6.语言模型

1. 定义

- *通俗来讲*
- 语言模型评价一句话是否“合理”或“是人话”
- *数学上讲*
- P(今天天气不错) > P(今错不天天气)
- 语言模型用于计算文本的**成句概率**

2. 分类

- 1.统计语言模型 （SLM） S = Statistics
  ngram语言模型等
- 2.神经语言模型（NLM） N = Neural
  rnn语言模型等
- 3.预训练语言模型（PLM） P = Pre-train
  Bert、GPT等
- 4.大语言模型（LLM） L = Large
  ChatGPT等

3. N-gram语言模型

- 如何计算成句概率？
- 用S代表句子，w代表单个字或词
- $S = w_1 w_2 w_3 w_4 w_5 \ldots w_n$
- $P(S) = P(w_1, w_2, w_3, w_4, w_5 \ldots w_n)$
- 成句概率 -> 词W1~Wn按顺序出现的概率
- $P(w_1, w_2, w_3, \ldots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_n|w_1, \ldots, w_{n-1})$

a. N-gram计算方法

- 如何计算P(今天)?
- P(今天) = Count(今天) / Count_total　　语料总词数

- P(天气|今天) = Count(今天 天气) / Count(今天)

- P(不错|今天 天气) = Count(今天 天气 不错) / Count(今天 天气)

- 二元组：今天 天气　　2 gram
- 三元组：今天 天气 不错　　3 gram

b. 简化–马尔可夫假设

- 困难：句子太多了!
- 对任意一门语言，N-gram数量都非常庞大，无法穷举，需要简化

- 马尔科夫假设
- $P(w_n|w_1,\ldots,w_{n-1}) \approx P(w_n|w_{n-3},w_{n-2},w_{n-1})$
- 假设第n个词出现的概率，仅受其前面有限个词影响
- P(今天天气不错) = P(今)*P(天|今) *P(天|今天) *P(气|天天) *P(不|天气) *P(错|气不)

c. 面临的问题

- 平滑问题（smoothing）

- 理论上说，任意的词组合成的句子，概率都不应当为零

- 如何给没见过的词或ngram分配概率即为平滑问题

- 也称折扣问题（discounting）

-

- 1.回退（backoff）
- 当三元组a b c不存在时，退而寻找b c二元组的概率
- P(c | a b) = P(c | b) * Bow(ab)
- Bow(ab)称为二元组a b的回退概率
- 回退概率有很多计算方式，甚至可以设定为常数
- 回退可以迭代进行，如序列 a b c d
- P(d | a b c) = P(d | b c) * Bow(abc)
- P(d | bc) = P(d | c) * Bow(bc)
- P(d | c) = P(d) * Bow(c)

- P(word)不存在如何处理
- 加1平滑　add-one smooth
- 对于1gram概率P(word) = $\frac{Count(word)+1}{Count(total_{word})+V}$
- V为词表大小
- 对于高阶概率同样可以 $P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)+1}{c(w_{i-1})+V}$

加1平滑的分母为什么要加V（词表大小）？

因为这样能保证所有词的概率加起来的和仍然为1

d. 评价指标--困惑度计算

- 困惑度 perplexity

$$\begin{aligned} PP(S) &= P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{p(w_1 w_2 \dots w_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p(w_i | w_1 w_2 \dots w_{i-1})}} \end{aligned}$$

一般使用合理的目标文本来计算PPL，若PPL值低，则说明成句概率高，也就说明由此语言模型来判断，该句子的合理性高，这样是一个好的语言模型

- PPL值与成句概率成反比
- 另一种PPL，用对数求和代替小数乘积

$$PP(S) = 2^{-\frac{1}{N}\sum log(P(w_i))}$$

e. 代码实现

对每个n-gram存储

1. 出现n元组出现的总次数 ngram_count_dict
2. 出现n元组中每个词的条件概率（这个可以由(n-1)- gram得到）ngram_count_prob_dict

困惑度计算 Python

```python
import math
from collections import defaultdict


class NgramLanguageModel:
    def __init__(self, corpus=None, n=3):
        self.n = n
        self.sep = "_"      # 用来分割两个词，没有实际含义，只要是字典里不存在的符号都可以
        self.sos = "<sos>"    #start of sentence，句子开始的标识符
        self.eos = "<eos>"    #end of sentence，句子结束的标识符
        self.unk_prob = 1e-5  #给unk分配一个比较小的概率值，避免集外词概率为0
        self.fix_backoff_prob = 0.4  #使用固定的回退概率
        self.ngram_count_dict = dict((x + 1, defaultdict(int)) for x in range(n))
        self.ngram_count_prob_dict = dict((x + 1, defaultdict(int)) for x in range(n))
        self.ngram_count(corpus)
        self.calc_ngram_prob()

    #将文本切分成词或字或token
    def sentence_segment(self, sentence):
        return sentence.split()
    #return jieba.lcut(sentence)

    #统计ngram的数量
    def ngram_count(self, corpus):
        for sentence in corpus:
            word_lists = self.sentence_segment(sentence)
            word_lists = [self.sos] + word_lists + [self.eos]  #前后补充开始符和结尾符
            for window_size in range(1, self.n + 1):           #按不同窗长扫描文本
                for index, word in enumerate(word_lists):
                    #取到末尾时窗口长度会小于指定的gram，跳过那几个
                    if len(word_lists[index:index + window_size]) != window_size:
                        continue
                    #用分隔符连接word形成一个ngram用于存储
                    ngram = self.sep.join(word_lists[index:index + window_size])
                    self.ngram_count_dict[window_size][ngram] += 1
        #计算总词数，后续用于计算一阶ngram概率
        self.ngram_count_dict[0] = sum(self.ngram_count_dict[1].values())
        return
```

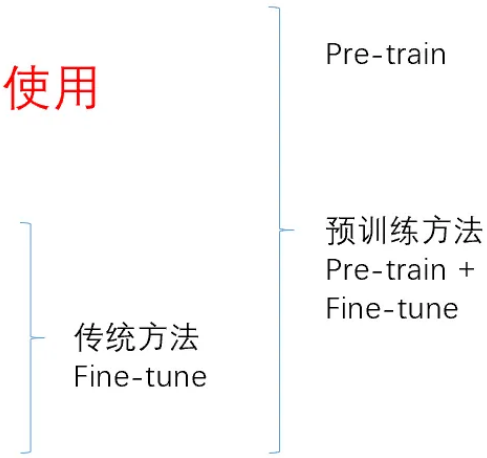
```
39
40
41      #计算ngram概率
42      def calc_ngram_prob(self):
43          for window_size in range(1, self.n + 1):
44              for ngram, count in self.ngram_count_dict[window_size].items()
:
45                  if window_size > 1:
46                      ngram_splits = ngram.split(self.sep)              #ngr
am        :a b c
47                      ngram_prefix = self.sep.join(ngram_splits[:-1])   #ngr
am_prefix :a b
48                      ngram_prefix_count = self.ngram_count_dict[window_size
 - 1][ngram_prefix] #Count(a,b)
49                  else:
50                      ngram_prefix_count = self.ngram_count_dict[0]     #cou
nt(total word)
51                  # word = ngram_splits[-1]
52                  # self.ngram_count_prob_dict[word + "|" + ngram_prefix] =
count / ngram_prefix_count
53                  self.ngram_count_prob_dict[window_size][ngram] = count / n
gram_prefix_count
54          return
55
56      #获取ngram概率，其中用到了回退平滑，回退概率采取固定值
57      def get_ngram_prob(self, ngram):
58          n = len(ngram.split(self.sep))
59          if ngram in self.ngram_count_prob_dict[n]:
60              #尝试直接取出概率
61              return self.ngram_count_prob_dict[n][ngram]
62          elif n == 1:
63              #一阶gram查找不到，说明是集外词，不做回退
64              return self.unk_prob
65          else:
66              #高于一阶的可以回退
67              ngram = self.sep.join(ngram.split(self.sep)[1:])
68              return self.fix_backoff_prob * self.get_ngram_prob(ngram)
69
70
71      #回退法预测句子概率
72      def calc_sentence_ppl(self, sentence):
73          word_list = self.sentence_segment(sentence)
74          word_list = [self.sos] + word_list + [self.eos]
75          sentence_prob = 0
76          for index, word in enumerate(word_list):
77              ngram = self.sep.join(word_list[max(0, index - self.n + 1):ind
ex + 1])
78              prob = self.get_ngram_prob(ngram)
              # print(ngram, prob)
```

```
            sentence_prob += math.log(prob)
        return 2 ** (sentence_prob * (-1 / len(word_list)))



if __name__ == "__main__":
    corpus = open("sample.txt", encoding="utf8").readlines()
    lm = NgramLanguageModel(corpus, 3)
    print("词总数:", lm.ngram_count_dict[0])
    print(lm.ngram_count_prob_dict)
    print(lm.calc_sentence_ppl("c d b d b"))
```

4. 预训练语言模型

- 1.收集海量无标注文本数据
- 2.进行模型预训练，并在任务模型中使用

- 3.设计模型结构
- 4.收集/标注训练数据
- 5.使用标注数据进行模型训练
- 6.真实场景模型预测



有些任务你从来没见过，但是你也不需要从头开始学习

例如：你要判断说中文的一个人有没有礼貌，你不需要从头开始学习中文

因此需要训练一个通用的模型框架，再根据下游任务进行fine-tune

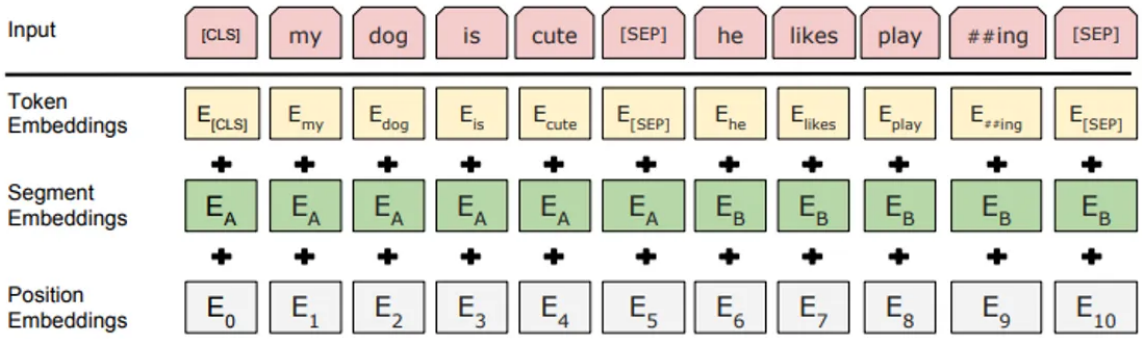
1. 预训练和词向量有些类似，都可以在进行下游任务前进行训练
2. 但也有不同，词向量是静态的，BERT是动态的，可以结合语境

我喜欢吃苹果　　苹果和华为哪个牌子好

5. BERT

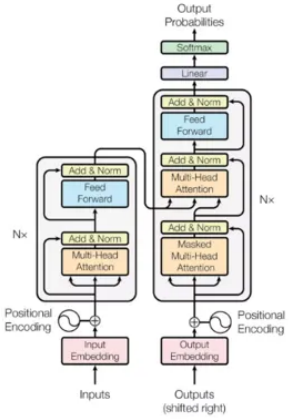
在做词embedding时，一般的NLP是做一层embedding，即： 10×1 转为 10×128

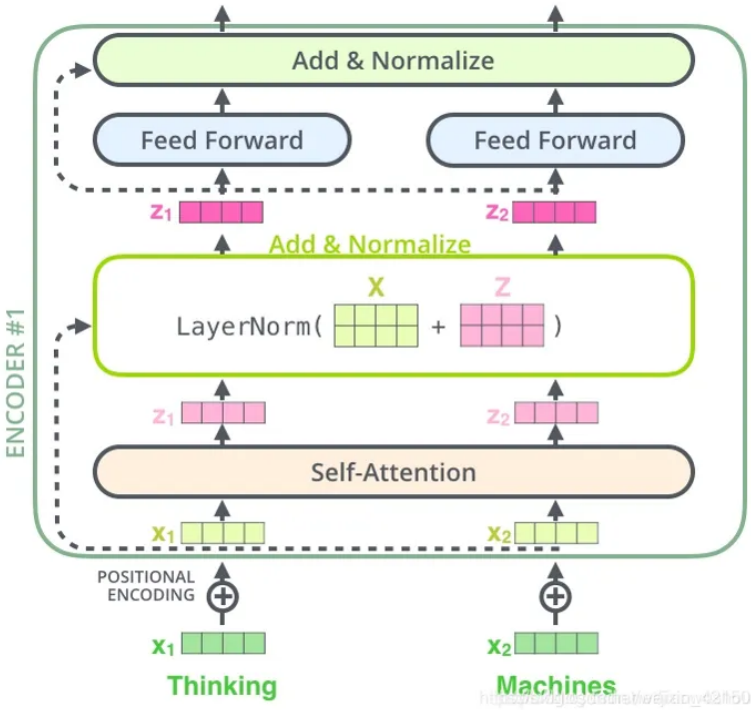
但BERT在word2vec时，考虑了切分句子和位置信息



- 加入 [CLS] [SEP] 来标记文本起始位置
- Segment embedding 判断来源语句
- Position embedding 带入语序信息
- 加和后会做Layer Normalization

BERT的模型主体结构使用Google自己在17年提出的Transformer结构





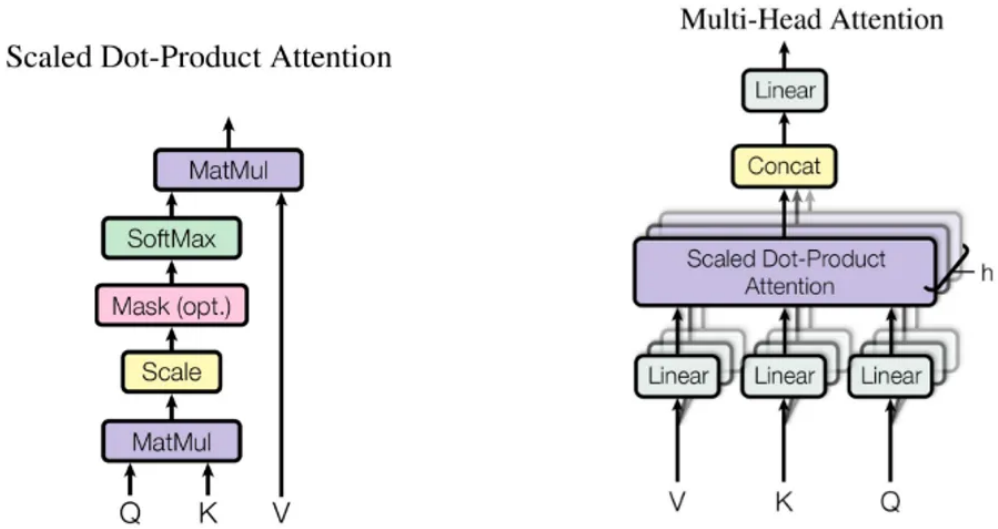
	今	天	天	气	不	错
今	0.12	0.123	-1.324	0.571	-0.669	0.982
天						
天						
气						
不						
错						

逐行softmax

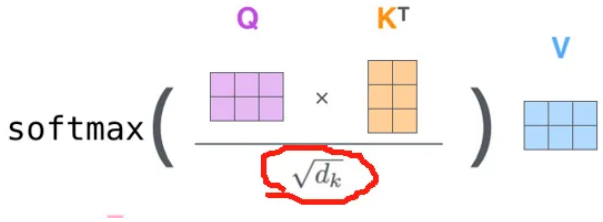
Q * K.T

Q*K：实际上得到的是句子中每个字或词与其他字或词的重要度，相比RNN，能够得到长距离的依赖关系

a. 关于Multi-Head



相当于同时训练多个模型，最后拼起来，利用模型集成的思想。



关于这个 d_k ，目的是除以一个数，使得过softmax后，数值不要太接近于one-hot向量，因为我们倾向于认为词和词之间是有联系的。

b. 训练方式

1. Mask Language Model （MLM）
2. Next Sentence Prediction （NSP）

- 1.完形填空
- Mask Language Model
- Bidirectional Language Model
- 依照一定概率，用[mask]掩盖文本中的某个字或词

```
Input Sequence  : The man went to [MASK] store with [MASK] dog
Target Sequence :                  the                 his
```

- 2.句子关系预测
- Next Sentence Prediction
- [CLS] 师徒四人历经艰险[SEP] 取得真经[SEP] -> True
- [CLS] 师徒四人历经艰险[SEP] 火烧赤壁[SEP] -> False

c. BERT的优势

- 1、通过预训练利用了海量无标注文本数据
- 2、相比词向量，BERT的文本表示结合了语境
- 3、Transformer模型结构有很强的拟合能力，词与词之间的距离不会造成关系计算上的损失
- 4、效果大幅提升

d. BERT的劣势

- 1.预训练需要数据，时间，和机器（开源模型缓解了这一问题）
- 2.难以应用在生成式任务上
- 3.参数量大，运算复杂，满足不了部分真实场景性能需求
- 4.没有下游数据做fine-tune，效果依然不理想