new)} = w_{ki}^{(old)} - \eta \cdot \sum_{j=1}^V (y_j - p_j) \cdot w_{ij}' \cdot x_k$$

了解到网络的基本结构和训练方法后，我们一起来看下 Word2vec 两种特殊的网络结构。

## CBOW

---

CBOW 的全称为 Continuous Bag-of-Word Model，通过单词的上下文来预测当前单词。在计算隐藏层的输出时，CBOW 并没有直接使用上下文单词的输入向量，而是将其相加并取其均值（质心），即：

$$h_i = \frac{1}{C} (v_{w_1} + v_{w_2} + \dots + v_{w_C})$$

多个词预测一个词，所以损失函数为：

$$\begin{aligned} E &= -\log(p(w_j|w_{k,1}, w_{k,2}, \dots, w_{k,C})) \\ &= -v_{w_j}^T h_i + \log\left(\sum_{i=1}^V \exp(v_{w_i}^T v_{w_k})\right) \end{aligned}$$

## Skip-Gram

---

Skip-Gram 的结构如下图所示，正好与 CBOW 结构相反。与上面的模型相比，其输出的不再是一个多项式分布，而是 C 个多项式分布（要预测 C 个单词），所以：

$$u_{c,j} = u_j = v'_{w_j}^T v_{w_j} \quad c = 1, 2, \dots, C$$

因为预测数量增多，所以损失函数改为：

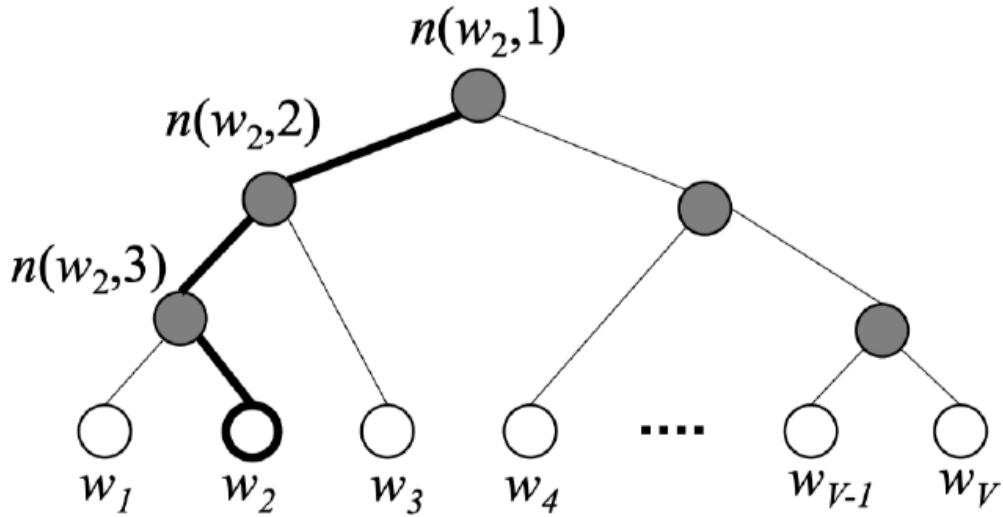
$$\begin{aligned} E &= -\log(p(w_{1,j}, w_{2,j}, \dots, w_{C,j} | w_k)) \\ &= -\log \prod_{c=i}^C \frac{\exp(v'_{w_c}^T v_{w_k})}{\sum_{i=1}^V \exp(v'_{w_i}^T v_{w_k})} \end{aligned}$$

## 分层的Softmax

---

使用哈夫曼二叉树的叶节点来表示语料库的所有单词

词。以下图为例，黑色为内部节点，白色的为叶子节点表示一个单词  $w_i$ ，每一个叶子节点都存在唯一的路径通往根节点，高亮部分为节点  $w_2$  通向根节点的路径， $L(w_2) = 4$ ，我们可以用这条路径来评估单词出现的概率。



在 Hierarchical Softmax 模型中，叶子结点（单词）没有输出向量，但每一个内部节点都有一个输出向量  $v'_{n_{w,i}}$ ，单词作为输出的概率可以表示为：

$$p(w|w_k) = \prod_{i=1}^{L(w)-1} \sigma[\rho[n(w, i+1) = ch(n(w, i))] \cdot v'_{n(w,i)}^T v_{w_k}]$$

其中  $ch(n)$  为节点  $n$  的左子式（Word2Vec 使用的是 Huffman 树，必有左子式）， $n(w, i)$  表示根节点到单词  $w$  的第  $i$  个单元， $v'_{n(w,i)}$  是内部节点  $n(w, i)$  的向量表示， $h$  是隐藏层的输出值， $\sigma(\cdot)$  为 Sigmoid 函数， $\rho[\cdot]$  有如下定义：

$$\rho[x] = \begin{cases} 1 & \text{if } x \text{ is true} \\ -1 & \text{if } x \text{ is false} \end{cases}$$

以上图为例，假设我们需要计算  $w_2$  的输出概率，等同于从根节点到叶节点做随机游走的概率。我们定义内部节点  $n$  左移和右移的概率：

$$p(n, left) = \sigma(v_n^T \cdot v_k)$$

$$p(n, right) = 1 - \sigma(v_n^T \cdot v_k) = \sigma(-v_n^T \cdot v_k)$$

根据  $w_2$  的路径我们有：

$$p(w_2|w_k) = p(n(w_2, 1), left) \cdot p(n(w_2, 2), left) \cdot p(n(w_2, 3), right)$$

$$= \sigma(v'_{n(w_2,1)}^T \cdot v_k) \cdot \sigma(v'_{n(w_2,2)}^T \cdot v_k) \cdot \sigma(-v'_{n(w_2,3)}^T \cdot v_k)$$

不难得出：

$$\sum_{i=1}^V p(w_i|w_k) = 1$$

现在我们来看一下内部节点的参数  $v_{n(w,i)}$  是如何更新的。我们以单个输入输出的简单模型为例：

$$E = -\log(p(w|w_k)) = -\sum_{i=1}^{L(w)-1} \log[\sigma(\rho \cdot v'_{n(w,i)}^T v_{w_k})]$$

对  $v'_{n(w,i)}^T v_{w_k}$  求偏导：

$$\begin{aligned}\frac{\partial E}{\partial v'_{n(w,i)}^T v_{w_k}} &= [\sigma(\rho \cdot v'_{n(w,i)}^T v_{w_k})] \rho \\ &= \begin{cases} \sigma(v'_{n(w,i)}^T v_{w_k}) - 1 & \text{if } \rho = 1 \\ \sigma(v'_{n(w,i)}^T v_{w_k}) & \text{if } \rho = -1 \end{cases} \\ &= \sigma(v'_{n(w,i)}^T v_{w_k}) - t_{n(w,i)}\end{aligned}$$

其中，当  $\rho = 1$  时  $t_{n(w,i)} = 1$ ，当  $\rho = 0$  时  $t_{n(w,i)} = 0$ 。

我们再对内部节点向量  $v'_{n(w,i)}$  求偏导：

$$\frac{\partial E}{\partial v'_{n(w,i)}} = \frac{\partial E}{\partial v'_{n(w,i)}^T v_{w_k}} \cdot \frac{\partial v'_{n(w,i)}^T v_{w_k}}{\partial v'_{n(w,i)}} = (\sigma(v'_{n(w,i)}^T v_{w_k}) - t_{n(w,i)}) v_{w_k}$$

所以内部节点向量的更新公式为：

$$v'_{n(w,i)}^{(new)} = v'_{n(w,i)}^{(old)} - \eta (\sigma(v'_{n(w,i)}^T v_{w_k}) - t_{n(w,i)}) v_{w_k}$$

我们可以把  $\sigma(v'_{n(w,i)}^T v_{w_k}) - t_{n(w,i)}$  理解为内部节点路径的预测误差，在实际的训练过程中，这个误差会非常小。

在实际的应用中，Huffman 树将代替原本的隐藏层到输出层的结构。

## 负采样

以一定的概率选取负样本，使得每次迭代只需要修改一部分参数，给定一些变量及其概率，随机采样使得其满足变量出现的概率。

节省了计算量

保证了模型训练的效果，其一模型每次只需要更新采样的词的权重，不用更新所有的权重，那样会很慢，其二中心词其实只跟它周围的词有关系，位置离着很远的词没有关系，也没必要同时训练更新 negative sampling 每次让一个训练样本仅仅更新一小部分的权重参数，从而降低梯度下降过程中的计算量。

如果 vocabulary 大小为 1 万时，当输入样本 ("fox", "quick") 到神经网络时，“fox”经过 one-hot 编码，在输出层我们期望对应 “quick” 单词的那个神经元结点输出 1，其余 9999 个都应该输出 0。在这里，这 9999 个我们期望输出为 0 的神经元结点所对应的单词我们为 negative word. negative sampling 的想法也很直接，将随机选择一小部分的 negative words，比如选 10 个 negative words 来更新对应的权重参数。

先将概率以累积概率分布的形式分布到一条线段上，以  $a = 0.2, b = 0.3, c = 0.5$  为例， $a$  所处线段为  $[0, 0.2]$ ， $b$  所处线段为  $[0.2, 0.5]$ ， $c$  所处线段为  $[0.5, 1]$ ，然后定义一个大小为  $m$  的数组，并与上面的线段做一次映射，这样我们便知道了数组内的每个单元所对应的字符了，这种情况下算法的时间复杂度为  $O(1)$ ，空间复杂度为  $O(m)$ ， $m$  越小精度越大，但空间复杂度也会变得更大。

## Jay Alammar 文章

嵌入 (embedding) 是机器学习中最迷人的想法之一。如果你曾经使用 Siri、Google Assistant、Alexa、Google 翻译，甚至智能手机键盘进行下一词预测，那么你很有可能从这个已经成为自然语言处理模型核心的想法中受益。

在过去的几十年中，嵌入技术用于神经网络模型已有相当大的发展。尤其是最近，其发展包括导致BERT和GPT2等尖端模型的语境化嵌入。

Word2vec是一种有效创建词嵌入的方法，它自2013年以来就一直存在。但除了作为词嵌入的方法之外，它的一些概念已经被证明可以有效地创建推荐引擎和理解时序数据。在商业的、非语言的任务中。像Airbnb、阿里巴巴、Spotify这样的公司都从NLP领域中提取灵感并用于产品中，从而为新型推荐引擎提供支持。

在这篇文章中，我们将讨论嵌入的概念，以及使用word2vec生成嵌入的机制。让我们从一个例子开始，熟悉使用向量来表示事物。你是否知道你的个性可以仅被五个数字的列表（向量）表示？

### 个性嵌入：你是什么样的人？

如何用到100的范围来表示你是多么内向/外向（其中0是最内向的，100是最外向的）？你有没有做过像MBTI那样的人格测试，或者五大人格特质测试？如果你还没有，这些测试会问你一系列的问题，然后在很多维度给你打分，内向/外向就是其中之一。

Openness to experience	79	out of 100
Agreeableness	75	out of 100
Conscientiousness	42	out of 100
Negative emotionality	50	out of 100
Extraversion	58	out of 100

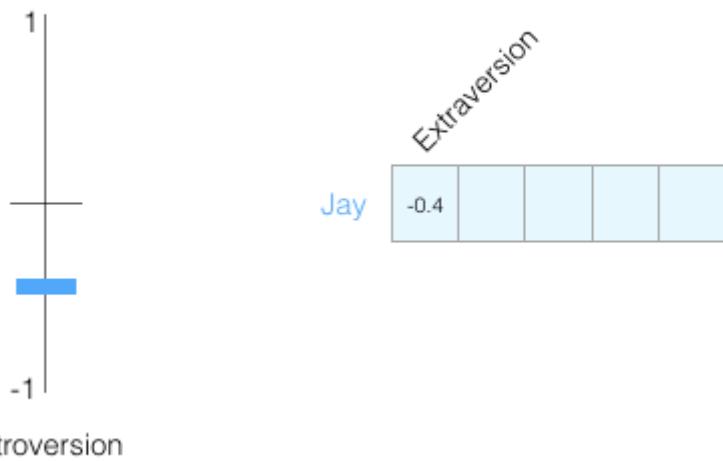
五大人格特质测试测试结果示例。它可以真正告诉你很多关于你自己的事情，并且在学术、人格和职业成功方面都具有预测能力。此处可以找到测试结果。

假设我的内向/外向得分为38/100。我们可以用这种方式绘图：



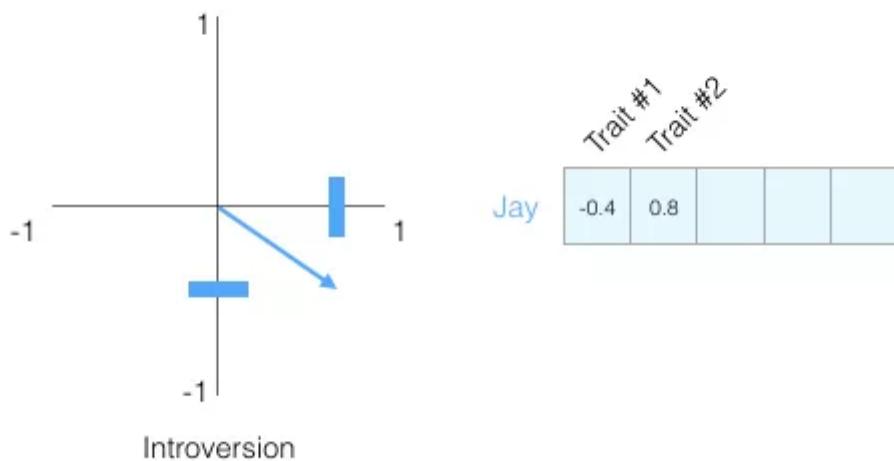
让我们把范围收缩到-1到1：

Extraversion



当你只知道这一条信息的时候，你觉得你有多了解这个人？了解不多。人很复杂，让我们添加另一测试的得分作为新维度。

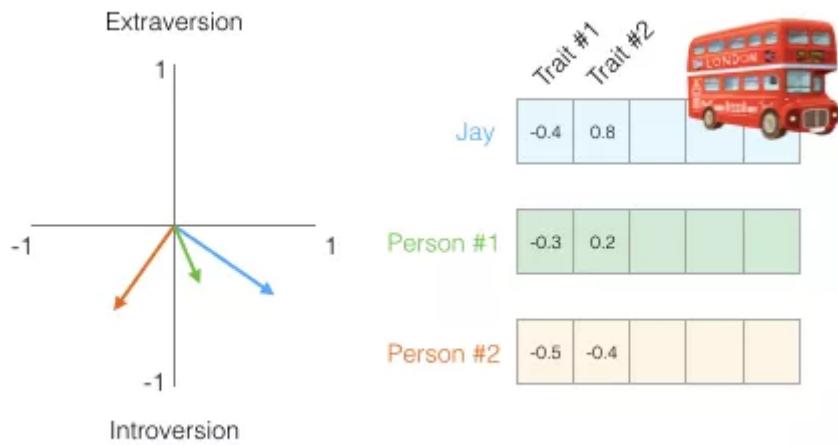
Extraversion



我们可以将两个维度表示为图形上的一个点，或者作为从原点到该点的向量。我们拥有很棒的工具来处理即将上场的向量们。

我已经隐藏了我们正在绘制的人格特征，这样你会渐渐习惯于在不知道每个维度代表什么的情况下，从一个人格的向量表示中获得价值信息。

我们现在可以说这个向量部分地代表了我的人格。当你想要将另外两个人与我进行比较时，这种表示法就有用了。假设我被公共汽车撞了，我需要被性格相似的人替换，那在下图中，两个人中哪一个更像我？



处理向量时，计算相似度得分的常用方法是余弦相似度：

$$\text{cosine\_similarity}(\begin{matrix} \text{Jay} \\ -0.4 & 0.8 \end{matrix}, \begin{matrix} \text{Person \#1} \\ -0.3 & 0.2 \end{matrix}) = 0.87 \quad \checkmark$$

$$\text{cosine\_similarity}(\begin{matrix} \text{Jay} \\ -0.4 & 0.8 \end{matrix}, \begin{matrix} \text{Person \#2} \\ -0.5 & -0.4 \end{matrix}) = -0.20$$

1号替身在性格上与我更相似。指向相同方向的向量（长度也起作用）具有更高的余弦相似度。

再一次，两个维度还不足以捕获有关不同人群的足够信息。心理学已经研究出了五个主要人格特征（以及大量的子特征），所以让我们使用所有五个维度进行比较：

	Trait #1	Trait #2	Trait #3	Trait #4	Trait #5
Jay	-0.4	0.8	0.5	-0.2	0.3
Person #1	-0.3	0.2	0.3	-0.4	0.9
Person #2	-0.5	-0.4	-0.2	0.7	-0.1

使用五个维度的问题是我们不能在二维平面绘制整齐小箭头了。这是机器学习中的常见问题，我们经常需要在更高维度的空间中思考。但好在余弦相似度仍然有效，它适用于任意维度：

Jay                              Person #1

`cosine_similarity(`

-0.4	0.8	0.5	-0.2	0.3
------	-----	-----	------	-----

 , 

-0.3	0.2	0.3	-0.4	0.9
------	-----	-----	------	-----

`) = 0.66` ✓

Jay                              Person #2

`cosine_similarity(`

-0.4	0.8	0.5	-0.2	0.3
------	-----	-----	------	-----

 , 

-0.5	-0.4	-0.2	0.7	-0.1
------	------	------	-----	------

`) = -0.37`

余弦相似度适用于任意数量的维度。这些得分比上次的得分要更好，因为它们是根据被比较事物的更高维度算出的。

在本节的最后，我希望提出两个中心思想：

1.我们可以将人和事物表示为代数向量（这对机器来说很棒！）。

2.我们可以很容易地计算出相似的向量之间的相互关系。

1- We can represent things  
(and people) as vectors of  
numbers  
(Which is great for machines!)

Jay    

-0.4	0.8	0.5	-0.2	0.3
------	-----	-----	------	-----

2- We can easily calculate how  
similar vectors are to each other

The people most similar to Jay are:

`cosine_similarity` ▼

Person #1	0.86
Person #2	0.5
Person #3	-0.20

## 词嵌入

通过上文的理解，我们继续看看训练好的词向量实例（也被称为词嵌入）并探索它们的一些有趣属性。

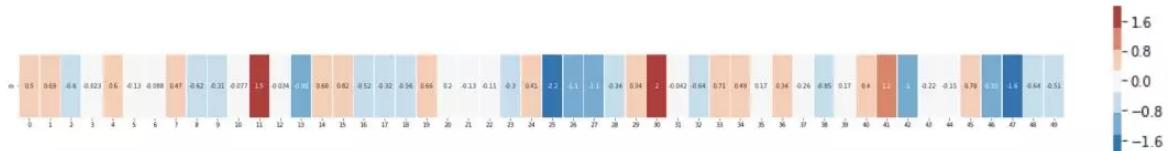
这是一个单词“king”的词嵌入（在维基百科上训练的GloVe向量）：

```
[ 0.50451, 0.68607, -0.59517, -0.022801, 0.60046, -0.13498, -0.08813, 0.47377, -0.61798,
-0.31012, -0.076666, 1.493, -0.034189, -0.98173, 0.68229, 0.81722, -0.51874, -0.31503, -0.55809
, 0.66421, 0.1961, -0.13495, -0.11476, -0.30344, 0.41177, -2.223, -1.0756, -1.0783, -0.34354,
0.33505, 1.9927, -0.04234, -0.64319, 0.71125, 0.49159, 0.16754, 0.34344, -0.25663, -0.8523,
0.1661, 0.40102, 1.1685, -1.0137, -0.21585, -0.15155, 0.78321, -0.91241, -1.6106, -0.64426,
-0.51042 ]
```

这是一个包含50个数字的列表。通过观察数值我们看不出什么，但是让我们稍微给它可视化，以便比较其它词向量。我们把所有这些数字放在一行：



让我们根据它们的值对单元格进行颜色编码（如果它们接近2则为红色，接近0则为白色，接近-2则为蓝色）：



我们将忽略数字并仅查看颜色以指示单元格的值。现在让我们将“king”与其它单词进行比较：

“king”



“Man”

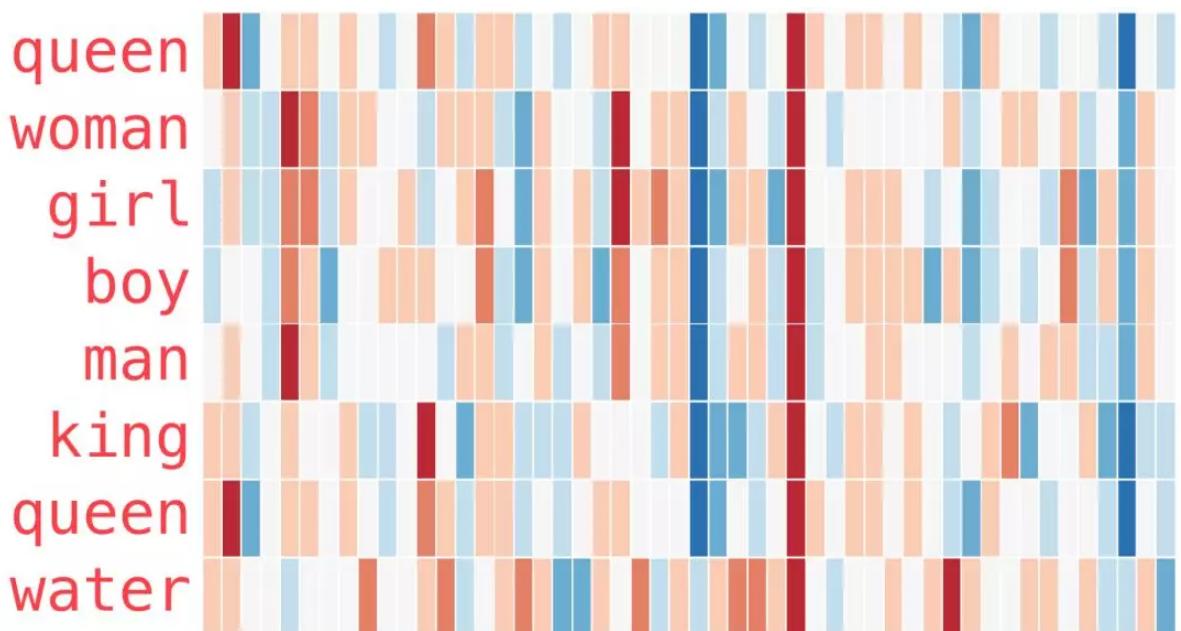


“Woman”



看看“Man”和“Woman”彼此之间是如何比它们任一个单词与“King”相比更相似的？这暗示你一些事情。这些向量图示很好的展现了这些单词的信息/含义/关联。

这是另一个示例列表（通过垂直扫描列来查找具有相似颜色的列）：



有几个要点需要指出：

- 1.所有这些不同的单词都有一条直的红色列。它们在这个维度上是相似的（虽然我们不知道每个维度是什么）
- 2.你可以看到“woman”和“girl”在很多地方是相似的，“man”和“boy”也是一样
- 3.“boy”和“girl”也有彼此相似的地方，但这些地方却与“woman”或“man”不同。这些是否可以总结出一个模糊的“youth”概念？可能吧。
- 4.除了最后一个单词，所有单词都是代表人。我添加了一个对象“water”来显示类别之间的差异。你可以看到蓝色列一直向下并在“water”的词嵌入之前停下了。
- 5.“king”和“queen”彼此之间相似，但它们与其它单词都不同。这些是否可以总结出一个模糊的“royalty”概念？

**类比**

展现嵌入奇妙属性的著名例子是类比。我们可以添加、减去词嵌入并得到有趣的结果。一个著名例子是公式：“king”-“man”+“woman”：

```
model.most_similar(positive=["king", "woman"], negative=["man"])

[('queen', 0.8523603677749634),
 ('throne', 0.7664333581924438),
 ('prince', 0.7592144012451172),
 ('daughter', 0.7473883032798767),
 ('elizabeth', 0.7460219860076904),
 ('princess', 0.7424570322036743),
 ('kingdom', 0.7337411642074585),
 ('monarch', 0.721449077129364),
 ('eldest', 0.7184862494468689),
 ('widow', 0.7099430561065674)]
```

在python中使用Gensim库，我们可以添加和减去词向量，它会找到与结果向量最相似的单词。该图像显示了最相似的单词列表，每个单词都具有余弦相似性。

我们可以像之前一样可视化这个类比：

king - man + woman  $\approx$  queen



由“king-man + woman”生成的向量并不完全等同于“queen”，但“queen”是我们在此集合中包含的400,000个字嵌入中最接近它的单词。

现在我们已经看过训练好的词嵌入，接下来让我们更多地了解训练过程。但在我们开始使用word2vec之前，我们需要看一下词嵌入的父概念：神经语言模型。

## 语言模型

如果要举自然语言处理最典型的例子，那应该就是智能手机输入法中的下一单词预测功能。这是个被数十亿人每天使用上百次的功能。



下一单词预测是一个可以通过语言模型实现的任务。语言模型会通过单词列表(比如说两个词)去尝试预测可能紧随其后的单词。

在上面这个手机截屏中，我们可以认为该模型接收到两个绿色单词(thou shalt)并推荐了一组单词("not"就是其中最有可能被选用的一个)：

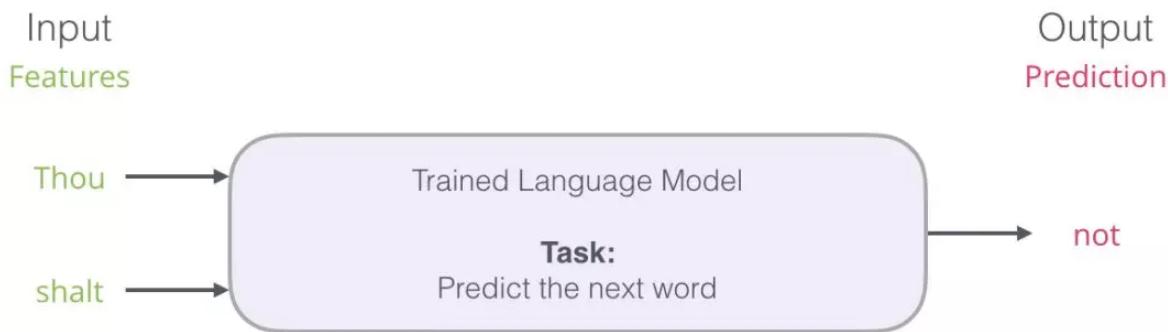
## input/feature #1

## input/feature #2

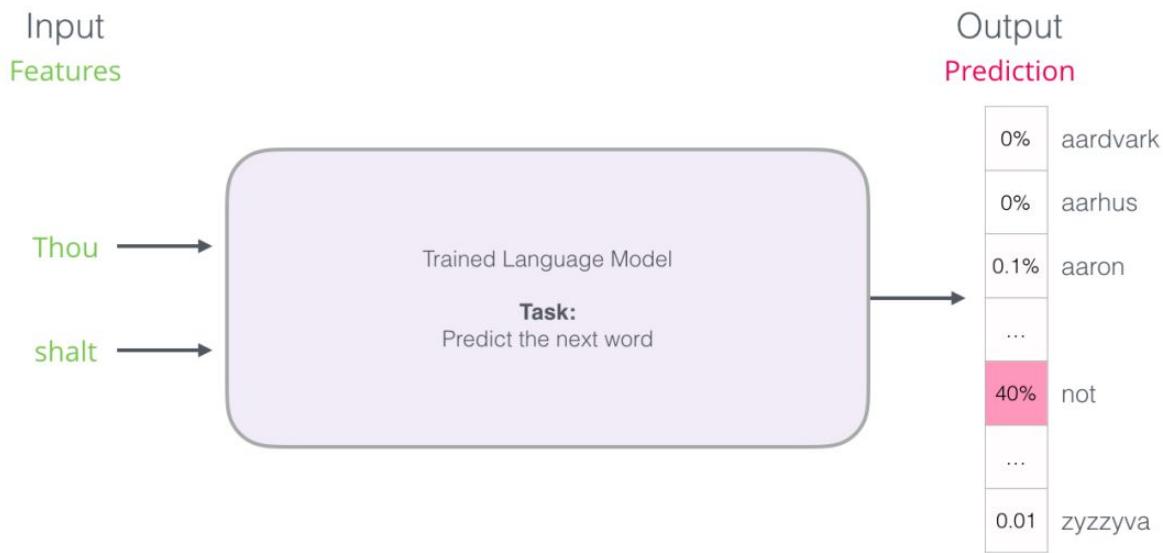
## output/label

# Thou shalt

我们可以把这个模型想象为这个黑盒：

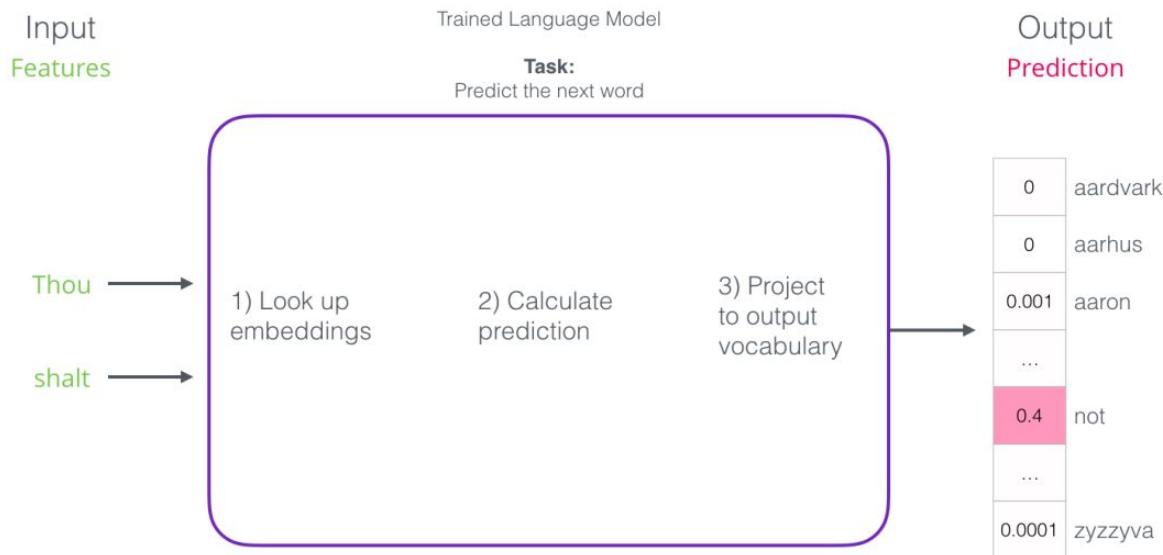


但事实上，该模型不会只输出一个单词。实际上，它对所有它知道的单词(模型的词库，可能有几千到几百万个单词)的按可能性打分，输入法程序会选出其中分数最高的推荐给用户。

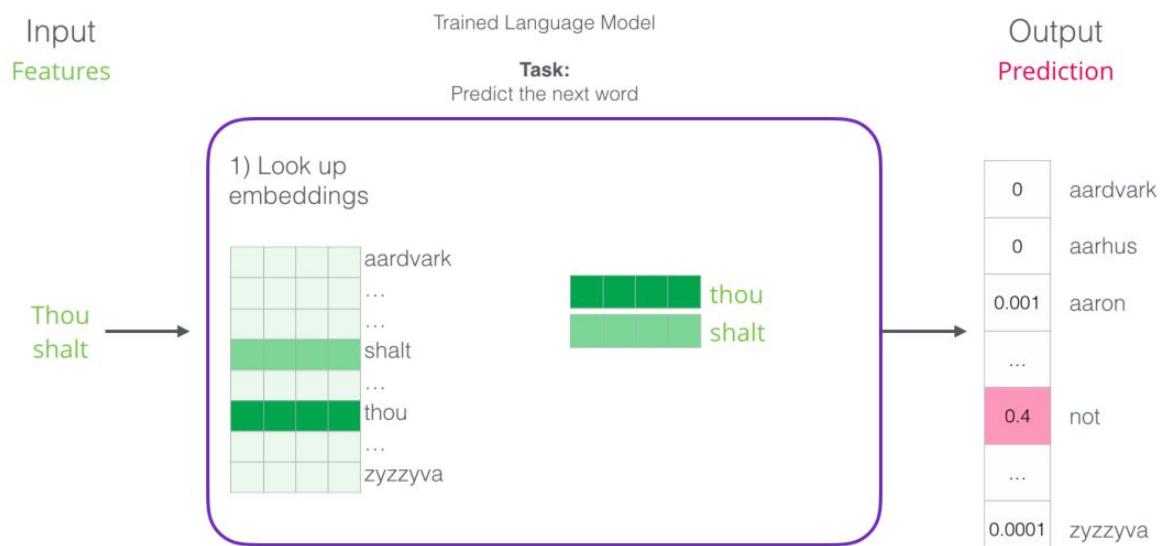


自然语言模型的输出就是模型所知单词的概率评分，我们通常把概率按百分比表示，但是实际上，40%这样的分数在输出向量组是表示为0.4

自然语言模型(请参考Bengio 2003)在完成训练后，会按如下中所示法人三步完成预测：



第一步与我们最相关，因为我们讨论的就是Embedding。模型在经过训练之后会生成一个映射单词表所有单词的矩阵。在进行预测的时候，我们的算法就是在这个映射矩阵中查询输入的单词，然后计算出预测值：



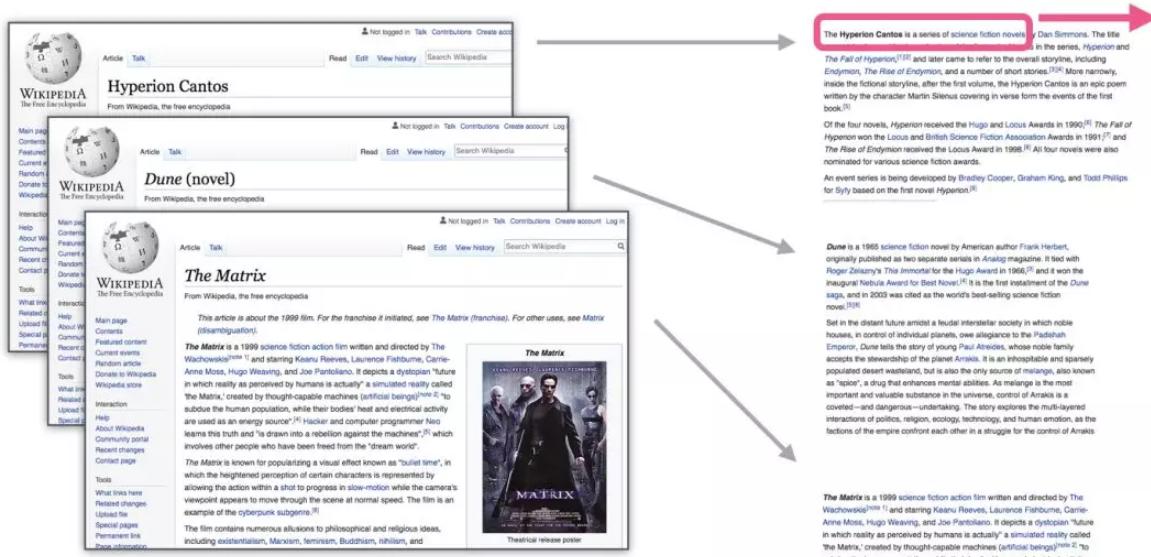
现在让我们重点放到模型训练上，来学习一下如何构建这个映射矩阵。

## 语言模型训练

相较于多数其他机器学习模型，语言模型有一个很大优势，那就是我们有丰富的文本来训练语言模型。所有我们的书籍、文章、维基百科、及各种类型的文本内容都可用。相比之下，许多其他机器学习的模型开发就需要手工设计数据或者专门采集数据。

我们通过找常出现在每个单词附近的词，就能获得它们的映射关系。机制如下：

- 先是获取大量文本数据(例如所有维基百科内容)
- 然后我们建立一个可以沿文本滑动的窗(例如一个窗里包含三个单词)
- 利用这样的滑动窗就能为训练模型生成大量样本数据。



当这个窗口沿着文本滑动时，我们就能(真实地)生成一套用于模型训练的数据集。为了明确理解这个过程，我们看下滑动窗是如何处理这个短语的：

在一开始的时候，窗口锁定在句子的前三个单词上：

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou shalt not make a machine in the likeness of a human mind ...

Dataset

input 1	input 2	output

我们把前两个单词单做特征，第三个单词单做标签：

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou shalt not make a machine in the ...

Dataset

input 1	input 2	output
thou	shalt	not

这时我们就生产了数据集中的第一个样本，它会被用在我们后续的语言模型训练中。

接着，我们将窗口滑动到下一个位置并生产第二个样本：

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	

input 1	input 2	output
thou	shalt	not
shalt	not	make

这时第二个样本也生成了。

不用多久，我们就能得到一个较大的数据集，从数据集中我们能看到在不同的单词组后面会出现的单词：

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

在实际应用中，模型往往在我们滑动窗口时就被训练的。但是我觉得将生成数据集和训练模型分为两个阶段会显得更清晰易懂一些。除了使用神经网络建模之外，大家还常用一项名为N-gams的技术进行模型训练。

如果想了解现实产品从使用N-gams模型到使用神经模型的转变，可以看一下Swiftkey (我最喜欢的安卓输入法)在2015年的发表一篇博客，文中介绍了他们的自然语言模型及该模型与早期N-gams模型的对比。我很喜欢这个例子，因为这个它能告诉你如何在营销宣讲中把Embedding的算法属性解释清楚。

## 顾及两头

根据前面的信息进行填空：

Jay was hit by a \_\_\_\_\_

在空白前面，我提供的背景是五个单词(如果事先提及到'bus')，可以肯定，大多数人都会把bus填入空白中。但是如果我再给你一条信息——比如空白后的一个单词，那答案会有变吗？

Jay was hit by a \_\_\_\_\_ bus

这下空白处改填的内容完全变了。这时'red'这个词最有可能适合这个位置。从这个例子中我们能学到，一个单词的前后词语都带信息价值。事实证明，我们需要考虑两个方向的单词(目标单词的左侧单词与右侧单词)。那我们该如何调整训练方式以满足这个要求呢，继续往下看。

## Skipgram模型

我们不仅要考虑目标单词的前两个单词，还要考虑其后两个单词。

Jay was hit by a \_\_\_\_\_ bus in...

by	a	red	bus	in
----	---	-----	-----	----

如果这么做，我们实际上构建并训练的模型就如下所示：

input 1	input 2	input 3	input 4	output
by	a	bus	in	red

上述的这种架构被称为连续词袋(CBOW)，在一篇关于word2vec的论文中有阐述。

还有另一种架构，它不根据前后文(前后单词)来猜测目标单词，而是推测当前单词可能的前后单词。我们设想一下滑动窗在训练数据时如下图所示：

Jay was hit by a red bus in...

--	--	--	--	--

绿框中的词语是输入词，粉框则是可能的输出结果；

这里粉框颜色深度呈现不同，是因为滑动窗给训练集产生了4个独立的样本：

Jay was hit by a red bus in...

by	a	red	bus	in
----	---	-----	-----	----

input	output
red	by
red	a
red	bus
red	in

这种方式称为Skipgram架构。我们可以像下图这样将展示滑动窗的内容。

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word

这样就为数据集提供了4个样本:

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a

然后我们移动滑动窗到下一个位置:

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a

这样我们又产生了接下来4个样本:

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

在移动几组位置之后，我们就能得到一批样本:

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

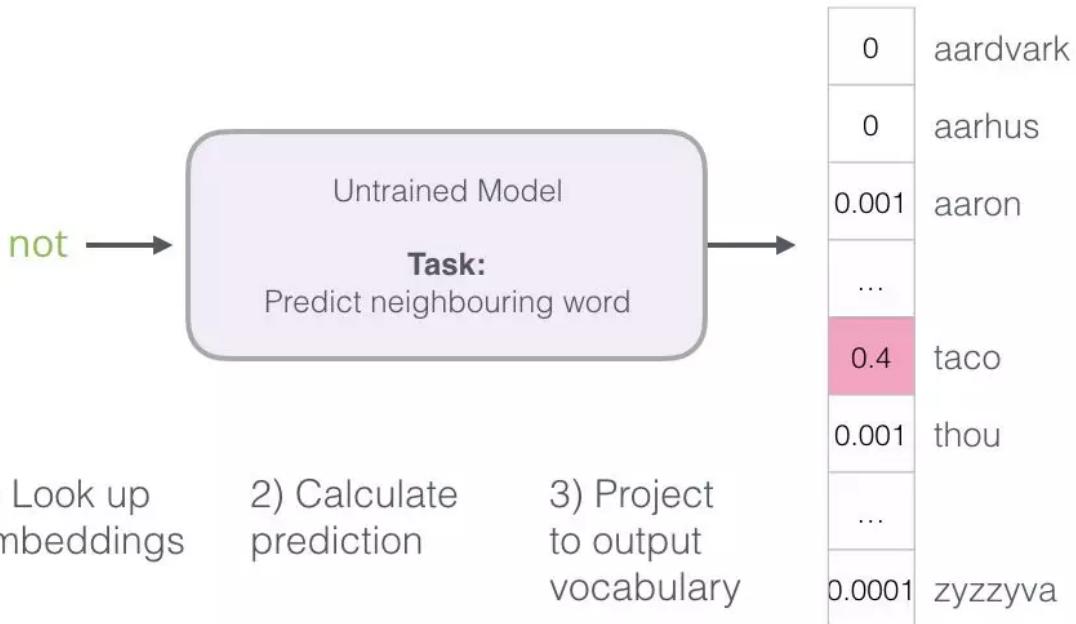
### 重新审视训练过程

现在我们已经从现有的文本中获得了Skipgram模型的训练数据集，接下来让我们看看如何使用它来训练一个能预测相邻词汇的自然语言模型。

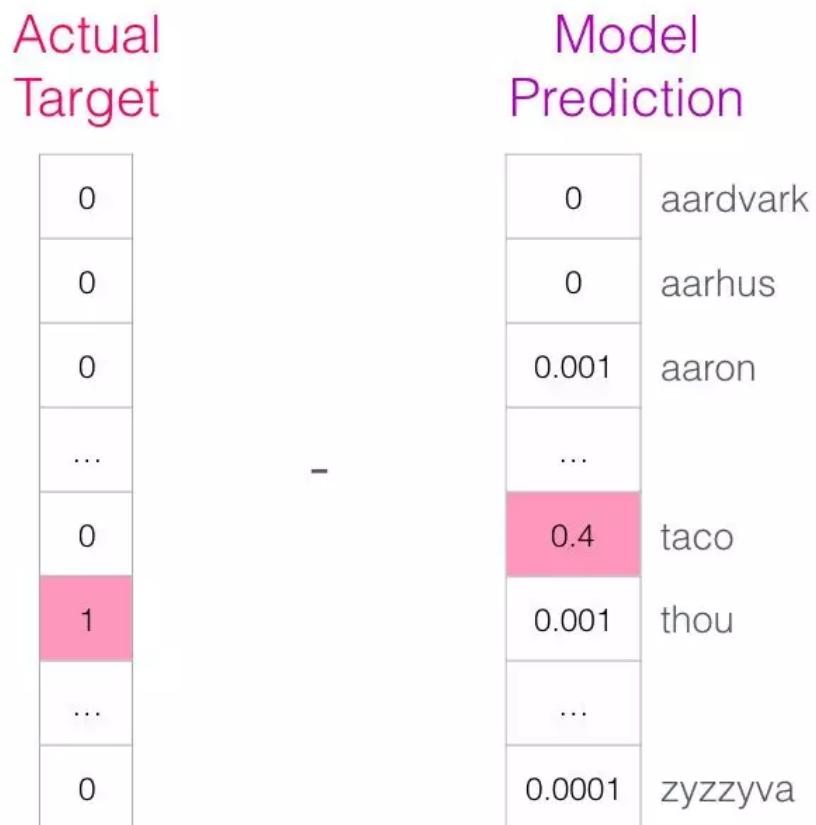
input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness



从数据集中的第一个样本开始。我们将特征输入到未经训练的模型，让它预测一个可能的相邻单词。



该模型会执行三个步骤并输入预测向量(对应于单词表中每个单词的概率)。因为模型未经训练，该阶段的预测肯定是错误的。但是没关系，我们知道应该猜出的是哪个单词——这个词就是我训练集数据中的输出标签：

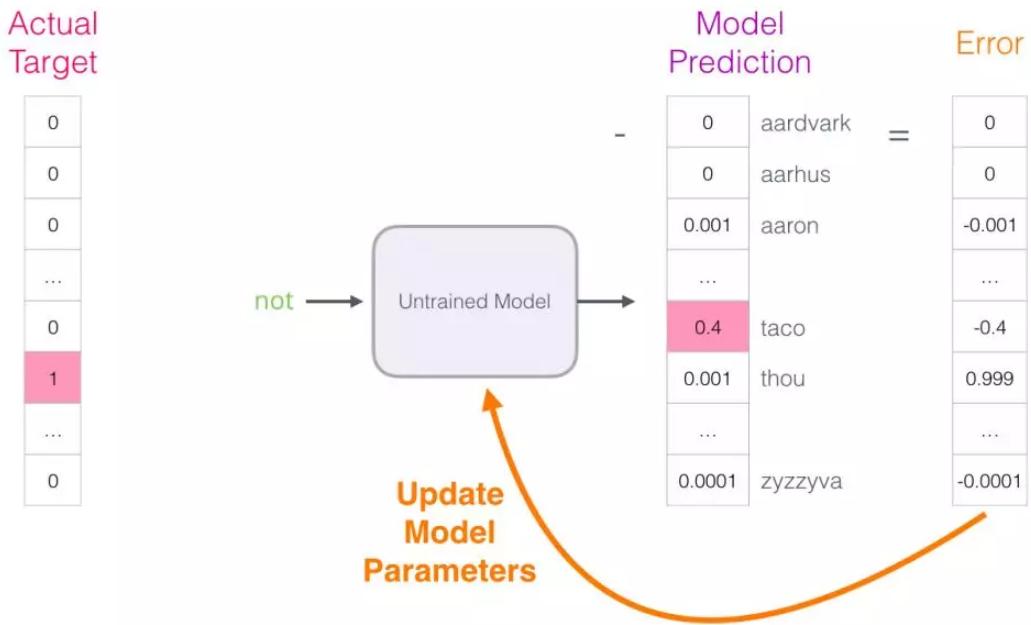


目标单词概率为1，其他所有单词概率为0，这样数值组成的向量就是“目标向量”。

模型的偏差有多少？将两个向量相减，就能得到偏差向量：

Actual Target	Model Prediction	Error
0	0 aardvark	0
0	0 aarhus	0
0	0.001 aaron	-0.001
...	...	...
0	0.4 taco	-0.4
1	0.001 thou	0.999
...	...	...
0	0.0001 zyzyva	-0.0001

现在这一误差向量可以被用于更新模型了，所以在下一轮预测中，如果用not作为输入，我们更有可能得到thou作为输出了。



这其实就是训练的第一步了。我们接下来继续对数据集内下一份样本进行同样的操作，直到我们遍历所有的样本。这就是一轮（epoch）了。我们再多做几轮（epoch），得到训练过的模型，于是就可以从中提取嵌入矩阵来用于其他应用了。

以上确实有助于我们理解整个流程，但这依然不是word2vec真正训练的方法。我们错过了一些关键的想法。

## 负例采样

回想一下这个神经语言模型计算预测值的三个步骤：



1) Look up embeddings

2) Calculate prediction

**3) Project to output vocabulary**

**[Computationally Intensive]**

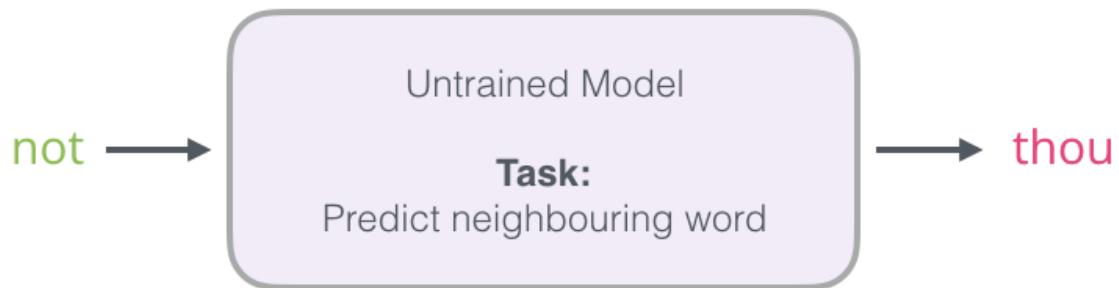
从计算的角度来看，第三步非常昂贵 - 尤其是当我们将需要在数据集中为每个训练样本都做一遍（很容易就多达数千万次）。我们需要寻找一些提高表现的方法。

一种方法是将目标分为两个步骤：

1. 生成高质量的词嵌入（不要担心下一个单词预测）。
2. 使用这些高质量的嵌入来训练语言模型（进行下一个单词预测）。

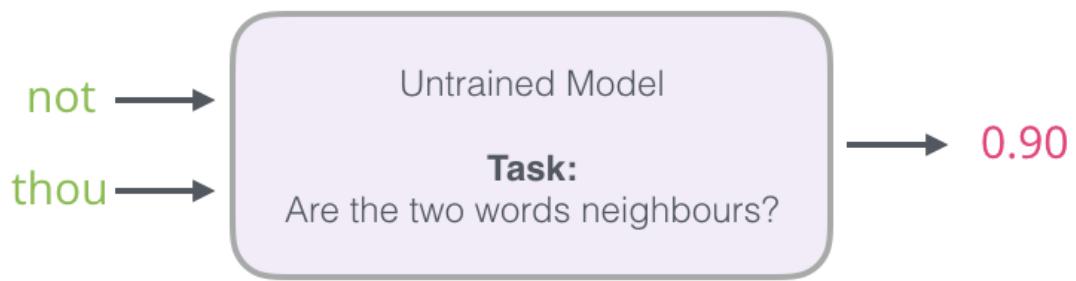
在本文中我们将专注于第1步（因为这篇文章专注于嵌入）。要使用高性能模型生成高质量嵌入，我们可以改变一下预测相邻单词这一任务：

Change Task from



将其切换到一个提取输入与输出单词的模型，并输出一个表明它们是否是邻居的分数（0表示“不是邻居”，1表示“邻居”）。

To:



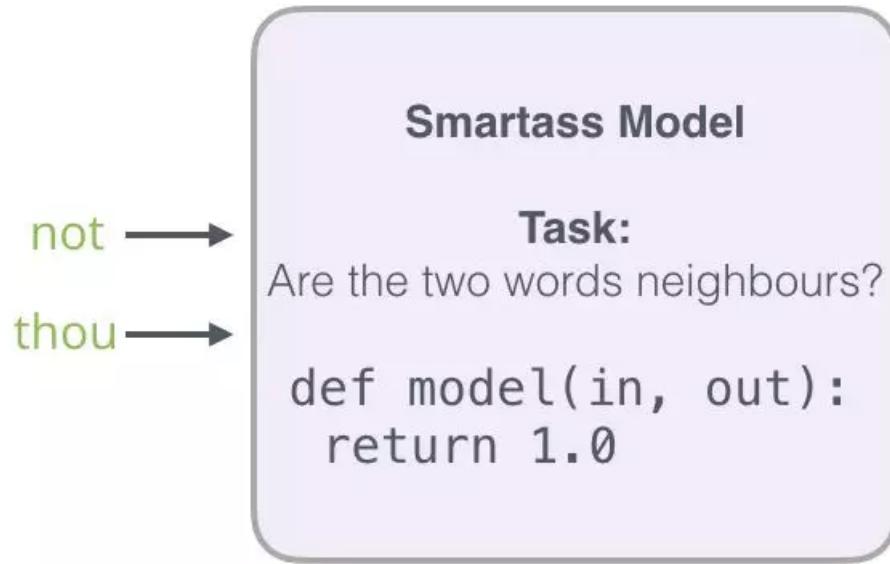
这个简单的变换将我们需要的模型从神经网络改为逻辑回归模型——因此它变得更简单，计算速度更快。

这个开关要求我们切换数据集的结构——标签值现在是一个值为0或1的新列。它们将全部为1，因为我们添加的所有单词都是邻居。

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

现在的计算速度可谓神速啦——在几分钟内就能处理数百万个例子。但是我们还需要解决一个漏洞。如果所有的例子都是邻居（目标：1），我们这个“天才模型”可能会被训练得永远返回1——准确性是百分百了，但它什么东西都学不到，只会产生垃圾嵌入结果。



为了解决这个问题，我们需要在数据集中引入负样本 - 不是邻居的单词样本。我们的模型需要为这些样本返回0。模型必须努力解决这个挑战——而且依然必须保持高速。

input word	output word	target
not	thou	1
not		0
not		0
not	shalt	1
not	make	1

>> Negative examples

对于我们数据集中的每个样本，我们添加了负面示例。它们具有相同的输入字词，标签为0。

但是我们作为输出词填写什么呢？我们从词汇表中随机抽取单词

Pick randomly from vocabulary  
(random sampling)

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

这个想法的灵感来自噪声对比估计。我们将实际信号（相邻单词的正例）与噪声（随机选择的不是邻居的单词）进行对比。这导致了计算和统计效率的巨大折衷。

### 噪声对比估计

<http://proceedings.mlr.press/v9/gutmann10a/gutmann10a.pdf>

### 基于负例采样的Skipgram (SGNS)

我们现在已经介绍了word2vec中的两个（一对）核心思想：负例采样，以及skipgram。

### Skipgram

shalt	not	make	a	machine
input		output		
make		shalt		
make		not		
make		a		
make		machine		

### Negative Sampling

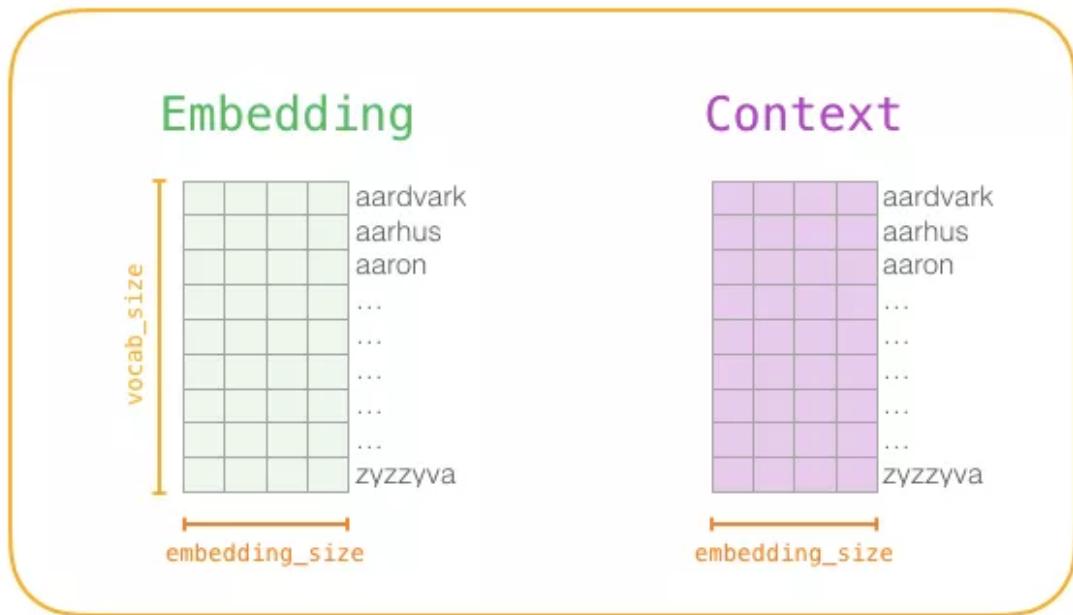
input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

### Word2vec训练流程

现在我们已经了解了skipgram和负例采样的两个中心思想，可以继续仔细研究实际的word2vec训练过程了。

在训练过程开始之前，我们预先处理我们正在训练模型的文本。在这一步中，我们确定一下词典的大小（我们称之为vocab\_size，比如说10,000）以及哪些词被它包含在内。

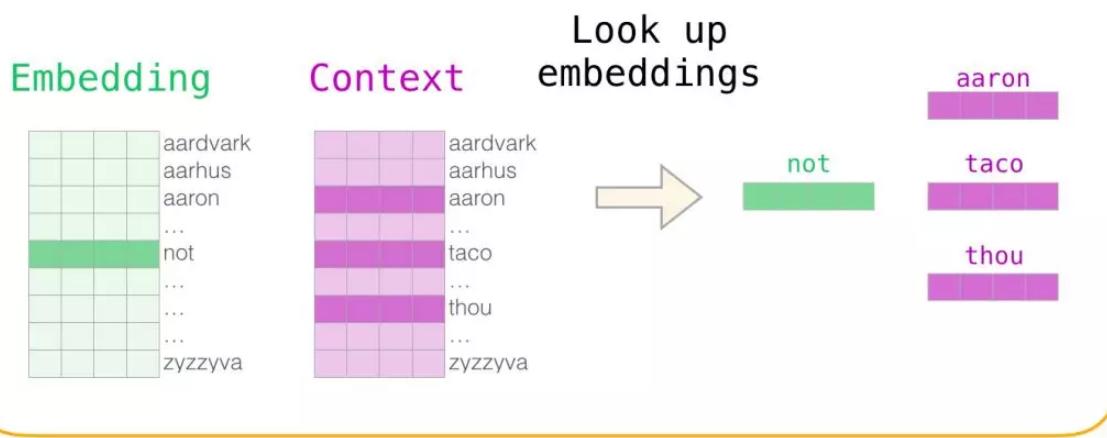
在训练阶段的开始，我们创建两个矩阵——Embedding矩阵和Context矩阵。这两个矩阵在我们的词汇表中嵌入了每个单词（所以vocab\_size是他们的维度之一）。第二个维度是我们希望每次嵌入的长度（embedding\_size——300是一个常见值，但我们在前文也看过50的例子）。



在训练过程开始时，我们用随机值初始化这些矩阵。然后我们开始训练过程。在每个训练步骤中，我们采取一个相邻的例子及其相关的非相邻例子。我们来看看我们的第一组：

dataset		model
input word	output word	target
not	thou	<b>1</b>
not	aaron	<b>0</b>
not	taco	<b>0</b>
not	shalt	<b>1</b>
not	mango	<b>0</b>
not	finglonger	<b>0</b>
not	make	<b>1</b>
not	plumbus	<b>0</b>
...	...	...

现在我们有四个单词：输入单词not和输出/上下文单词: thou (实际邻居词) , aaron和taco (负面例子)。我们继续查找它们的嵌入——对于输入词，我们查看Embedding矩阵。对于上下文单词，我们查看Context矩阵（即使两个矩阵都在我们的词汇表中嵌入了每个单词）。



然后，我们计算输入嵌入与每个上下文嵌入的点积。在每种情况下，结果都将是表示输入和上下文嵌入的相似性的数字。

input word	output word	target	input • output
not	thou	1	0.2
not	aaron	0	-1.11
not	taco	0	0.74

现在我们需要一种方法将这些分数转化为看起来像概率的东西——我们需要它们都是正值，并且处于0到1之间。sigmoid这一逻辑函数转换正适合用来做这样的事情啦。

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

现在我们可以将sigmoid操作的输出视为这些示例的模型输出。您可以看到taco得分最高，aaron最低，无论是sigmoid操作之前还是之后。

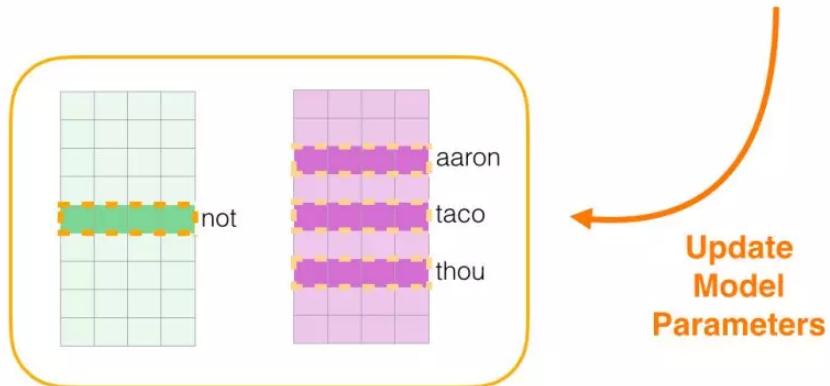
既然未经训练的模型已做出预测，而且我们确实拥有真实目标标签来作对比，那么让我们计算模型预测中的误差吧。为此我们只需从目标标签中减去sigmoid分数。

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

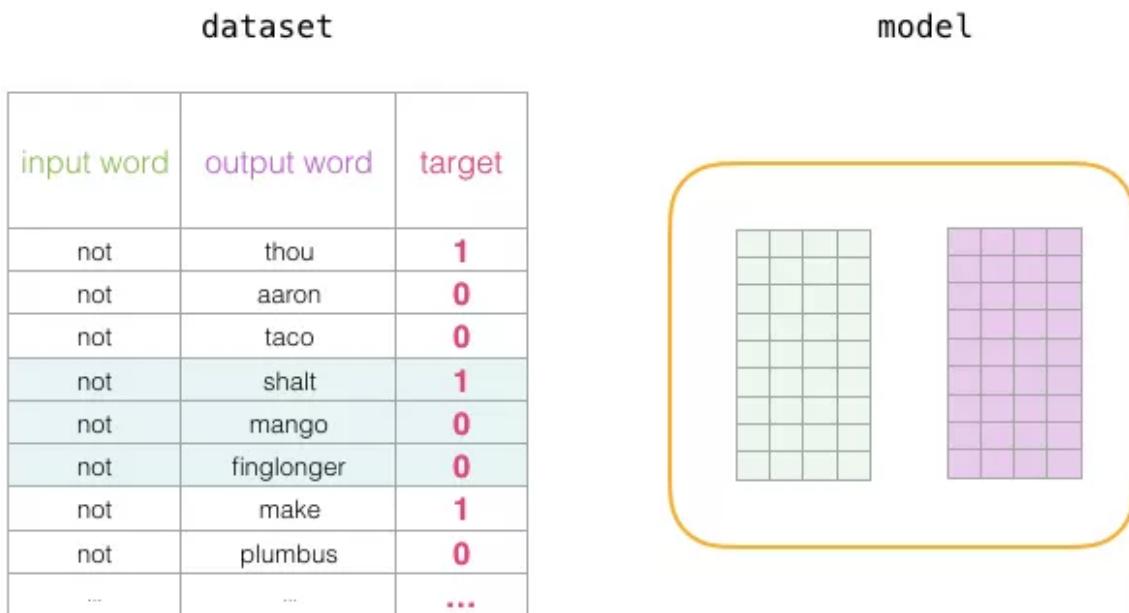
$$\text{error} = \text{target} - \text{sigmoid\_scores}$$

这是“机器学习”的“学习”部分。现在，我们可以利用这个错误分数来调整not、thou、aaron和taco的嵌入，使我们下一次做出这一计算时，结果会更接近目标分数。

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68



训练步骤到此结束。我们从中得到了这一步所使用词语更好一些的嵌入 (not, thou, aaron和taco)。我们现在进行下一步 (下一个相邻样本及其相关的非相邻样本)，并再次执行相同的过程。

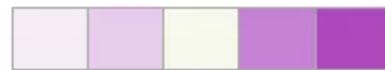


当我们循环遍历整个数据集多次时，嵌入会继续得到改进。然后我们就可以停止训练过程，丢弃Context矩阵，并使用Embeddings矩阵作为下一项任务的已被训练好的嵌入。

### 窗口大小和负样本数量

word2vec训练过程中的两个关键超参数是窗口大小和负样本的数量。

Window size: 5



Window size: 15



不同的任务适合不同的窗口大小。一种启发式方法是，**使用较小的窗口大小 (2-15)** 会得到这样的嵌入：两个嵌入之间的高相似性得分表明这些单词是可互换的（注意，如果我们只查看附近距离很近的单词，反义词通常可以互换——例如，好的和坏的经常出现在类似的语境中）。**使用较大的窗口大小 (15-50, 甚至更多)** 会得到相似性更能指示单词相关性的嵌入。在实际操作中，你通常需要对嵌入过程提供指导以帮助读者得到相似的“语感”。Gensim默认窗口大小为5（除了输入字本身以外还包括输入字之前与之后的两个字）。

Negative samples: 2

input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

Negative samples: 5

input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0
make	finglonger	0
make	plumbus	0
make	mango	0

负样本的数量是训练训练过程的另一个因素。原始论文认为5-20个负样本是比较理想的数量。它还指出，当你拥有足够大的数据集时，2-5个似乎就已经足够了。Gensim默认为5个负样本。

## BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

采用的是双向的Transformer的Encoder结构

通过预训练模型可以显著提高NLP的下游任务

现有的模型都是单向的语言模型，无法充分了解单词所在的上下文结构。

BERT受完型填空的启发，使用随机屏蔽一些单词。然后让模型根据上下文来预测被遮挡的单词。

BERT是真正的结合上下文进行训练，而ELMo只是左右分别训练

还提出了一种，下一个句子预测的任务预训练文本对,将token级提升到句子级以应用不同种类的下游任务

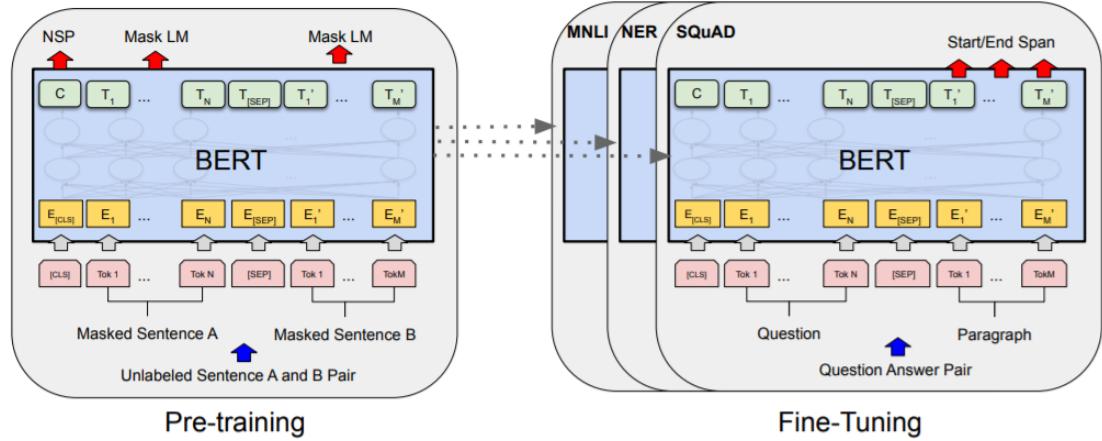
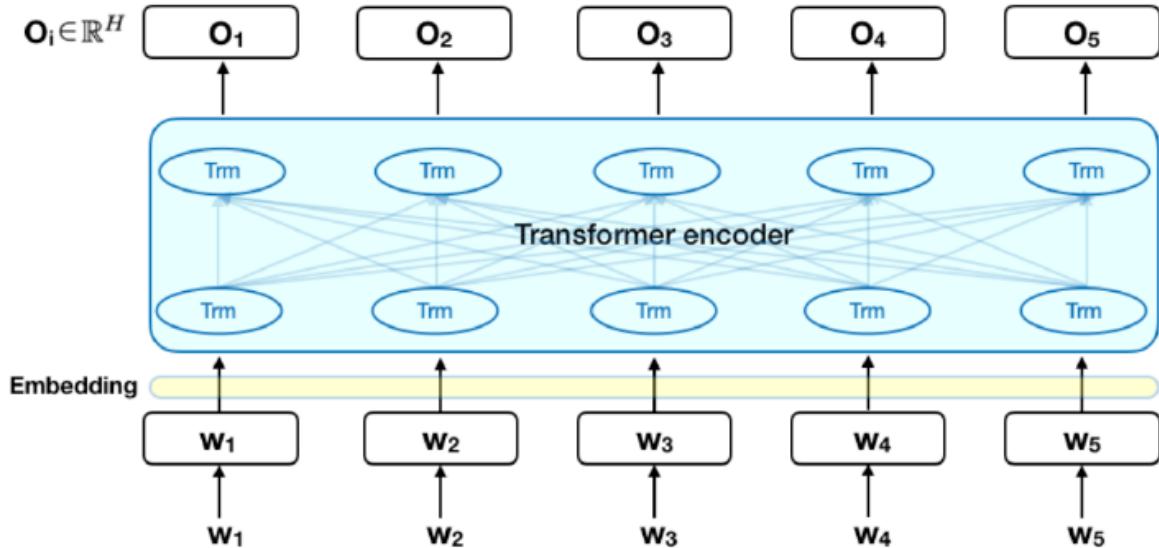


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

### Pre-training在未标记的数据上进行无监督学习

Fine-tuning阶段,BERT首先利用预训练得到的参数初始化模型,然后利用下游任务标记好的数据进行有监督学习,并对所有参数进行微调.所有下游任务都有单独的Fine-tuning模型,即使是使用相同预训练参数



BERT是双向的Transformer Encoder

### 输入/输出

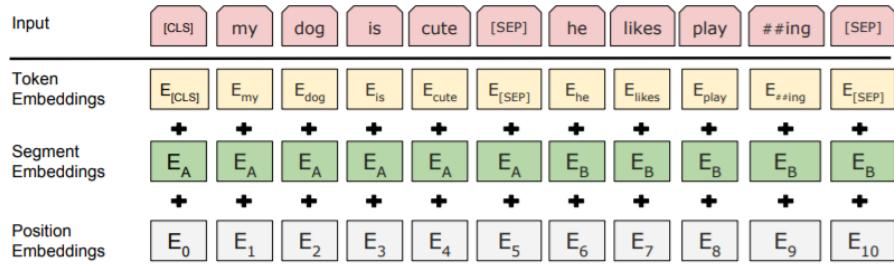


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

**Token Embeddings** 采用的 **WordPiece Embedding**, 共有 30000 个 token。每个 sequence 会以一个特殊的classification token [CLS] 开始, 同时这也会作为分类任务的输出; 句子间会以 special token [SEP] 进行分割。 **WordPiece Embedding**: n -gram 字符级 Embedding, 采用 BPE 双字节编码, 可以将单词拆分, 比如"loved" "loving" "loves" 会拆分成"lov", "ed", "ing", "es".

**Segment Embedding** 也可以用来分割句子, 但主要用来区分句子对。Embedding A 和 Embedding B 分别代表左右句子, 如果是普通的句子就直接用 Embedding A

**Position Embedding** 是用来给单词定位的, 直接使用 one-hot 编码。BERT 最终的 input 是三种不同的 Embedding 直接相加.

## 预训练

BERT 采用两种非监督任务来进行预训练, 一个是 token-level 级别的 Masked LM, 一个是 sentence-level 级别的Next Sentence Prediction

两个任务同时训练, 所以BERT的损失函数是两个任务的损失函数想加

### Masked LM

双向会导致数据泄露的问题, 即模型可以间接看到要预测的单词

Masked LM: 随机屏蔽一些token并通过上下文预测这些token, 在实验过程中, BERT会随机屏蔽每个序列中的15%的token并用[MASK]来代替

但这样会带来一个新的问题: [MASK] token 不会出现在下游任务中。为了缓解这种情况, 谷歌的同学采用以下三种方式来代替 [MASK] token:

- 80% 的 [MASK] token 会继续保持 [MASK];
- 10% 的 [MASK] token 会被随机的一个单词取代;
- 10% 的 [MASK] token 会保持原单词不变 (但是还是要预测)

最终 Masked ML 的损失函数是只由被 [MASK] 的部分来计算。

这样做的目的主要是为了告诉模型 [MASK] 是噪音, 以此来忽略标记的影响。

### Next Sentence Prediction

为了解决这个问题, 谷歌的同学训练了一个 sentence-level 的分类任务。具体来说, 假设有 AB 两个句对, 在训练过程 50% 的训练样本 A 下句接的是 B 作为正例; 而剩下 50% 的训练样本 A 下句接的是随机一个句子作为负例。并且通过 classification token 连接 Softmax 输出最后的预测概率。

**Input** =[CLS] the man went to [MASK] store [SEP]  
he bought a gallon [MASK] milk [SEP]

**Label** = IsNext

**Input** =(CLS) the man [MASK] to the store [SEP] penguin [MASK] are flight #less birds [SEP]  
**Label** = NotNext

# Fine-tuning

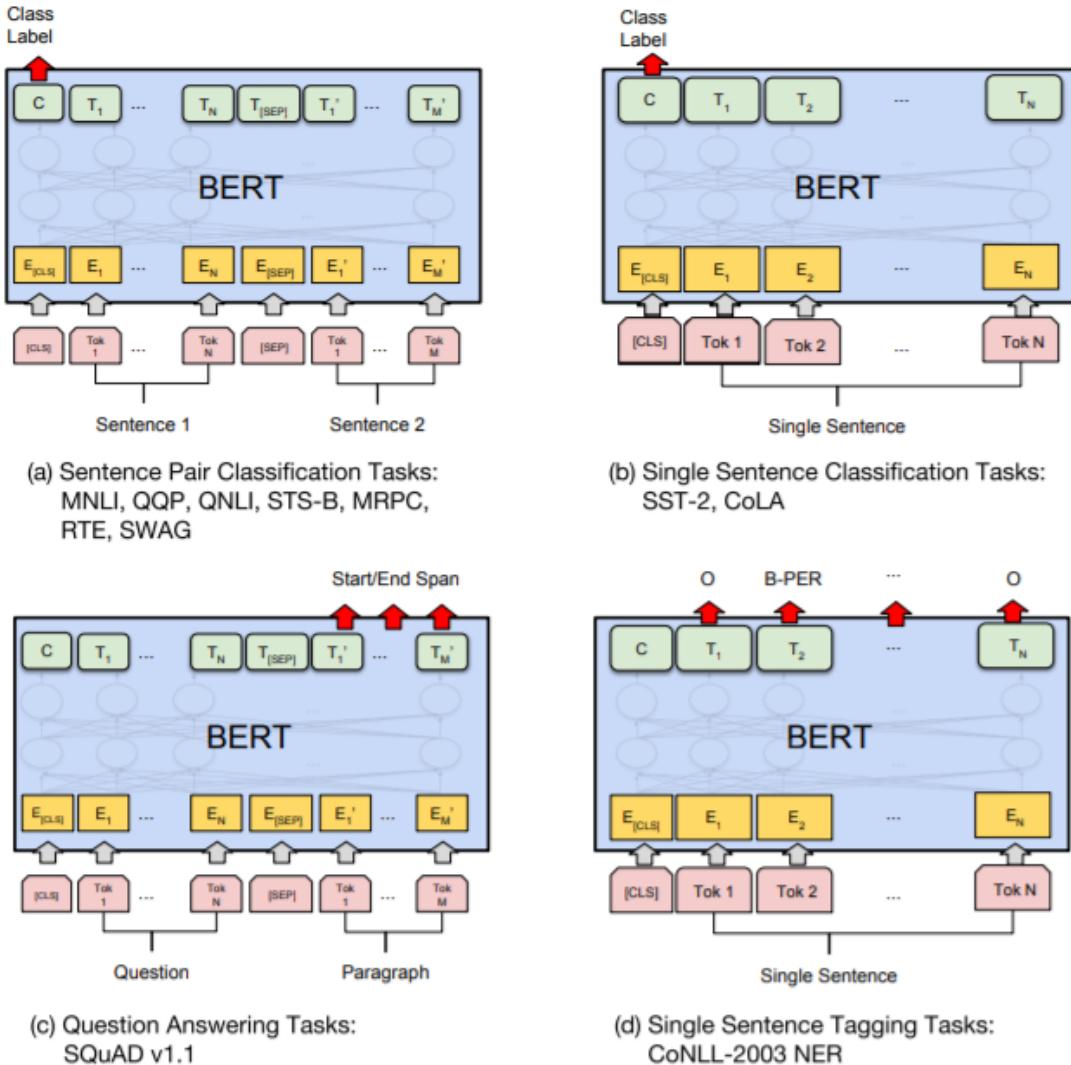


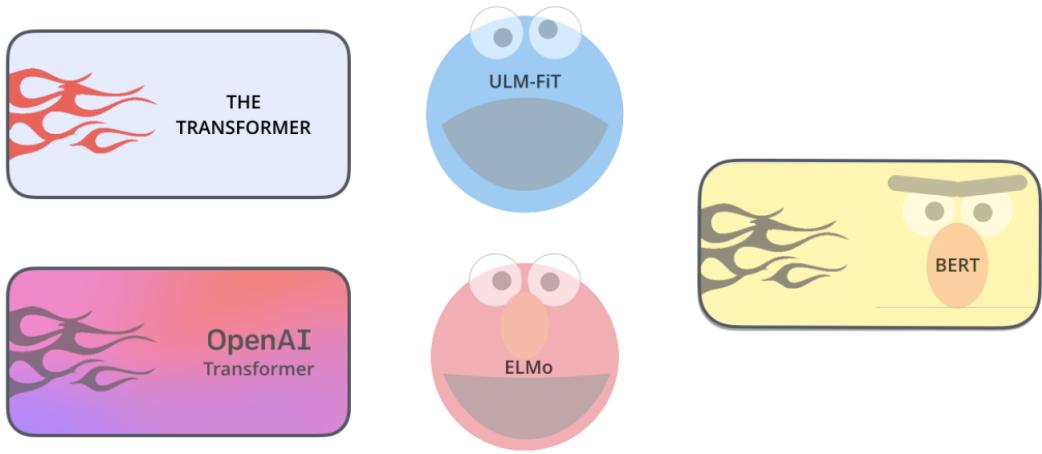
Figure 4: Illustrations of Fine-tuning BERT on Different Tasks.

预训练的Transformer已经完成了句子和句子对的表示学习,针对不同的下游任务,可以将具体的输入和输出适配到BERT中,并采用端到端的训练去微调模型参数

- a,b是句子级的任务,类似句子分类,情感分析等,输入句子或者句子对,在[CLS]位置接入Softmax输出Label
- c是token级别的任务,比如QA问题,输入问题和段落,在Paragraph对应输出的hidden vector后接上两个softmax,分别训练出Span的Start index和End index作为Question答案
- d是token级的任务,实体命名,接上softmax层可以输出具体的分类

## Jay Alammar文章

2018年可谓是自然语言处理（NLP）的元年，在我们如何以最能捕捉潜在语义关系的方式 来辅助计算机对的句子概念性的理解 这方面取得了极大的发展进步。此外， NLP领域的一些开源社区已经发布了很多强大的组件， 我们可以在自己的模型训练过程中免费的下载使用。（可以说今年是NLP的ImageNet时刻，因为这和几年前计算机视觉的发展很相似）

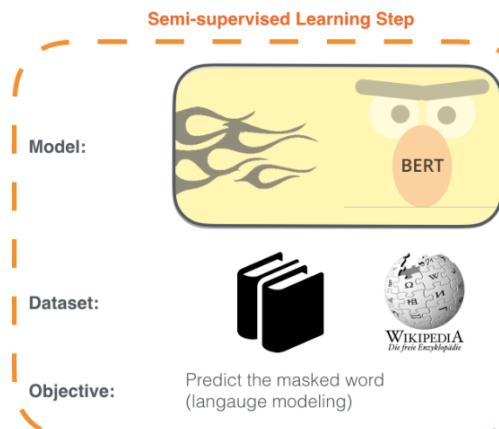


[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

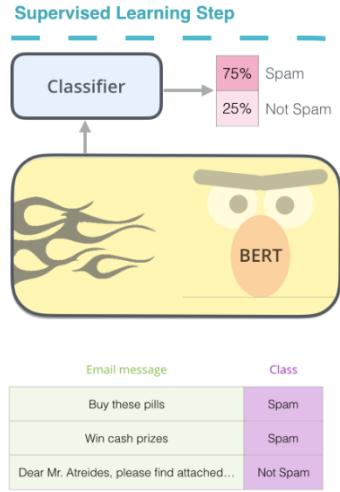
上图中，最新发布的BERT是一个NLP任务的里程碑式模型，它的发布势必会带来一个NLP的新时代。BERT是一个算法模型，它的出现打破了大量的自然语言处理任务的记录。在BERT的论文发布不久后，Google的研发团队还开放了该模型的代码，并提供了一些在大量数据集上预训练好的算法模型下载方式。Goole开源这个模型，并提供预训练好的模型，这使得所有人都可以通过它来构建一个涉及NLP的算法模型，节约了大量训练语言模型所需的时间，精力，知识和资源。

#### 1 - **Semi-supervised** training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.



#### 2 - **Supervised** training on a specific task with a labeled dataset.



The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

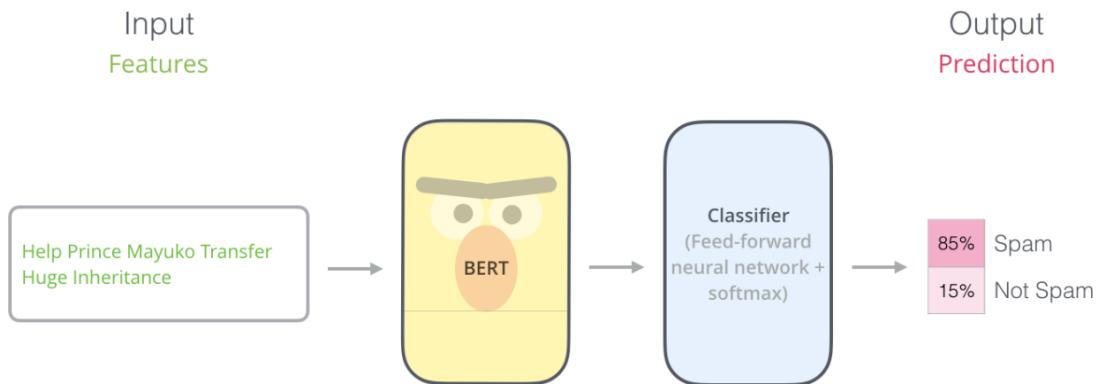
[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

BERT集成了最近一段时间内NLP领域中的一些顶尖的思想，包括但不限于 [Semi-supervised Sequence Learning](#) (by [Andrew Dai](#) and [Quoc Le](#)), [ELMo](#) (by [Matthew Peters](#) and researchers from [AI2](#) and [UW CSE](#)), [ULMFiT](#) (by fast.ai founder [Jeremy Howard](#) and [Sebastian Ruder](#)), and the [OpenAI transformer](#) (by OpenAI researchers [Radford](#), [Narasimhan](#), [Salimans](#), and [Sutskever](#)), and the Transformer ([Vaswani et al.](#))。

你需要注意一些事情才能恰当的理解BERT的内容，不过，在介绍模型涉及的概念之前可以使用BERT的方法。

## 示例：句子分类

使用BERT最简单的方法就是做一个文本分类模型，这样的模型结构如下图所示：



[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

为了训练一个这样的模型，（主要是训练一个分类器），在训练阶段BERT模型发生的变化很小。该训练过程称为微调，并且源于 [Semi-supervised Sequence Learning](#) 和 ULMFiT.。

为了更方便理解，我们下面举一个分类器的例子。分类器是属于监督学习领域的，这意味着你需要一些标记的数据来训练这些模型。对于垃圾邮件分类器的示例，标记的数据集由邮件的内容和邮件的类别2部分组成（类别分为“垃圾邮件”或“非垃圾邮件”）。

Email message	Class
Buy these pills	Spam
Win cash prizes	Spam
Dear Mr. Atreides, please find attached...	Not Spam

[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

## 模型架构

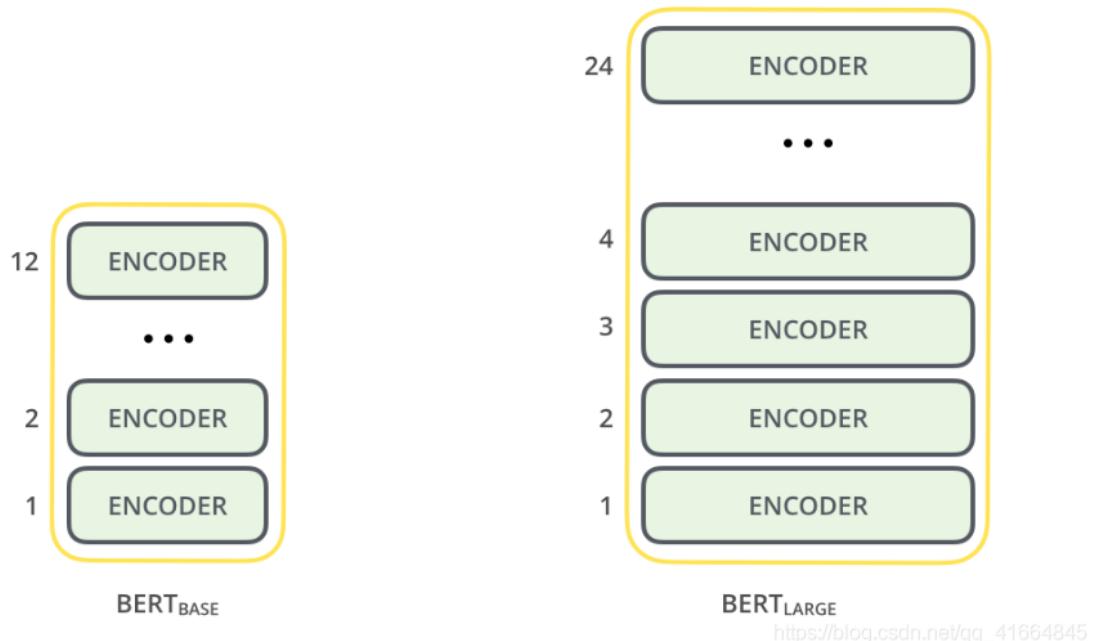
现在您已经了解了如何使用BERT的示例，让我们仔细了解一下他的工作原理。



BERT的论文中介绍了2种版本：

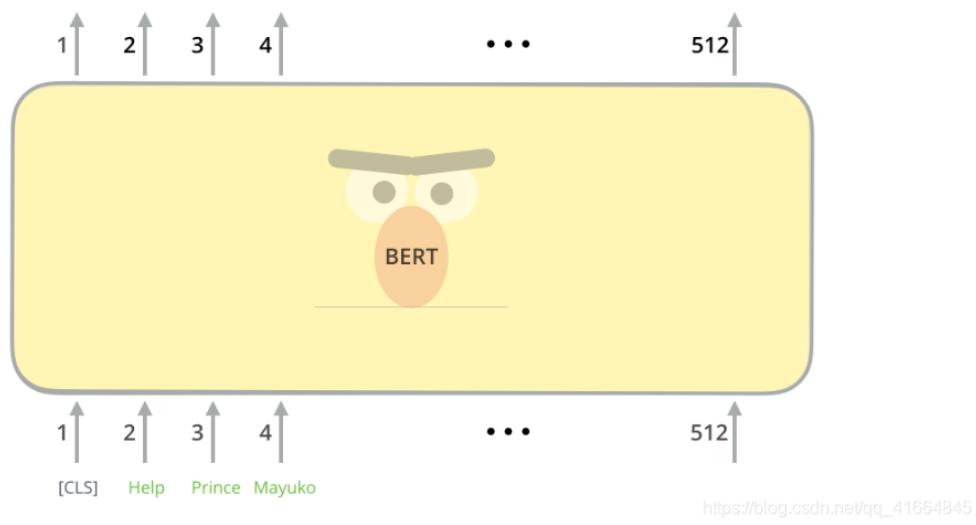
- BERT BASE - 与OpenAI Transformer的尺寸相当，以便比较性能
- BERT LARGE - 一个非常庞大的模型，它完成了本文介绍的最先进的结果。

BERT的基础集成单元是Transformer的Encoder。关于Transformer的介绍可以阅读作者之前的文章：[The Illustrated Transformer](#)，该文章解释了Transformer模型 - BERT的基本概念以及我们接下来要讨论的概念。



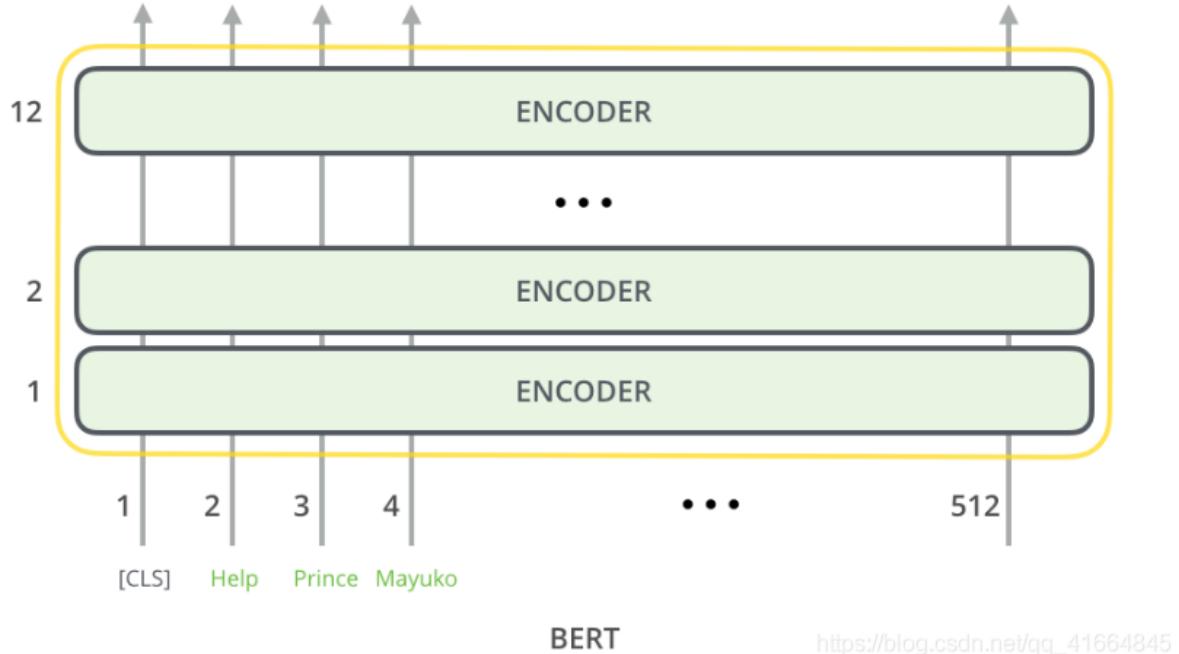
2个BERT的模型都有一个很大的编码器层数，（论文里面将此称为Transformer Blocks） - 基础版本就有12层，进阶版本有24层。同时它也有很大的前馈神经网络（768和1024个隐藏层神经元），还有很多attention heads（12-16个）。这超过了Transformer论文中的参考配置参数（6个编码器层，512个隐藏层单元，和8个注意头）

## 模型输入



输入的第一个字符为[CLS]，在这里字符[CLS]表达的意思很简单 - Classification（分类）。

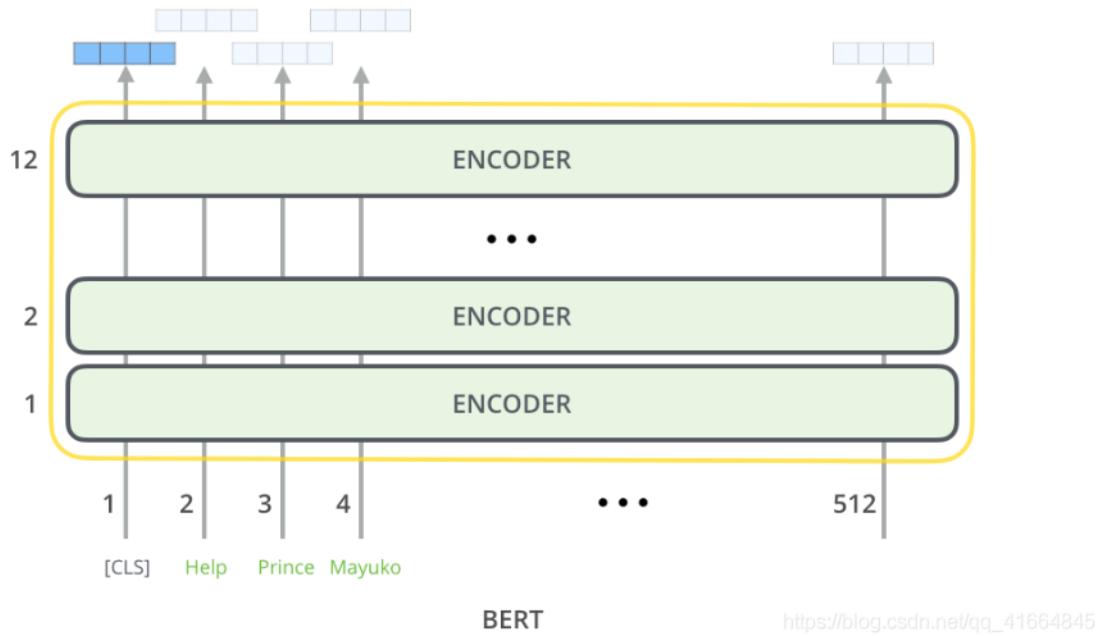
BERT与Transformer的编码方式一样。将固定长度的字符串作为输入，数据由下而上传递计算，每一层都用到了self attention，并通过前馈神经网络传递其结果，将其交给下一个编码器。



这样的架构，似乎是沿用了Transformer的架构（除了层数，不过这是我们可以设置的参数）。那么BERT与Transformer不同之处在哪里呢？可能在模型的输出上，我们可以发现一些端倪。

## 模型输出

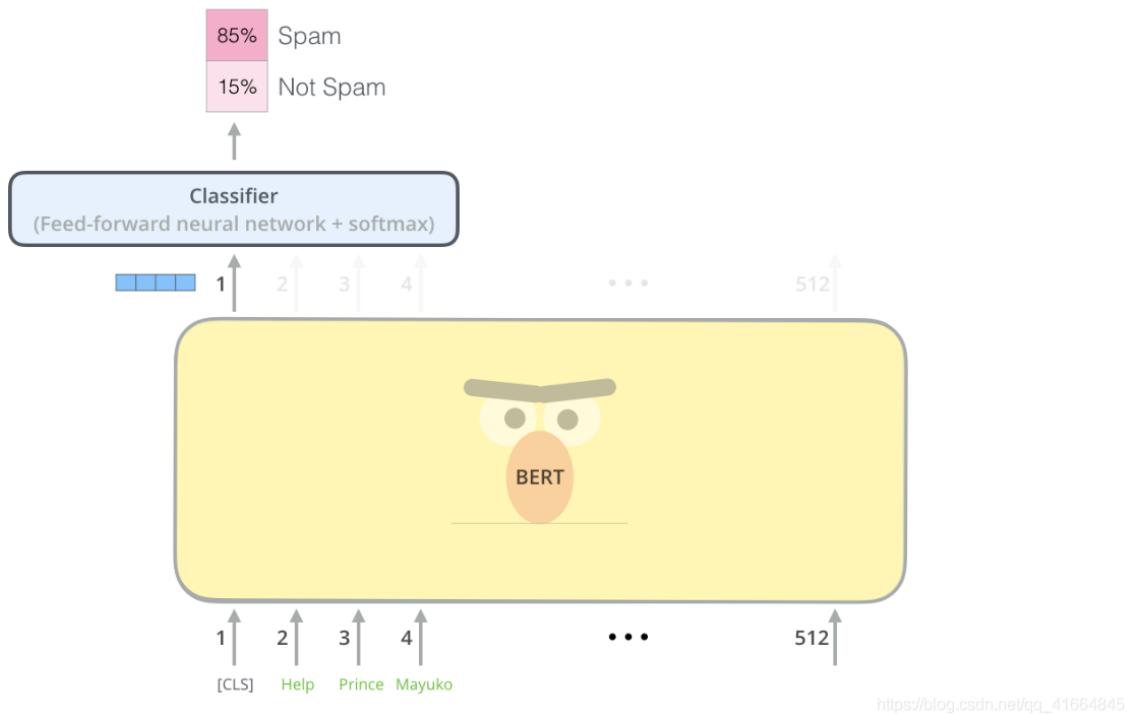
每个位置返回的输出都是一个隐藏层大小的向量（基本版本BERT为768）。以文本分类为例，我们重点关注第一个位置上的输出（第一个位置是分类标识[CLS]）。如下图



BERT

[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

该向量现在可以用作我们选择的分类器的输入，在论文中指出使用单层神经网络作为分类器就可以取得很好的效果。原理如下。：

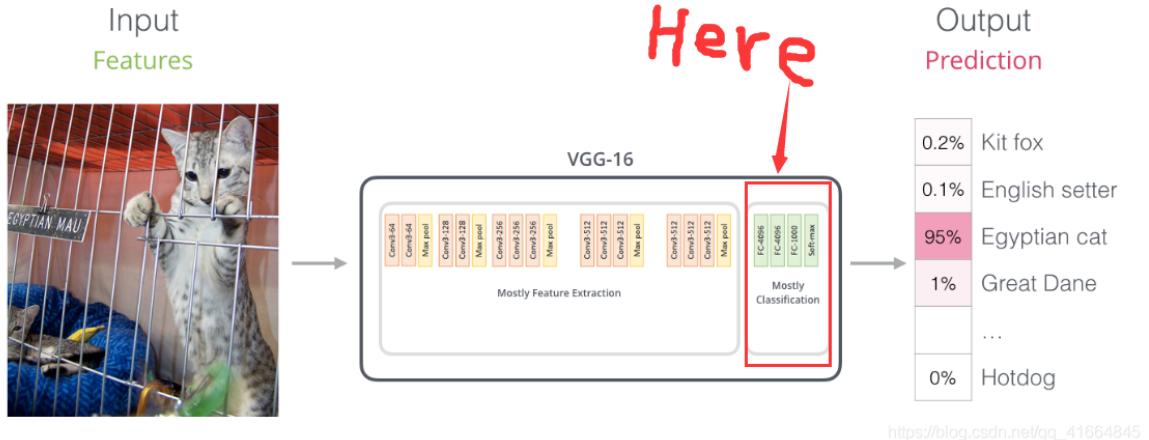


[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

例子中只有垃圾邮件和非垃圾邮件，如果你有更多的label，你只需要增加输出神经元的个数即可，另外把最后的激活函数换成softmax即可。

## Parallels with Convolutional Nets (BERT VS 卷积神经网络)

对于那些具有计算机视觉背景的人来说，这个矢量切换应该让人联想到VGGNet等网络的卷积部分与网络末端的完全连接的分类部分之间发生的事情。你可以这样理解，实质上这样理解也很方便。



## 词嵌入的新时代~

BERT的开源随之而来的是一种词嵌入的更新。到目前为止，词嵌入已经成为NLP模型处理自然语言的主要组成部分。诸如Word2vec和Glove 等方法已经广泛的用于处理这些问题，在我们使用新的词嵌入之前，我们有必要回顾一下其发展。

## Word Embedding Recap

为了让机器可以学习到文本的特征属性，我们需要一些将文本数值化的表示的方式。Word2vec算法通过使用一组固定维度的向量来表示单词，计算其方式可以捕获到单词的语义及单词与单词之间的关系。使用Word2vec的向量化表示方式可以用于判断单词是否相似，对立，或者说判断“男人”与“女人”的关系就如同“国王”与“王后”。（这些话是不是听腻了~ emmm水文必备）。另外还能捕获到一些语法的关系，这个在英语中很实用。例如“had”与“has”的关系如同“was”与“is”的关系。

这样的做法，我们可以使用大量的文本数据来预训练一个词嵌入模型，而这个词嵌入模型可以广泛用于其他NLP的任务，这是个好主意，这使得一些初创公司或者计算资源不足的公司，也能通过下载已经开源的词嵌入模型来完成NLP的任务。

更多关于词嵌入的信息，可以阅读我的之前文章：[什么是文本的词嵌入？](#)

## ELMo：语境问题

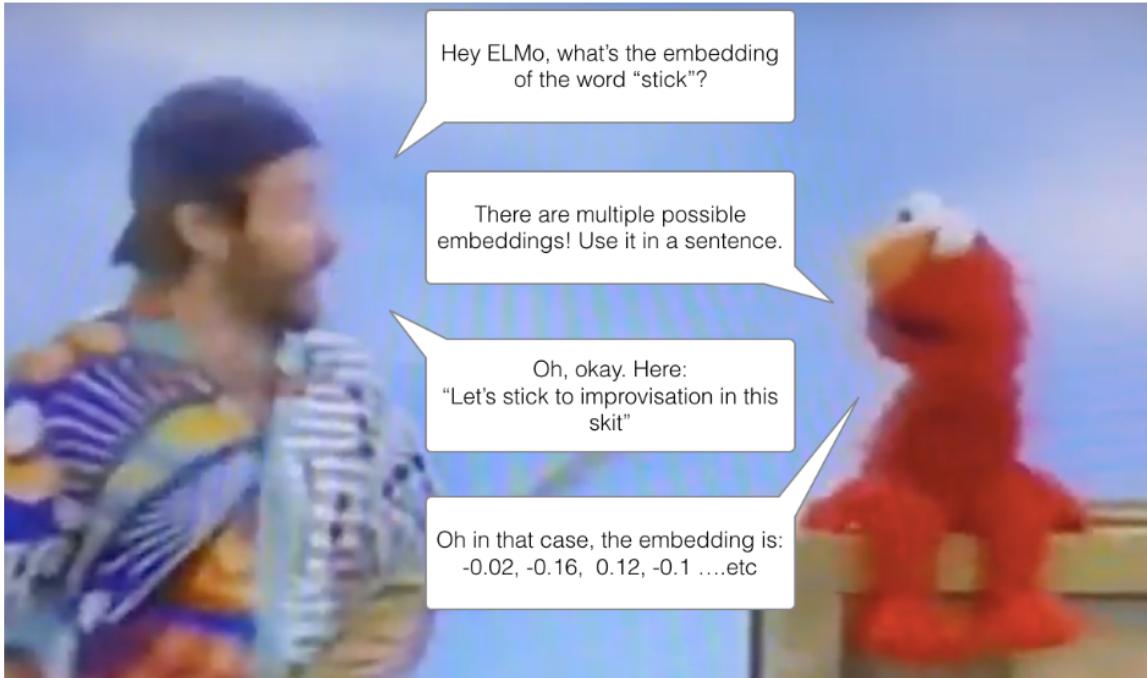
上面介绍的词嵌入方式有一个很明显的问题，因为使用预训练好的词向量模型，那么无论上下文的语境关系如何，每个单词都只有一个唯一的且已经固定保存的向量化形式”。Wait a minute “ - 出自([Peters et. al., 2017](#), [McCann et. al., 2017](#), and yet again [Peters et. al., 2018 in the ELMo paper](#))

“ Wait a minute ”这是一个欧美日常梗，示例：

我：兄弟，你认真学习深度，没准能拿80W年薪啊。

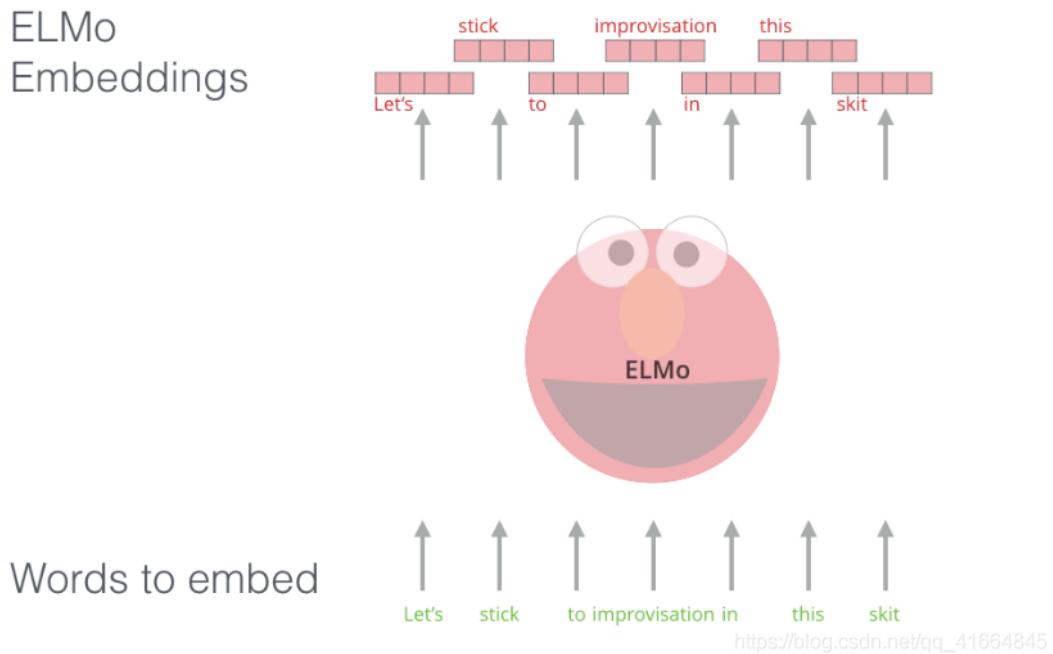
你：Wait a minute，这么好，你为啥不做。

这和中文的同音字其实也类似，用这个举一个例子吧，‘长’这个字，在‘长度’这个词中表示度量，在‘长高’这个词中表示增加。那么为什么我们不通过“长”周围是度或者是高来判断它的读音或者它的语义呢？嗖嘎，这个问题就派生出语境化的词嵌入模型。



Contextualized word-embeddings can give words different embeddings based on the meaning they carry in the context of the sentence.  
Also, RIP Robin Williams  
[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

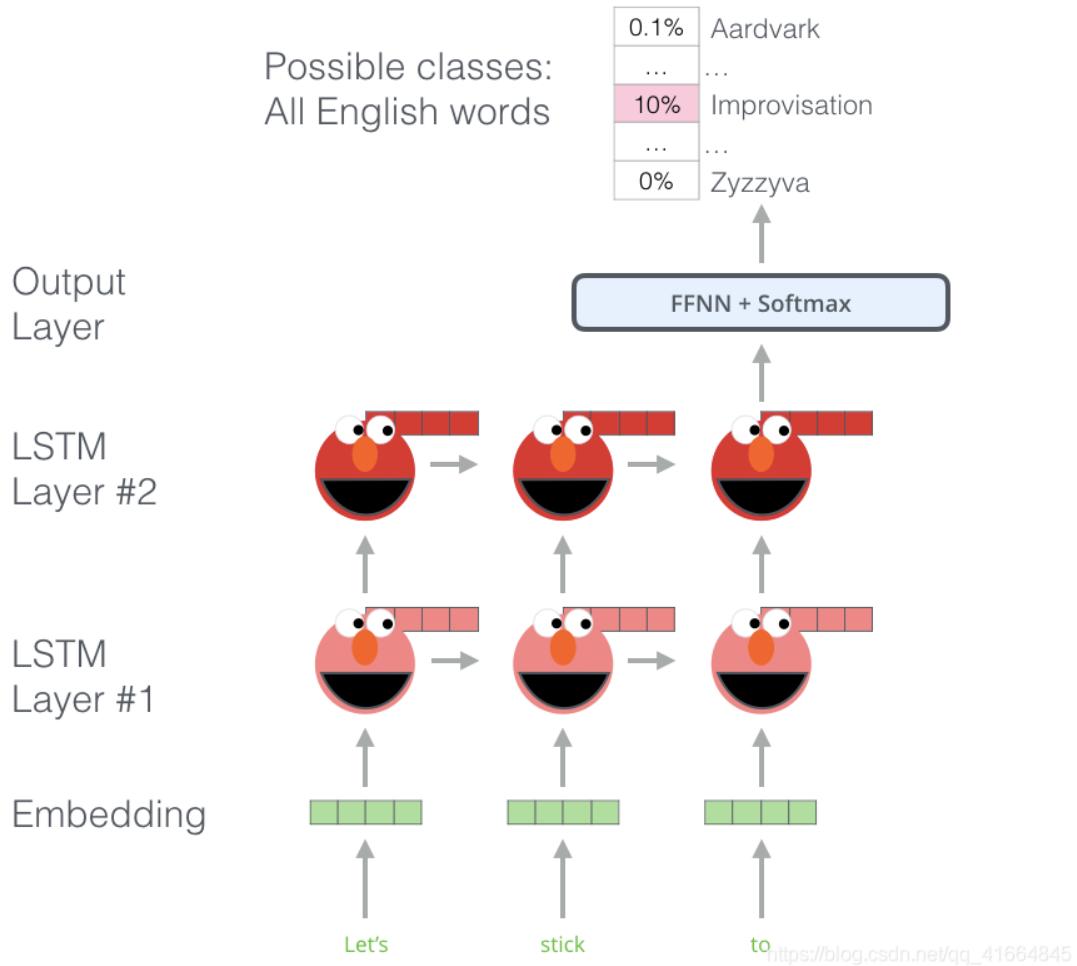
ELMo改变Word2vec类的将单词固定为指定长度的向量的处理方式，它是在为每个单词分配词向量之前先查看整个句子，然后使用bi-LSTM来训练它对应的词向量。



ELMo为解决NLP的语境问题作出了重要的贡献，它的LSTM可以使用与我们任务相关的大量文本数据来进行训练，然后将训练好的模型用作其他NLP任务的词向量的基准。

ELMo的秘密是什么？

ELMo会训练一个模型，这个模型接受一个句子或者单词的输入，输出最有可能出现在后面的一个单词。想想输入法，对啦，就是这样的道理。这个在NLP中我们也称作Language Modeling。这样的模型很容易实现，因为我们拥有大量的文本数据且我们可以在不需要标签的情况下学习。



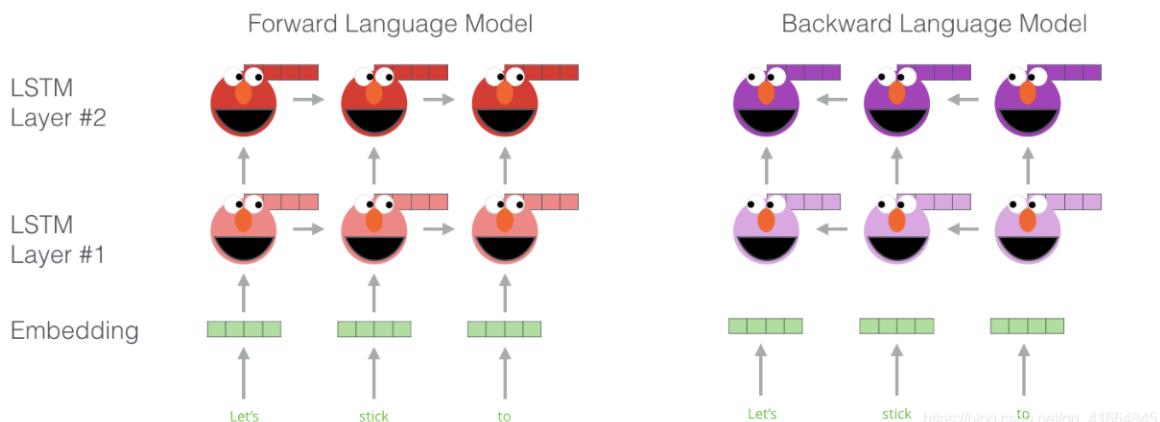
上图介绍了ELMo预训练的过程的步骤的一部分：

我们需要完成一个这样的任务：输入“Let's stick to”，预测下一个最可能出现的单词，如果在训练阶段使用大量的数据集进行训练，那么在预测阶段我们可能准确的预测出我们期待的下一个单词。比如输入“机器”，在“学习”和“买菜”中它最有可能的输出会是‘学习’而不是‘买菜’。

从上图可以发现，每个展开的LSTM都在最后一步完成预测。

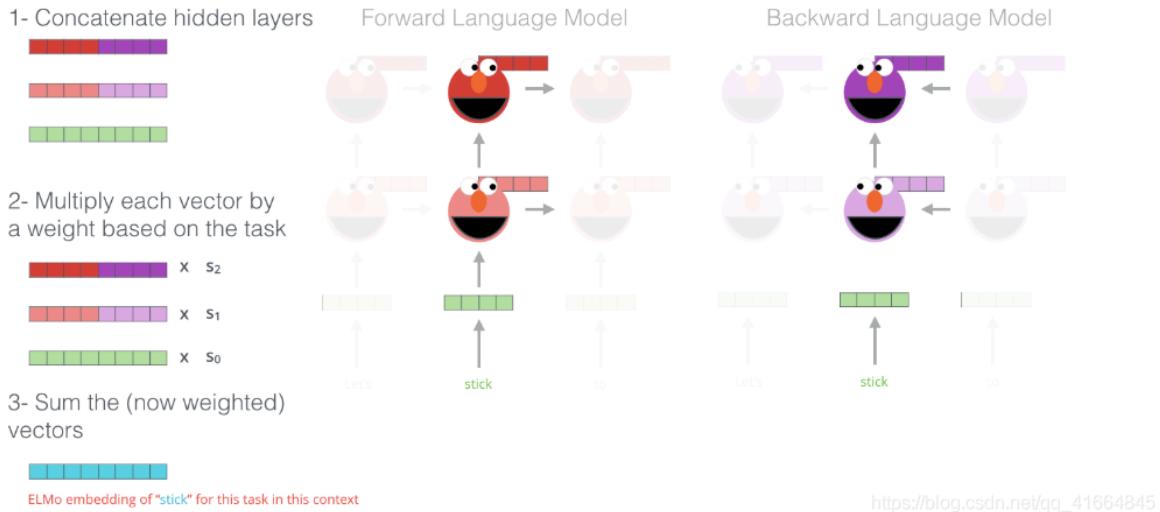
对了真正的ELMo会更进一步，它不仅能判断下一个词，还能预测前一个词。（Bi-Lstm）

Embedding of “stick” in “Let's stick to” - Step #1



ELMo通过下图的方式将hidden states（的初始的嵌入）组合起来提炼出具有语境意义的词嵌入方式（全连接后加权求和）

### Embedding of "stick" in "Let's stick to" - Step #2



## ULM-FiT: NLP领域应用迁移学习

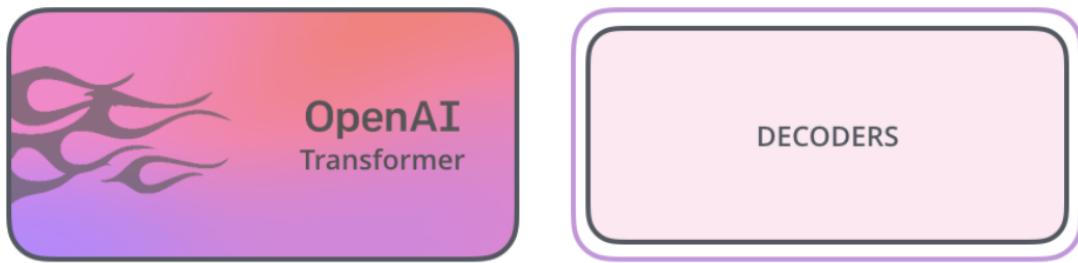
ULM-FiT机制让模型的预训练参数得到更好的利用。所利用的参数不仅限于embeddings，也不仅限于语境embedding，ULM-FiT引入了Language Model和一个有效微调该Language Model来执行各种NLP任务的流程。这使得NLP任务也能像计算机视觉一样方便的使用迁移学习。

## The Transformer: 超越LSTM的结构

Transformer论文和代码的发布，以及其在机器翻译等任务上取得的优异成果，让一些研究人员认为它是LSTM的替代品，事实上却是Transformer比LSTM更好的处理long-term dependancies（长程依赖）问题。Transformer Encoding和Decoding的结构非常适合机器翻译，但是怎么利用他来做文本分类的任务呢？实际上你只用使用它来预训练可以针对其他任务微调的语言模型即可。

## OpenAI Transformer: 用于语言模型的Transformer解码器预训练

事实证明，我们并不需要一个完整的transformer结构来使用迁移学习和一个很好的语言模型来处理NLP任务。我们只需要Transformer的解码器就行了。The decoder is a good choice because it's a natural choice for language modeling (predicting the next word) since it's built to mask future tokens – a valuable feature when it's generating a translation word by word.



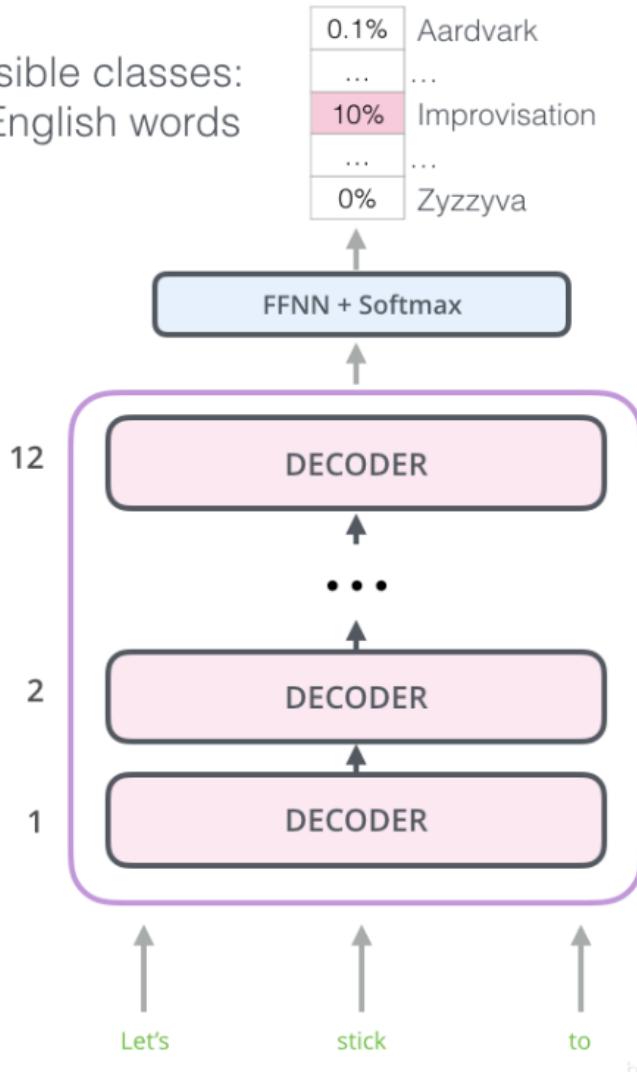
The OpenAI Transformer is made up of the decoder stack from the Transformer

[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

该模型堆叠了十二个Decoder层。由于在该设置中没有Encoder，因此这些Decoder将不具有Transformer Decoder层具有的Encoder - Decoder attention层。然而，取而代之的是一个self attention层 (masked so it doesn't peak at future tokens)。

通过这种结构调整，我们可以继续在相似的语言模型任务上训练模型：使用大量的未标记数据集训练，来预测下一个单词。举个例子：你那7000本书喂给你的模型，（书籍是极好的训练样本~比博客和推文好很多。）训练框架如下：

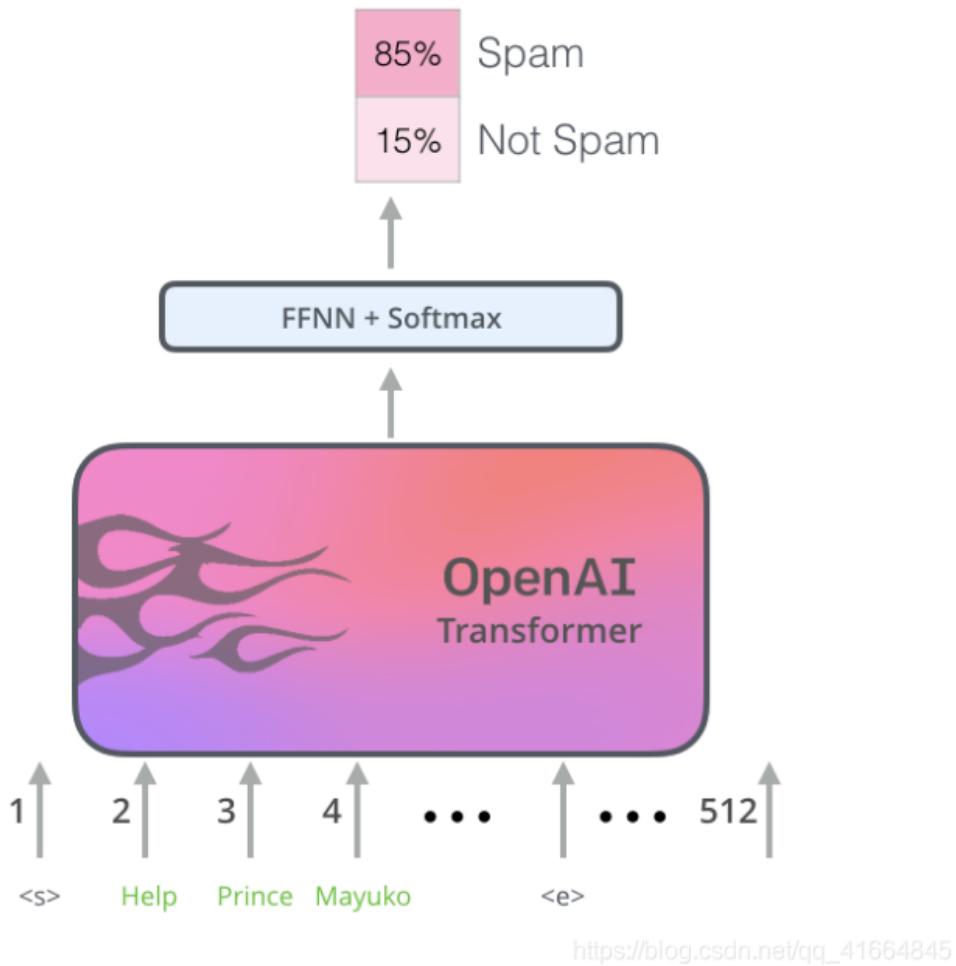
Possible classes:  
All English words



[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

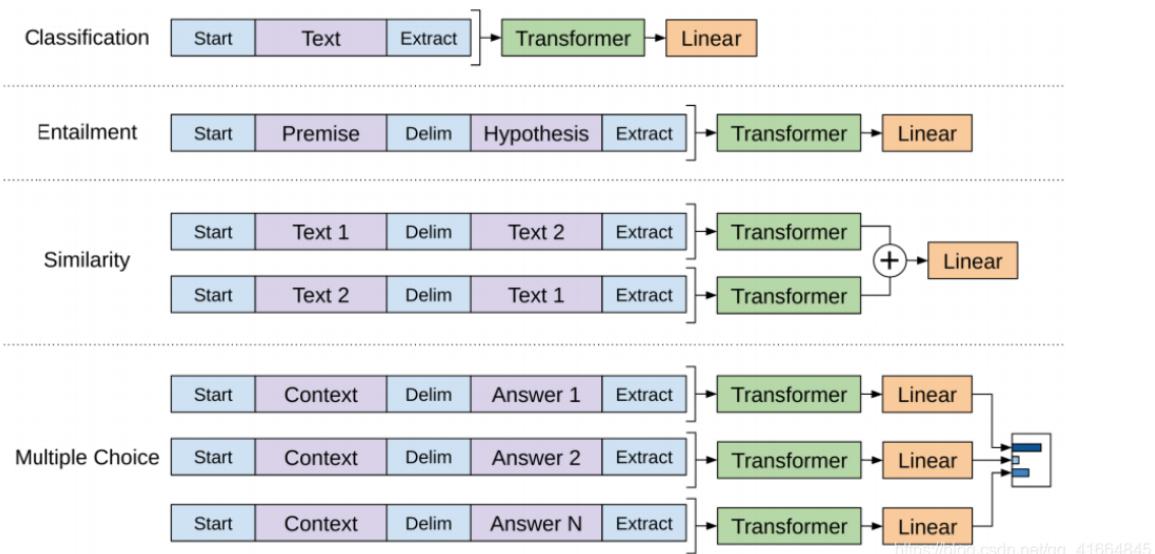
## Transfer Learning to Downstream Tasks

通过OpenAI的transformer的预训练和一些微调后，我们就可以将训练好的模型，用于其他下游NLP任务啦。（比如训练一个语言模型，然后拿他的hidden state来做分类。），下面就介绍一下这个骚操作。（还是如上面例子：分为垃圾邮件和非垃圾邮件）



[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

OpenAI论文概述了许多Transformer使用迁移学习来处理不同类型NLP任务的例子。如下图例子所示：



[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

## BERT: From Decoders to Encoders

OpenAI transformer为我们提供了基于Transformer的精密的预训练模型。但是从LSTM到Transformer的过渡中，我们发现少了些东西。ELMo的语言模型是双向的，但是OpenAI的transformer是前向训练的语言模型。我们能否让我们的Transformer模型也具有Bi-Lstm的特性呢？

R-BERT: “Hold my beer”

# Masked Language Model

BERT说：“我要用transformer的encoders”

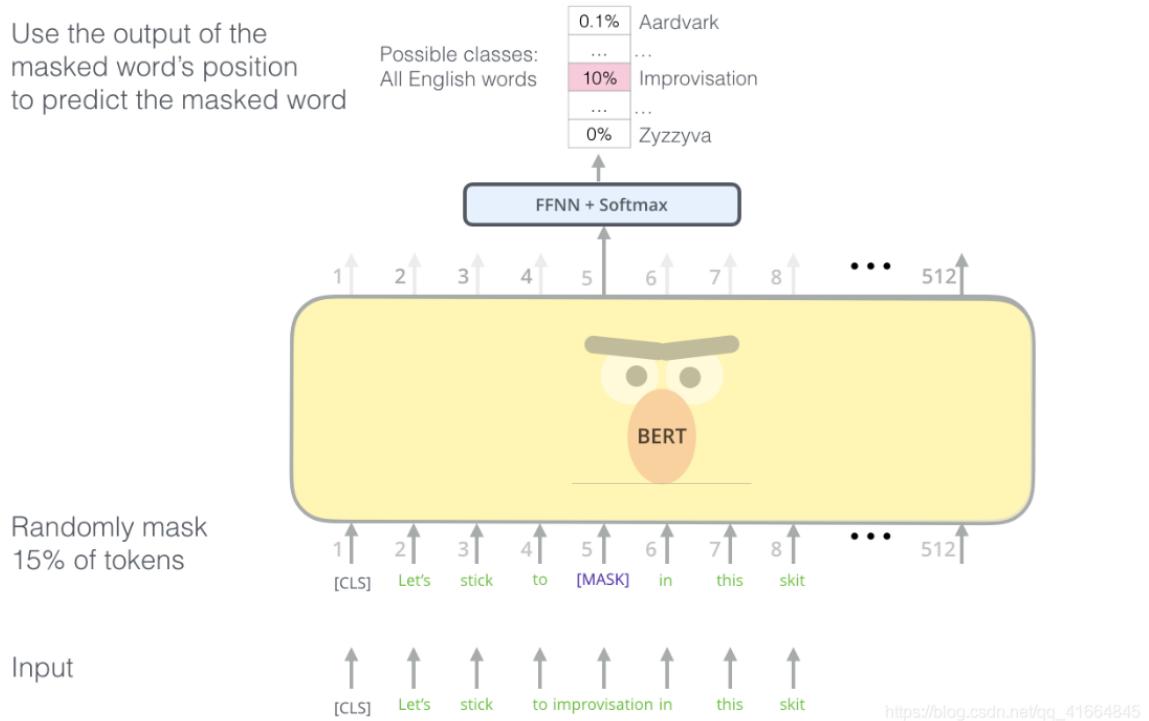
Ernie不屑道：“呵呵，你不能像Bi-Lstm一样考虑文章”

BERT自信回答道：“我们会用masks”

解释一下Mask：

语言模型会根据前面单词来预测下一个单词，但是self-attention的注意力只会放在自己身上，那么这样100%预测到自己，毫无意义，所以用Mask，把需要预测的词给挡住。

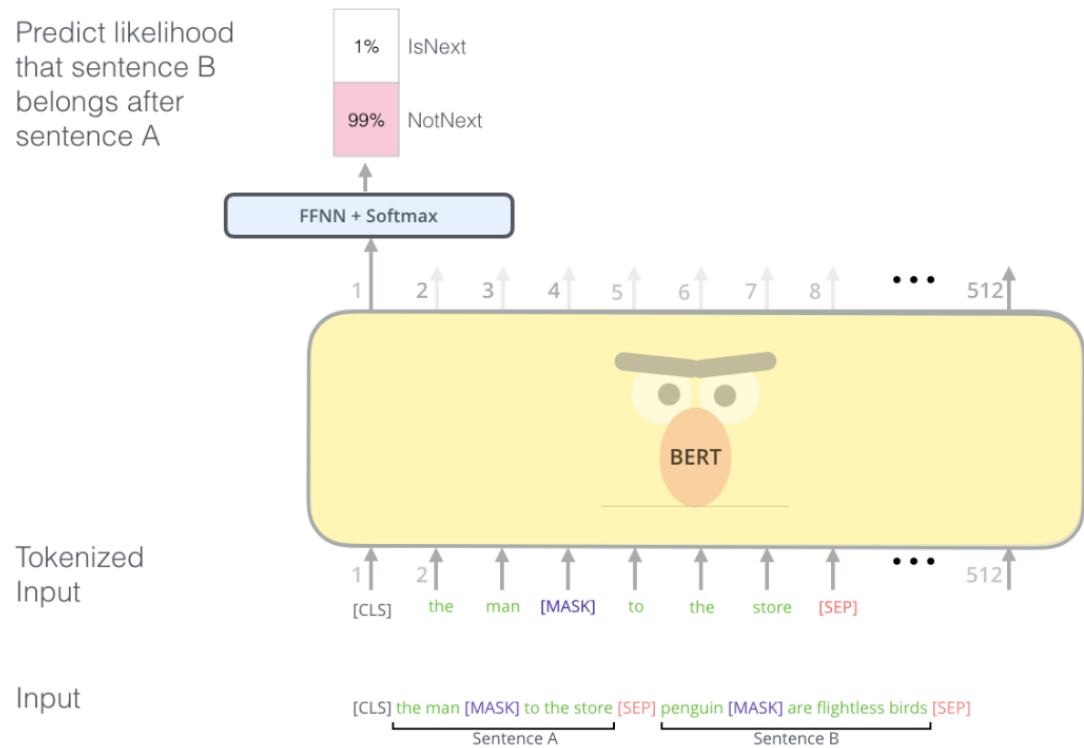
如下图：



## Two-sentence Tasks

我们回顾一下OpenAI transformer处理不同任务的输入转换，你会发现在某些任务上我们需要2个句子作为输入，并做一些更为智能的判断，比如是否相似，比如给出一个维基百科的内容作为输入，同时在放入一条针对该条目的问题，那么我们的算法模型能够处理这个问题吗？

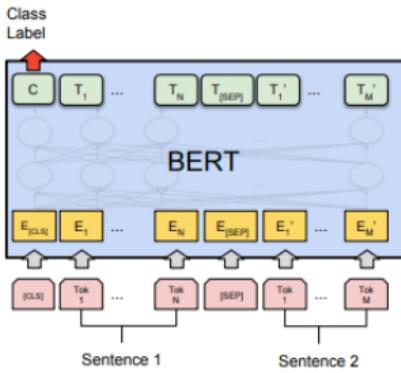
为了使BERT更好的处理2个句子之间的关系，预训练的过程还有一个额外的任务：给定2个句子（A和B），A与B是否相似？（0或者1）



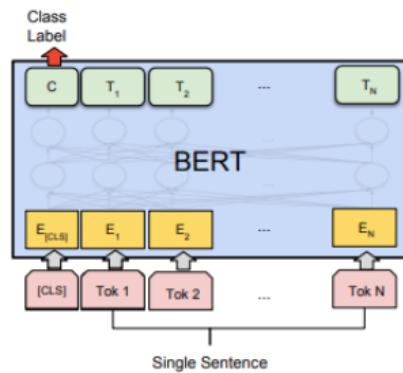
## 特殊NLP任务

BERT的论文为我们介绍了几种BERT可以处理的NLP任务：

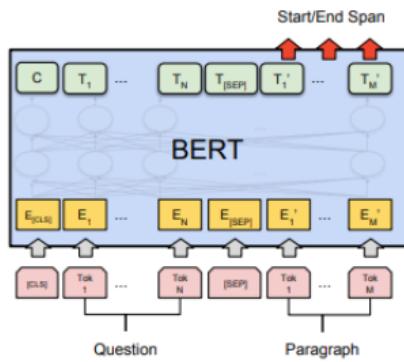
1. 短文本相似
2. 文本分类
3. QA机器人
4. 语义标注



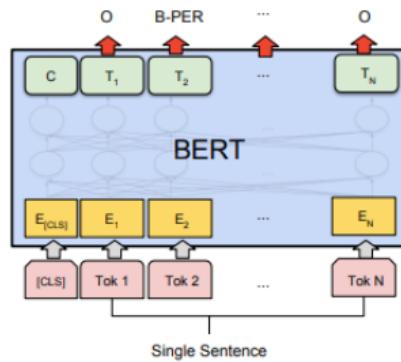
(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG



(b) Single Sentence Classification Tasks:  
SST-2, CoLA



(c) Question Answering Tasks:  
SQuAD v1.1

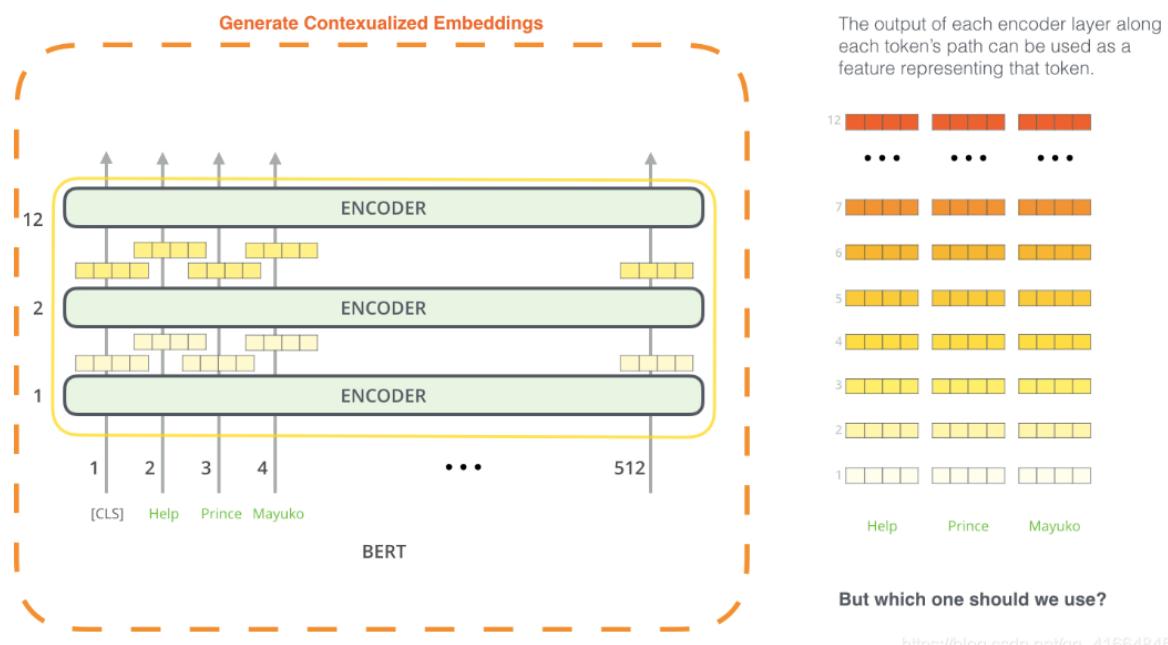


(d) Single Sentence Tagging Tasks:  
CoNLL-2003 NER

[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

## BERT用做特征提取

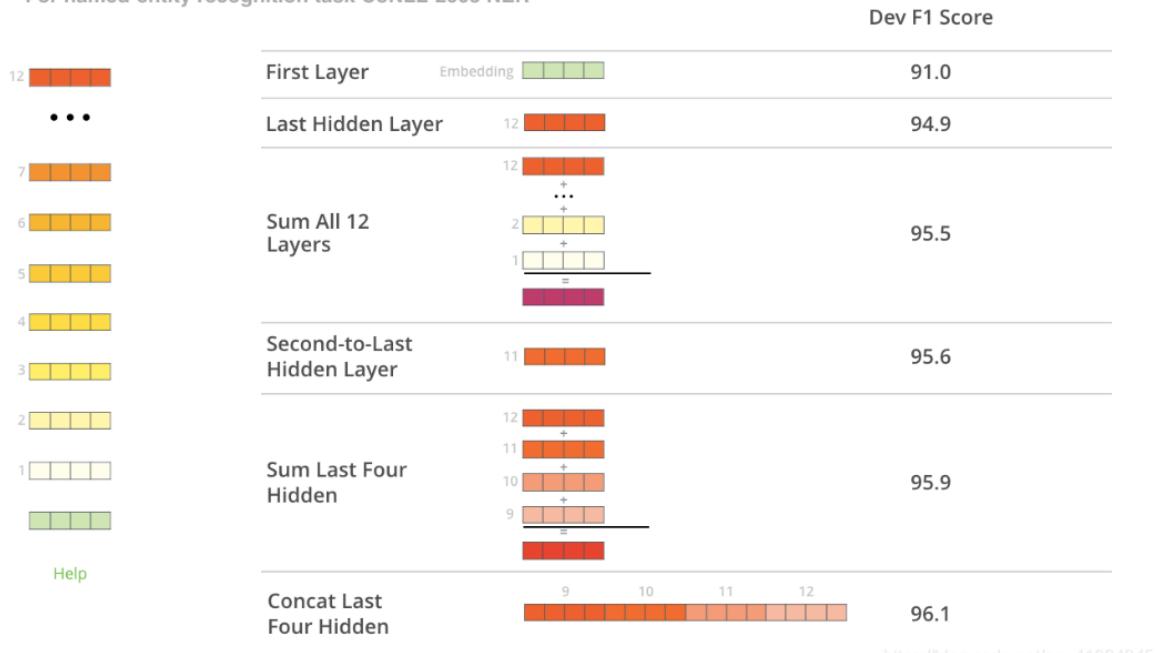
微调方法并不是使用BERT的唯一方法，就像ELMo一样，你可以使用预选训练好的BERT来创建语境化词嵌入。然后你可以将这些嵌入提供给现有的模型。



[https://blog.csdn.net/qq\\_41664845](https://blog.csdn.net/qq_41664845)

哪个向量最适合作为上下文嵌入？我认为这取决于任务。本文考察了六种选择（与微调模型相比，得分96.4）：

What is the best contextualized embedding for “Help” in that context?  
For named-entity recognition task CoNLL-2003 NER



## 如何使用BERT

使用BERT的最佳方式是通过 [BERT FineTuning with Cloud TPUs](#) 谷歌云上托管的笔记。如果你未使用过谷歌云TPU可以试试看，这是个不错的尝试。另外BERT也适用于TPU, CPU和GPU

下一步是查看BERT仓库中的代码：

1. 该模型在modeling.py (BertModel类) 中构建，与vanilla Transformer编码器完全相同。
2. run\_classifier.py是微调过程的一个示例。它还构建了监督模型的分类层。如果要构建自己的分类器，请查看该文件中的create\_model()方法。
3. 可以下载几种预先训练的模型。涵盖102种语言的多语言模型，这些语言都是在维基百科的数据基础上训练而成的。
4. BERT不会将单词视为tokens。相反，它注重WordPieces。tokenization.py是将你的单词转换为适合BERT的wordPieces的tokensizer。

## Attention Is All You Need

Transformer完全基于Attention机制

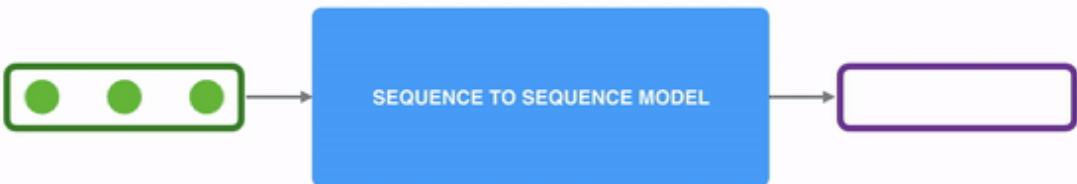
基于RNN的Seq2Seq模型难以处理长序列句子,因此顺序性也无法并行处理,基于CNN的Seq2Seq模型虽然可以实现并行,但消耗内存

Self-Attention是一种将序列不同位置联系起来并计算序列表示的注意力机制

Transformer完全使用Self-Attention

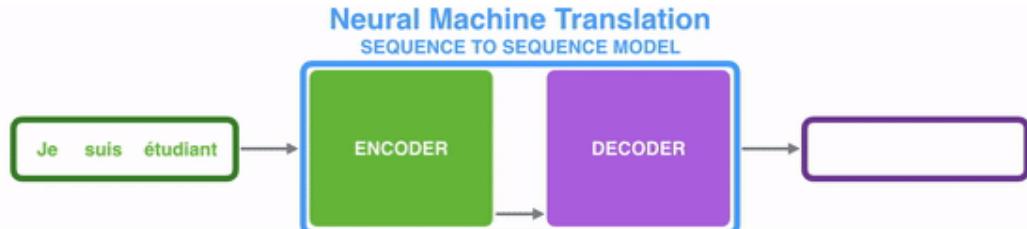
## Seq2Seq

Seq2Seq可以理解为输入一个序列,然后经过一个黑盒可以得到另一个序列



黑盒中是Encoder-Decoder框架.输入体一个序列,然后编码器进行编码得到上下文信息C然后通过解码器逐一解码,得到另一个序列

- 输入/输出序列都是Embedding向量
- 上下文信息C是一个向量,其维度与编码器的数量有关,256,512
- 逐一解码,解码器根据上下文C和先前生成的历史信息生成此时刻的输出

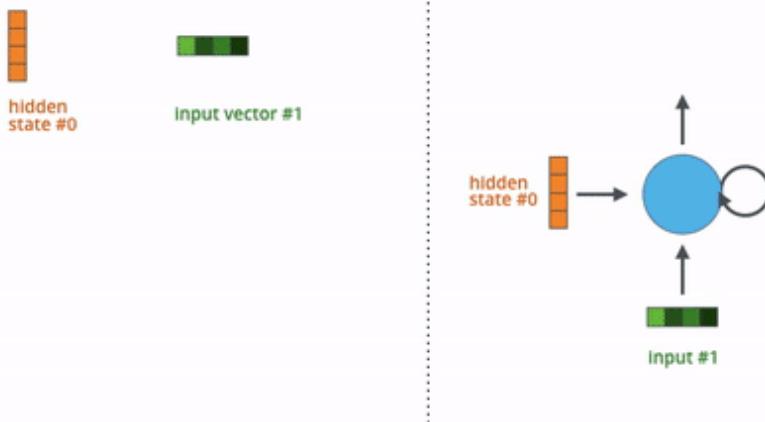


## 传统RNN

### Recurrent Neural Network

#### Time step #1:

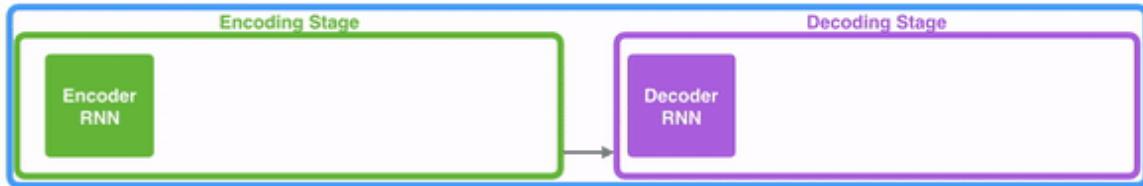
An RNN takes two input vectors:



在编码器之间进行传递的其实是隐藏层的状态

## Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL



Je suis étudiant

\

编码器中最后一个 RNN 的隐藏状态就是要传给解码器的上下文信息 Context。

## Attention

对于Encoder-Decoder框架,上下文C的质量决定着模型的性能.RNN无法处理长序列的问题

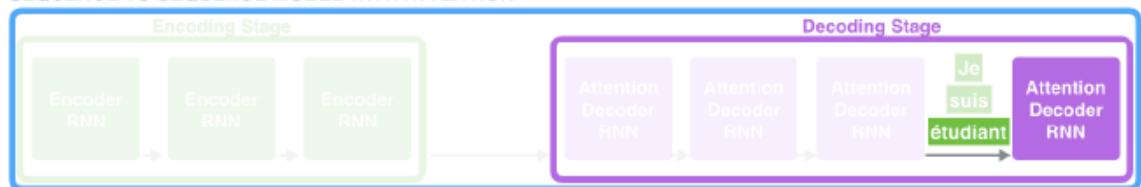
Attention允许模型根据需要来关注输入序列的某些相关部分

Attention主要应用于Seq2Seq的解码器中.

Time step: 7

## Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



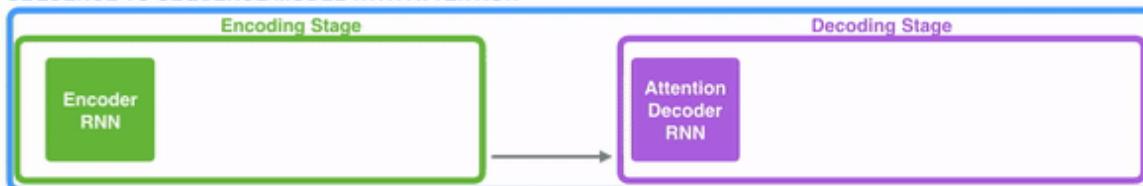
注意力机制使解码器能够于生成英语翻译之前将注意力集中在单词étudiant上

## Attention与Seq2Seq主要区别

- 更多的Context信息:编码器不传递编码阶段的最后一个隐藏状态.而是将所有的隐藏状态传递给解码器
- 解码时加入Attention

## Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



Je suis étudiant

解码器中加入Attention的具体步骤

1. 查看编码器隐藏状态的集合（每个编码器隐藏状态都与输入句子的某个单词有很大关联）；
2. 给每个隐藏状态打分（计算编码器的隐藏状态与解码器的隐藏状态的相似度）；
3. 将每个隐藏状态打分通过的 Softmax 函数计算最后的概率；
4. 将第 3 步计算的概率作为各个隐藏状态的权重，并加权求和得到当前 Decoder 所需的 Context 信息。

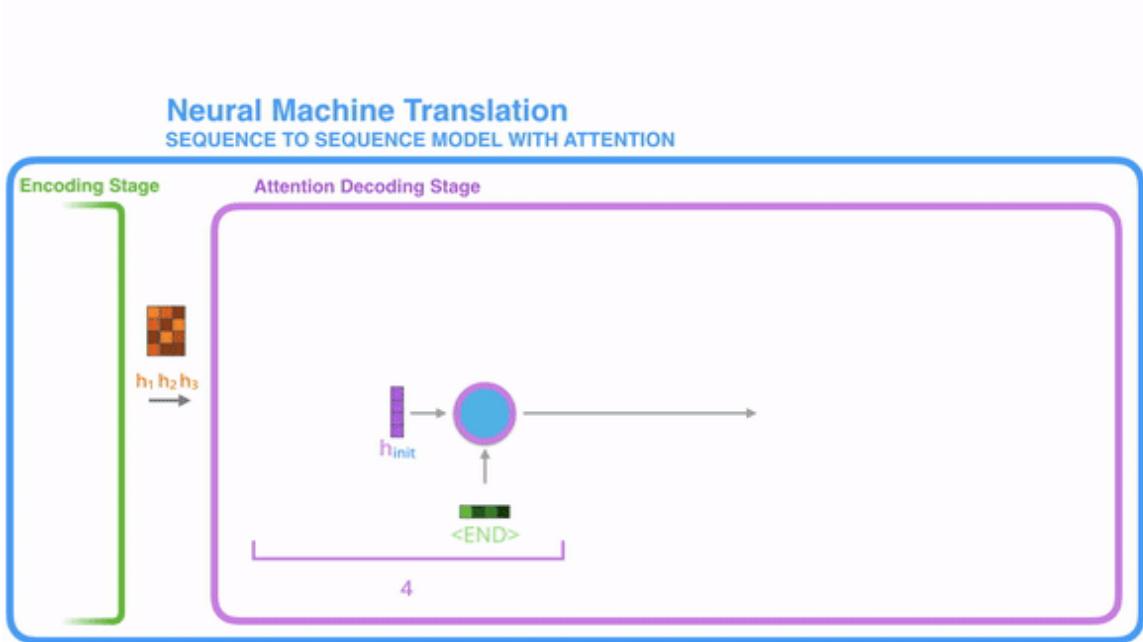
这种 Attention 操作在解码器每次解码的时候都需要进行

#### Attention at time step 4



Attention 的工作流程：

1. 解码器中的第一个 RNN 有两个输入：一个是表示 标志的 Embedding 向量，另一个来自解码器的初始隐藏状态；
2. RNN 处理两个输入，并产生一个输出和一个当前 RNN 隐藏层状态向量 ( $h_4$ )，输出将直接被舍去；
3. 然后是 **Attention 操作**：利用**编码器传来的隐藏层状态集合**和刚刚得到 RNN 的隐藏层状态向量 ( $h_4$ ) 去计算当前的上下文向量 ( $C_4$ )；
4. 然后拼接  $h_4$  和  $C_4$ ，并将拼接后的向量送到前馈神经网络中；
5. 前馈神经网络的到的输出即为当前的输出单词的 Embedding 向量；
6. 将此 RNN 得到的单词向量并和隐藏层状态向量 ( $h_4$ )，作为下一个 RNN 的输入，重复计算直到解码完成。

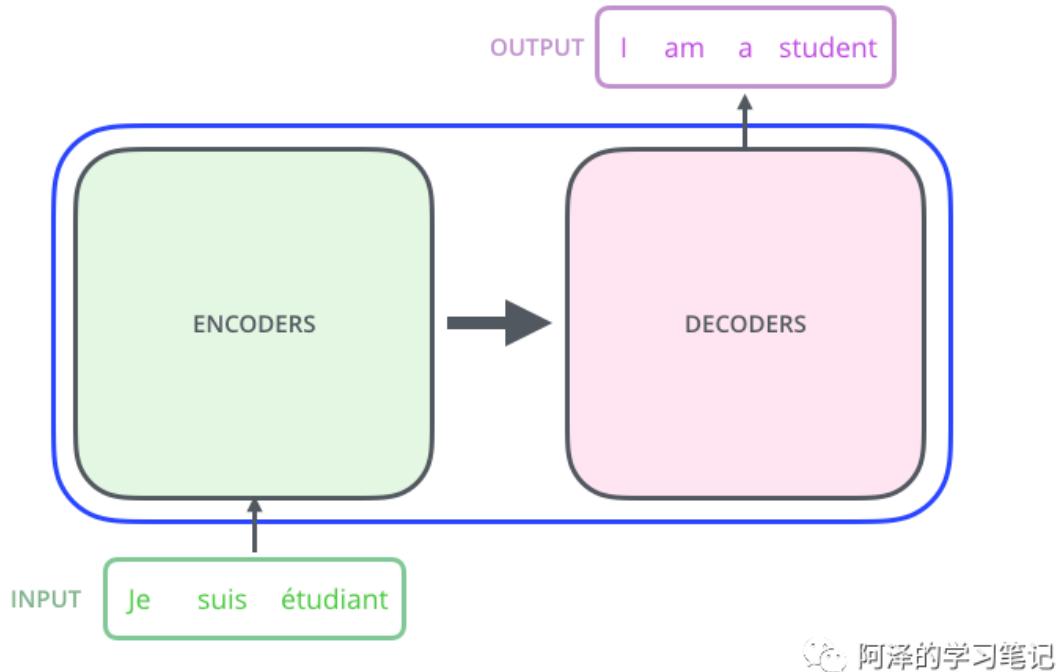


左边是隐藏层状态的集合，右边是对隐藏的一个加权结果。

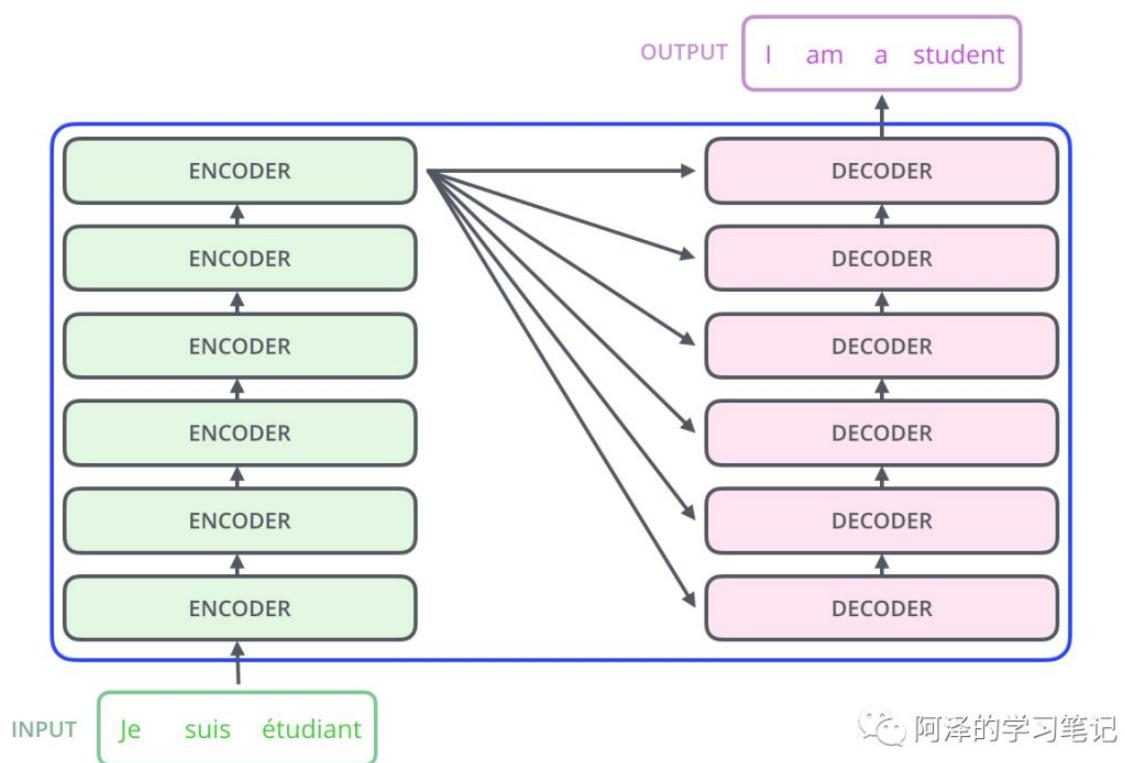
这里的模型并不是盲目地将输出中的第一个单词与输入中的第一个单词对齐，事实上，它从训练的时候就已经学会了如何排列语言对中的单词。下面再给出 Attention 论文中的一个例子（模型在输出“European Economic Area”时，其与法语中的顺序恰好的是相反的）：

## Transformer

### 模型结构



编码部分是堆了六层编码器，解码部分也堆了六个解码器。



所有的编码器在结构上都是相同的，每一个都被分成两个子层

编码器的输入首先经过一层 Self-Attention，这一层主要用来帮助编码器在对特定的单词进行编码时查看输入句子中的其他单词，后面我们会仔细研究 Self-Attention。

Self-Attention 层的输出进入给前馈神经网络（Feed Forward Neural Network，以下简称 Feed Forward），所有前馈神经网络的结构都相同并且相互独立。

解码器拥有三层，除了 Self-Attention 和 Feed Forward 外，还有一层 Encoder-Decoder Attention 层（以下简称 Attention 层，区别于 Self-Attention），Attention 层位于 Self-Attention 和 Feed Forward 层之间，主要用来帮助解码器将注意力集中在输入语句的相关部分（类似于 Seq2Seq 模型中的 Attention）。

## Encoder Side编码器

在 Self-Attention 层中，这些单词之间存在依赖关系；但 Feed Forward 层没有依赖，所以可以在 Feed Forward 层并行化训练。

### Self-Attention

谷歌论文中首次提出 Self-Attention 概念，我们来看一下它是怎么工作的。

假设我们现在要翻译下面的这个句话：

"The animal didn't cross the street because it was too tired"

这个句子中的 it 指的是什么呢？是指 street 还是 animal 呢？这对人来说比较简单，但是对算法来说就没那么简单了。

而 Self-Attention 就可以解决这个问题，在处理 it 时会将它和 animal 联系起来。

Self-Attention 允许当前处理的单词查看输入序列中的其他位置，从而寻找到有助于编码这个单词的线索。

以下图为例，展示了第五层的编码器，当我们在编码 it 时，Attention 机制将 The animal 和自身的 it 编码到新的 it 上。

**第一步**，我们对于每个单词来说我们都一个 Embedding 向量，下图绿色部分。

此外，对于每个单词我们还会有三个向量分别为：查询向量（Querry）、键向量（Key）和值向量（Value），这些向量是单词的 Embedding 向量分别与对应的查询矩阵  $W^Q$ 、键矩阵  $W^K$  和值矩阵  $W^V$  相乘得来的。

对于 Embedding 向量来说是 512 维，而对于新向量来说是 64 维。（大小是架构设定的。）

**第二步**，我们来看下 Self-Attention 是如何计算的。

假设我们正在计算 Thinking 的 Self-Attention。我们需要将这个单词和句子中的其他单词得分，这个分数决定了我们将一个单词编码到某个位置时，需要将多少注意力放在句子的其他部分。

这个得分是通过当前单词的 Querry 向量和其他词的 Key 向量做内积得到的。

（也可以理解为当前单词的是由句子的所有单词加权求和得到的，现在计算的是当前单词和其他单词的分数，这个分数将用于后面计算各个单词对当前单词的贡献权重。）

**第三步**，将这个分数处以 8（Value 向量是 64 维，取平方根，主要是为了稳定梯度）。然后将分数通过 Softmax 标准化，使它们都为正，加起来等于1。

经过 Softmax 后的分数决定了序列中每个单词在当前位置的表达量（如，对着 Thinking 这个位置来说， $= 0.88 * \text{Thinking} + 0.12 * \text{Machines}$ ）。Softmax 分数越高表示与当前单词的相关性更大。

**第四步**，将每个单词的 Value 向量乘以 Softmax 分数并相加得到一个汇总的向量，这个向量便是 Self-Attention 层的输出向量。

得到的这个向量会输送给下一层的 Feed Forward 网络。

在实际的实现过程中，为了快速计算，我们是通过矩阵运算来完成的。

首先是输入矩阵与查询矩阵、键矩阵和值矩阵。

然后用 softmax 计算权重，并加权求和：

## Multi-Head Attention

论文中模型框架中画的 Multi-Head Attention 层，Multi-Head Attention 其实就是包含了多个 Self-Attention。所以 Multi-Head Attention 有多个不同的查询矩阵、键矩阵和值矩阵，**为 Attention 层提供了多个表示空间。**

Transformer 中每层 Multi-Head Attention 都会使用八个独立的矩阵，所以也会得到 8 个独立的 Z 向量：

但是 Feed Forward 层并不需要 8 个矩阵，它只需要一个矩阵（每个单词对应一个向量）。所以我们需要一种方法把这8个压缩成一个矩阵。这边还是采用矩阵相乘的方式将 8 个 Z 向量拼接起来，然后乘上另一个权值矩阵 W，得到后的矩阵可以输送给 Feed Forward 层。

## 总模型

## Positional Encoding位置编码

模型可以完成单词的 Attention 编码了，但是目前还只是一个词袋结构，还需要描述一下单词在序列中顺序问题。

为了解决这个问题，Transformer 向每个输入的 Embedding 向量添加一个位置向量，有助于确定每个单词的绝对位置，或与序列中不同单词的相对位置：

这种方式之所以有用，大概率是因为，将配置信息添加到 Embedding 向量中可以在 Embedding 向量被投影到Q/K/V 向量后，通过 Attention 的点积提供 Embedding 向量之间有效的距离信息。

举个简单的例子，以 4 维为例：

下图显示的是，每一行对应一个位置编码向量。所以第一行就是我们要添加到第一个单词的位置向量。每一行包含512个值——每个值的值在1到-1之间（用不同的颜色标记）。

这张图是一个实际的位置编码的例子，包含 20 个单词和 512 维的 Embedding 向量。可以看到它从中间一分为二。这是因为左半部分的值是由一个 Sin 函数生成的，而右半部分是由另一个 Cos 函数生成的。然后将它们连接起来，形成每个位置编码向量。这样做有一个很大的优势：他可以将序列扩展到一个非常的长度。使得模型可以适配比训练集中出现的句子还要长的句子。

## Residuals 残差

每个编码器中的每个子层 (Self-Attention, Feed Forward) 都有一个围绕它的虚线，然后是一层 ADD & Normalize 的操作。

这个虚线其实就是一个残差，为了防止出现梯度消失问题。而 Add & Normalize 是指将上一层传过来的数据和通过残差结构传过来的数据相加，并进行归一化：

同样适用于解码器的子层：

## Decoder Side 解码器

编码器首先处理输入序列，然后将顶部编码器的输出转换成一组 Attention 矩阵 K 和 V，这两个矩阵主要是给每个解码器的“Encoder-Decoder Attention”层使用的，这有助于解码器将注意力集中在输入序列中的适当位置：

像我们处理编码器的输入一样，我们将输出单词的 Embedding 向量和位置向量合并，并输入到解码器中，然后通过解码器得到最终的输出结果。

在解码器中，Self-Attention 层只允许注意到输出单词注意它前面的单词信息。在实现过程中通过在将 Self-Attention 层计算的 Softmax 步骤时，屏蔽当前单词的后面的位置来实现的（设置为-inf）。

解码器中的“Encoder-Decoder Attention”层的工作原理与“Multi-Head Attention”层类似，只是它从其下网络中创建查询矩阵，并从编码器堆栈的输出中获取键和值矩阵（刚刚传过来的 K/V 矩阵）。

## Softmax Layer

解码器输出浮点数向量，我们怎么把它变成一个单词呢？

这就是最后一层 Linear 和 Softmax 层的工作了。

Linear 层是一个简单的全连接网络，它将解码器产生的向量投影到一个更大的向量上，称为 logits 向量。

假设我们有 10,000 个不同的英语单词，这时 logits 向量的宽度就是 10,000 个单元格，每个单元格对应一个单词的得分。这就解释了模型是怎么输出的了。

然后利用 Softmax 层将这些分数转换为概率。概率最大的单元格对应的单词作为此时的输出。

## Training 训练

训练时我们需要一个标注好的数据集。

为了形象化，我们假设词汇表只包含 5 个单词和一个结束符号：

然后我们给每个单词一个 One-Hot 向量：

假设我们刚开始进行训练，模型的参数都是随机初始化的，所以模型的输出和期望输出有所偏差。

我们计算两者的损失函数并通过反向传播的方式来更新模型。

在一个足够大的数据集上对模型进行足够长的时间的训练之后，我们希望生成的概率分布是这样的：

当训练好的得到模型后，我们需要为某个句子进行翻译。有两种方式确定输出单词：

- Greedy Decoding：直接取概率最大的那个单词；
- Beam Search：取最大的几个单词作为候选，分别运行一次模型，然后看一下哪组错误更少。这里的超参成为 Beam Size。

## Improving Language Understanding by Generative Pre-Training

---

NLP 领域中只有小部分标注过的数据，而有大量的数据是未标注，如何只使用标注数据将会大大影响深度学习的性能，所以为了充分利用大量未标注的原始文本数据，需要利用无监督学习来从文本中提取特征，最经典的例子莫过于词嵌入技术。

但是词嵌入只能 word-level 级别的任务（同义词等），没法解决句子、句对级别的任务（翻译、推理等）。出现这种问题原因有两个：

- 首先，是因为不清楚要下游任务，所以也就没法针对性的进行优化；
- 其次，就算知道了下游任务，如果每次都要大改模型也会得不偿失。

为了解决以上问题，作者提出了 GPT 框架，用一种半监督学习的方法来完成语言理解任务，GPT 的训练过程分为两个阶段：Pre-training 和 Fine-tuning。目的是在于学习一种通用的 Representation 方法，针对不同种类的任务只需略作修改便能适应。

## How GPT3 Works - Visualizations and Animations

---

一个经过训练的语言模型会生成文本。

我们可以选择一些文本作为输入传递给它，从而影响它的输出。

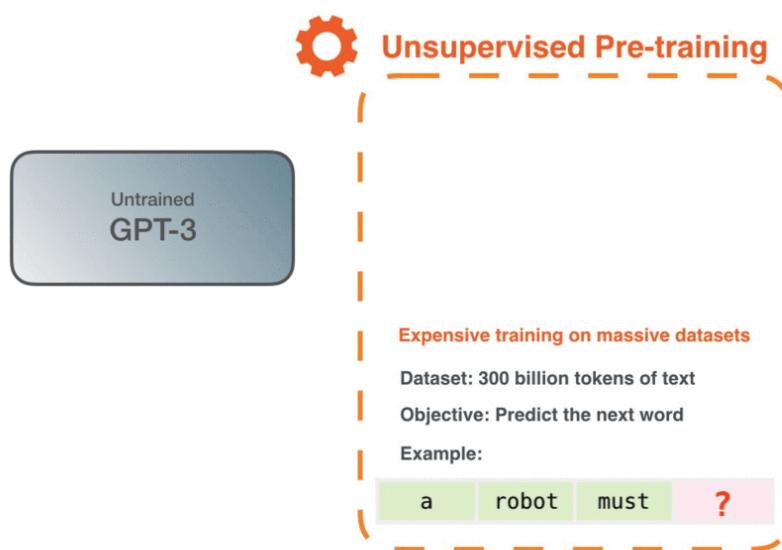
输出是由模型在扫描大量文本的训练期间 "学到" 的东西生成的。

**Input Prompt:** Recite the first law of robotics



**Output:**

训练是将模型暴露在大量文本中的过程。它已经做了一次并完成了。你现在看到的所有实验都是来自那个训练过的模型。据估计，它耗费了355个GPU年，花费了460万美元。



一个有3000亿个字符的数据集被用来生成模型的训练样本。例如，这是由上面那句话生成的三个训练样本。

你可以看到你如何在所有文本上滑动一个窗口，并生成很多例子。

**Text:** Second Law of Robotics: A robot must obey the orders given it by human beings

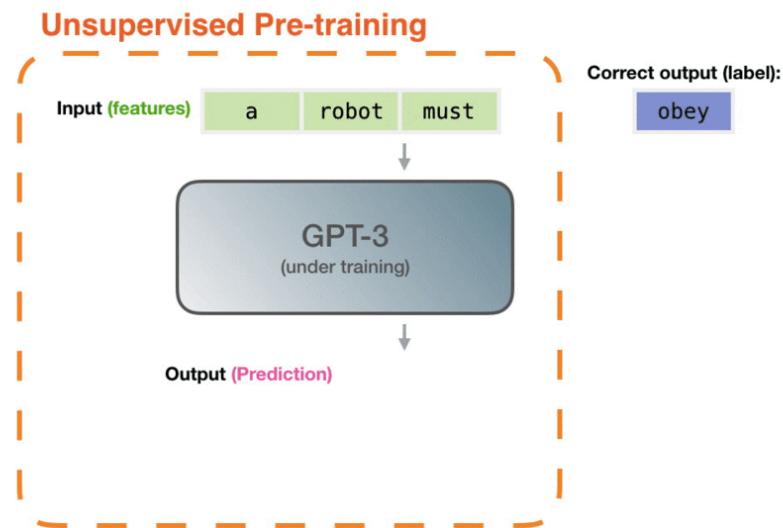
Generated training examples

Example #	Input (features)	Correct output (labels)
1	Second law of robotics : a	
2	Second law of robotics : a	robot
3	Second law of robotics : a robot	must
...		

当我们只给模型一个样本时：我们只给看特征，并让它预测下一个单词。

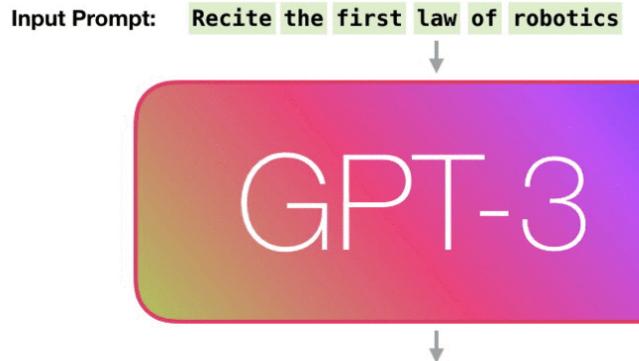
该模型的预测将是错误的。我们计算其预测中的错误，并更新模型，以便下次它做出更好的预测。

重复这个过程数百万次



现在让我们更详细地看看这些相同的步骤。

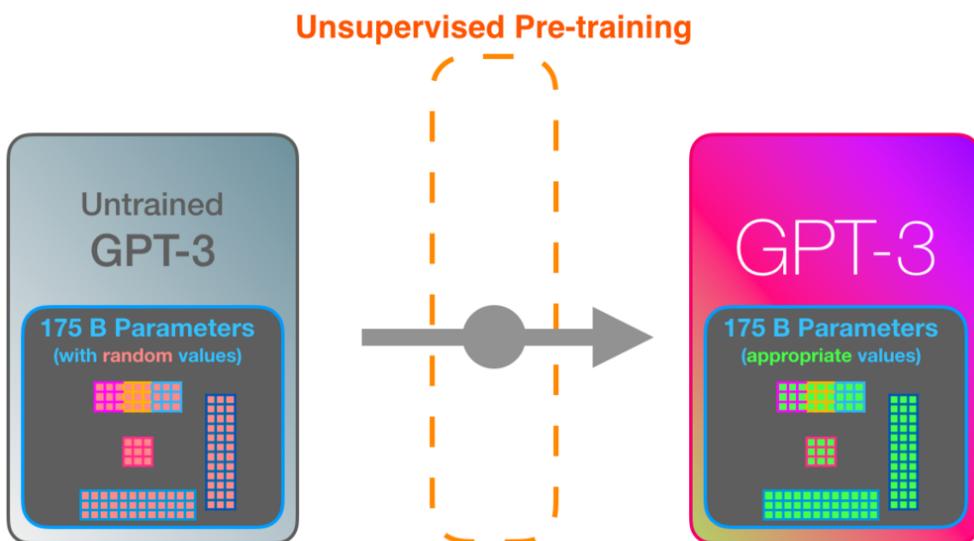
GPT3实际上一次只生成一个token的输出（现在我们假设一个token是一个词）。



请注意：这是对GPT-3工作原理的描述，而不是对它的新颖之处的讨论（主要是规模大得可笑）。其架构是基于的transformer解码器模型，参见这篇[论文](#)。

GPT3 极其巨大。它将从训练中学习到的内容编码成1750亿个参数。这些参数用于计算每次运行时生成的 token。

未经训练的模型以随机参数开始。训练以期找到更好的预测值。

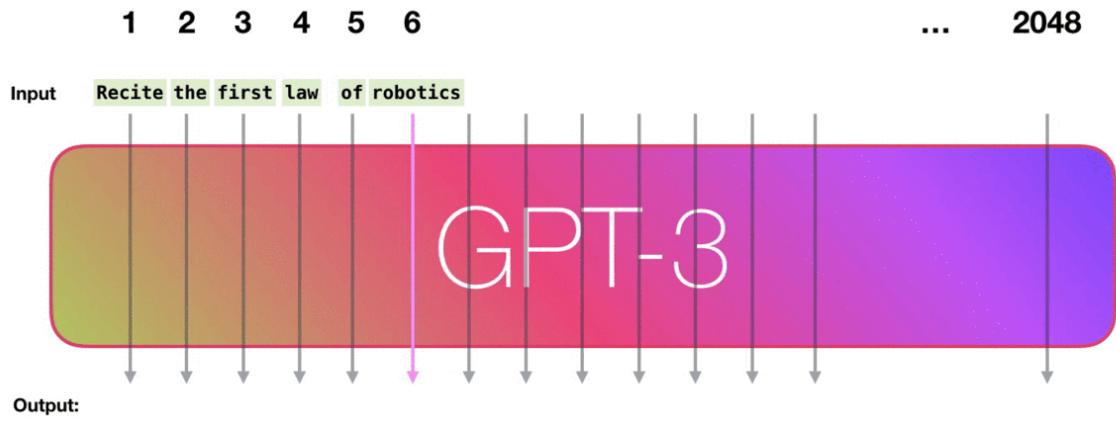


这些数字是模型里面数百个矩阵的一部分。预测主要就是大量的矩阵乘法。

在我的[YouTube 上的人工智能介绍](#)中，我展示了一个简单的机器学习模型，它只有一个参数。为解读这个1750亿个参数的怪兽开了个好头。

为了理解这些参数是如何分布和使用的，我们需要打开模型看看里面的情况。

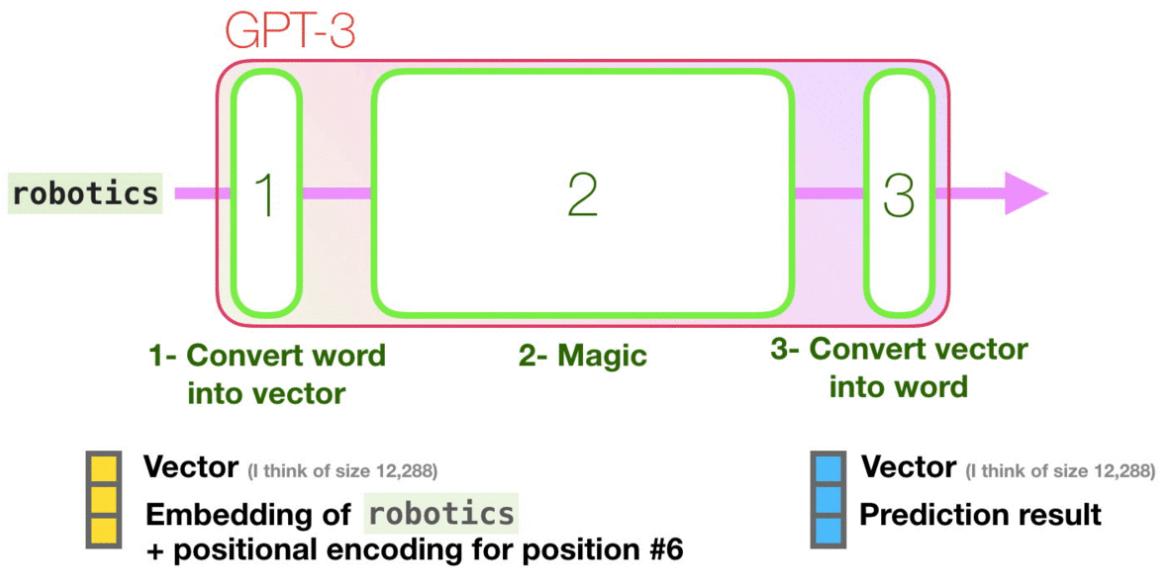
GPT3 的宽度是2048个 token。这是它的 "上下文窗口"。这意味着它沿着这2048条轨道处理 token。



让我们跟随紫轨，看看系统是如何处理"机器人"这个词并产生"A"的？

抽象的步骤：

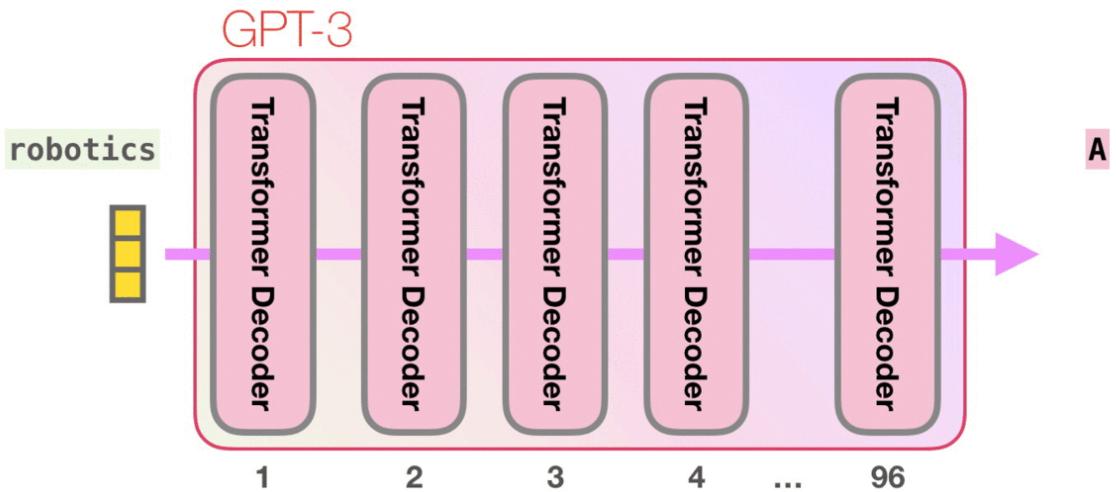
1. 将单词转换为[代表单词的向量（数字列表）](#)。
2. 计算预测值
3. 将所得向量转换为单词



GPT3的重要计算发生在其96个transformer解码层的堆栈中。

看到这些层了吗？这就是 "深度学习" 中的 "深度"。

这些层中的每一层都有1.8亿个参数来进行计算。



你可以在我的博文[图解GPT2](#)中看到解码器内部一切的详细解释。

与GPT3的不同之处在于密集自注意层和稀疏自注意层的交替。

这是GPT3内输入和响应 ("Okay human") 的X光片。注意，每一个token是如何通过整个层堆栈的。我们不关心第一个词的输出。当输入完成后，我们开始关心输出。我们把每个词都反馈到模型中。

在React代码生成的例子中，描述会是输入提示(绿色)，此外还有几个对代码描述的例子吧。而 React 代码会像这里的粉色 token 一样一个个地生成。

我的假设是，将引例和描述作为输入，用特定的 token 将例子和结果分开，然后输入到模型中。



这种方式让人印象深刻。因为我们只要等到GPT3的微调推出。它的性能将更加惊人。

微调实际上是更新模型的权重，让模型在某项任务中表现得更好。

