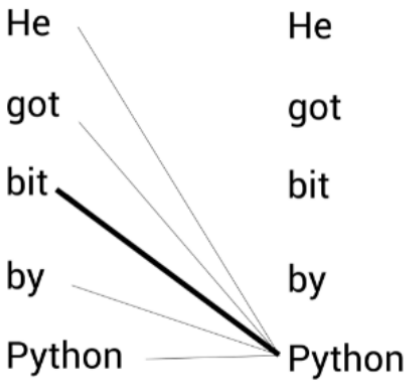
BERT的基本思想

BERT如此成功的一个原因之一是它是基于上下文(**context-based**)的嵌入模型，不像其他流行的嵌入模型，比如word2vec，是上下文无关的(**context-free**)。

Sentence A：He got bit by Python.
Sentence B：Python is my favorite programming language.

在句子A中，Python是蟒蛇的意思，而句子B中是一种编程语言。
word2vec会为这两个句子中的Python赋予相同的嵌入。因为它是上下文无关的。
BERT是基于上下文的模型，它可以根据上下文来生成单词的嵌入。因此它会给上面两个句子中的Python不同的嵌入向量。
BERT是如何理解上下文的?

句子A，为了理解单词 `Python` 的语境意思，BERT将单词 `Python` 与其他所有单词(包括自己)联系起来



BERT能通过 `bit` 一词理解此句中的 `Python` 指的是蛇。

BERT的原理

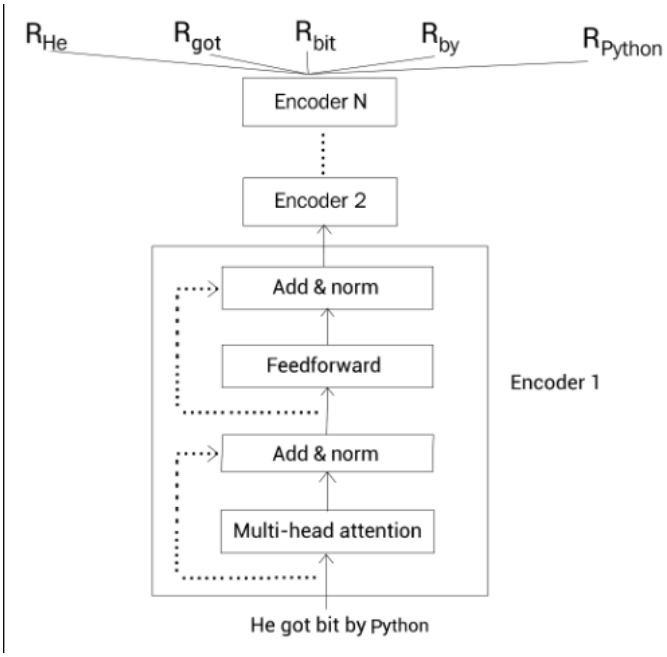
从BERT的全称，**B**idirectional **E**ncoder **R**epresentation from **T**ransformer(来自Transformer的双向编码器表征)，可以看出BERT是基于Transformer模型的

举一个例子来理解BERT是如何从Transformer中得到双向编码表示的

假设我们有一个句子A：`He got bit by Python`，把这个句子输入Transformer并得到了每个单词的上下文表示(嵌入表示)作为输出。

Transformer的编码器通过多头注意力机制理解每个单词的上下文，然后输出每个单词的嵌入向量。

如图，输入一个句子到Transformer的编码器，它输出句子中每个单词的上下文表示。我们可以叠加N个编码器。这样，通过BERT，给定一个句子，我们就得到了句子中每个单词的上下文嵌入向量表示。



RHe代表单词 `He` 的向量表示

BERT的配置

两种标准的配置：

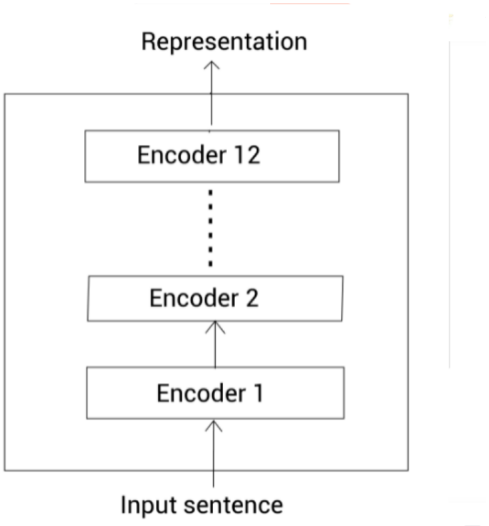
- BERT-base
- BERT-large

BERT-base

BERT-base包含12个编码器层。所有的编码器使用12个注意头。编码器中的全连接网络包含768个隐藏单元。因此，从该模型中得到的向量大小也就是768。

编码器层数记为L
注意力头数记为A
隐藏单元数记为H

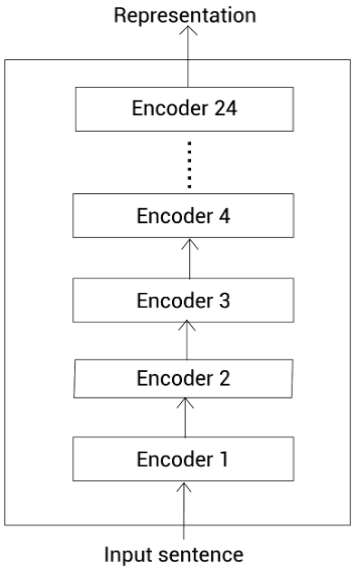
因此BERT-base模型，L = 12 , A = 12 , H = 768



BERT-large

BERT-large包含24个编码器层。所有的编码器使用16个注意头。编码器中的全连接网络包含1024个隐藏单元。因此，从该模型中得到的向量大小也就是1024。

因此BERT-large模型，L = 24 , A = 16 , H = 1024



BERT的其他配置

BERT-tiny, L = 2 , H = 128
BERT-mini, L = 4 , H = 256
BERT-small, L = 4 , H = 512
BERT-medium, L = 8 , H = 512

当计算资源有限时，我们可以使用这些更小的BERT模型。

预训练BERT模型

预训练的意思是，假设我们有一个模型m，首先我们为某种任务使用大规模的语料库训练模型m。

现在来了一个新任务，并有一个新模型，我们使用已经训练过的模型(预训练的模型)m的参数来初始化新的模型，而不是使用随机参数来初始化新模型。
然后根据新任务调整(微调)新模型的参数。也是迁移学习。

输入数据表示

在把数据输入到BERT之前，通过下面三个嵌入层将输入转换为**嵌入向量**：

- 标记嵌入(Token embedding)
- 片段嵌入(Segment embedding)
- 位置嵌入(Position embedding)

标记嵌入

首先，有一个**标记嵌入层**。

例子：Sentence A: Paris is a beautiful city. Sentence B: I love Paris.

首先对这两个句子分词，得到分词后的标记(单词)，然后连到一起。

tokens = [Paris, is, a, beautiful, city, I, love, Paris]

增加一个新的标记，叫作`[CLS]`标记，到第一个句子前面：增加一个新的标记，叫作`[SEP]`标记，到每个句子的结尾：

tokens = [[CLS], Paris, is, a, beautiful, city, I, love, Paris]

`[CLS]`标记只加在第一个句子前面，而`[SEP]`标记加到每个句子末尾。

`[CLS]`标记用于分类任务，而`[SEP]`标记用于表示每个句子的结尾。

使用一个叫作标记嵌入的嵌入层把这些标记转换为嵌入向量。

E [CLS]表示标记[CLS]的嵌入

片段嵌入

然后有一个片段嵌入层，用于区分给定的两个句子。

在对上面两个句子分词之后，我们得到了：

tokens = [[CLS], Paris, is, a, beautiful, city, [SEP], I, love, Paris, [SEP]]

把它输入片段嵌入层，片段嵌入层只返回两种嵌入，E A 或E B作为输出。即如果输入标记属于句子A，那么该标记会映射到嵌入E A ； 反之属于句子B的话，则映射到嵌入E B。

位置嵌入

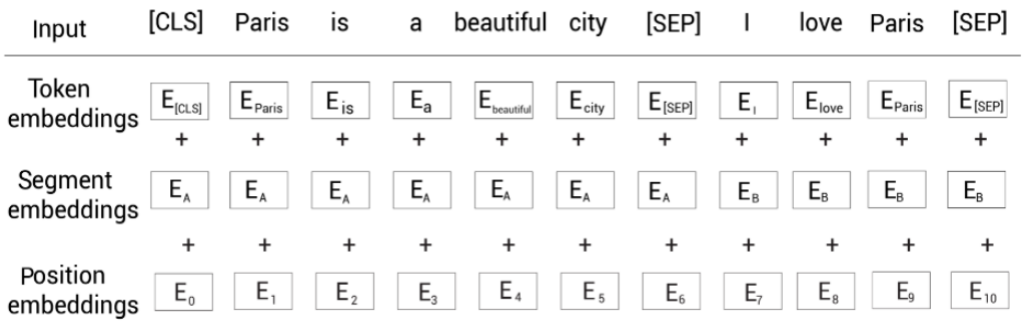
接下来，有一个位置嵌入层。

Transformer为了得到句子中单词的顺序信息，使用了位置编码，BERT本质上就是Transformer的编码器

所以需要提供句子中标记的位置信息，然后才能输入到BERT。

E 0表示标记[CLS]的位置嵌入

最终的表示



累加所有的嵌入表示作为BERT的input resperation

WordPiece分词器

BERT使用WordPiece分词器。

WordPiece分词器基于子词(subword)分词模式。

例子："Let us start pretraining the model."

使用WordPiece分词器来分词，会得到如下所示的标记：

tokens = [let, us, start, pre, ##train, ##ing, the, model]

在使用WordPiece分词器对句子进行分词时，单词`pretraining`被拆分为以下子词——`pre`、`##train`、`##ing`。

当使用WordPiece分词器分词，首先检测单词在词表(vocabulary)中是否存在，若存在，则作为标记；否则，我们将该单词拆分为一些子词，然后检查这些子词是否存在于词表。

如果某个子词存在于词表，那么将它作为一个标记；否则继续拆分子词，然后检查更小的子词是否存在于词表中。

这样，我们不断地拆分子词，并用词表检查子词，直到碰到单个字符为止。
接着增加`[CLS]`到句子的开始和`[SEP]`到句子的结尾：

tokens = [[CLS], let, us, start, pre, ##train, ##ing, the model, [SEP]]

如何预训练BERT模型

BERT模型的预训练基于两个任务：

- 屏蔽语言建模
- 下一句预测

屏蔽语言建模属于语言建模的一种，什么是语言建模？

语言建模

在语言建模任务中，我们训练模型根据给定的一系列单词来预测下一个单词。

语言建模分为两类：

- 自回归语言建模
- 自编码语言建模

自回归语言建模

将自回归语言建模归类为：

- 前向(左到右)预测
- 反向(右到左)预测

例子：考虑文本`Paris is a beautiful city. I love Paris`。

移除了单词`city`然后替换为空白符`__`：Paris is a beautiful __. I love Paris

现在，模型需要预测空白符实际的单词。

如果使用前向预测，那么我们的模型以从左到右的顺序阅读序列中的单词，直到空白符。

如果我们使用反向预测，那么我们的模型以从右到左的顺序阅读序列中的单词，直到空白符。

自回归模型天然就是单向的，意味着它们只会以一个方向阅读输入序列。

自编码语言建模

在预测时同时读入两个方向的序列。自编码语言模型天生就是双向的。

双向的模型能获得更好的结果

屏蔽语言建模

BERT是一个自编码语言模型，即预测时同时从两个方向阅读序列。