



AI大模型健康问诊系统

华信培训课程

课程目录

01 项目背景

02 相关知识

03 项目开发与部署





01/ 项目背景



- 随着医疗需求增长与 AI 技术发展，传统问诊模式面临效率低、资源分布不均等挑战。AI 大模型凭借海量数据处理与智能分析能力，可突破时间空间限制，快速响应健康咨询、初步分诊等需求，辅助提升基层医疗服务可及性，缓解医疗资源压力，为用户提供个性化、便捷化的健康管理方案，推动医疗服务数字化转型。



● 设计内容

- 本项目是基于自然语言处理技术、依托人工智能大模型而构建的智能健康问答系统，能够模拟人类的自然语言交流，与用户进行对话和互动，机器能够理解用户的问题或指令，并给出相应的回答或建议。

● 设计要求

- 智能健康问答系统需要进行行业数据收集，根据数据的特征进行相应的文字处理，选择适合的大模型进行模型训练及模型验证，根据验证结果进行参数的优化，并以图表的形式展现调优对比。

- 本项目基于医疗领域数据构建智能医疗问答系统，目的是为用户提供准确、高效、优质的医疗问答服务。

智能健康问诊

流涕不止怎么办

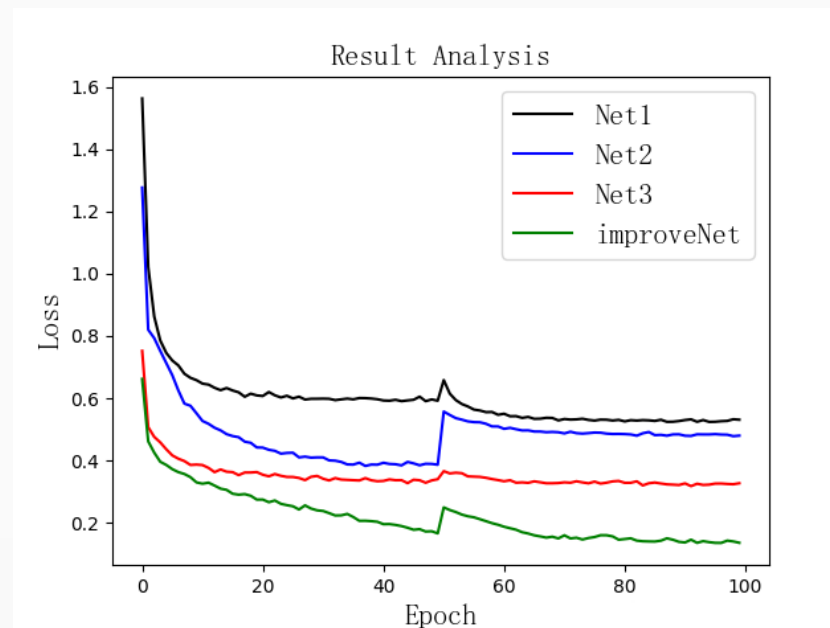
多考虑是属于鼻炎的情况

儿童呼吸道感染的临床表现有些什么？

咽部细菌分布；打喷嚏；咳嗽；顽固性咳嗽

鼓膜外伤的症状是什么？

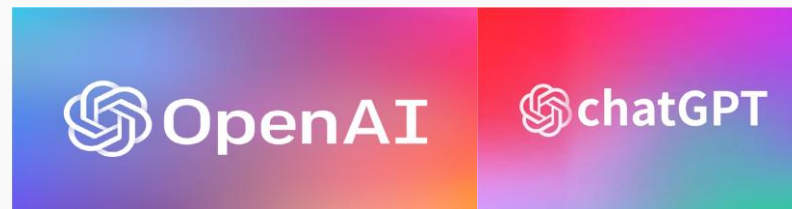
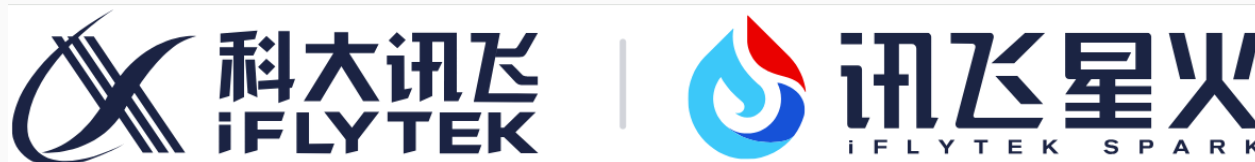
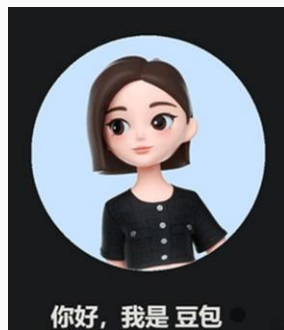
耳溢液；耳漏；耳痛；耳聋；听力减退





- **一款聊天机器人**

- 聊天机器人是一种**基于自然语言处理技术**的智能对话系统，能够模拟人类的自然语言交流，与用户进行对话和互动。聊天机器人能够理解用户的问题或指令，并给出相应的回答或建议。其目标是提供友好、智能、自然的对话体验。





02/ 相关知识





- 人工智能
- 机器学习
- 深度学习
- 自然语言处理
- Transformer模型
- GPT大模型



- **机器学习是从数据中自动分析获得模型，并利用模型对未知数据进行预测**
- **常用算法**
 - 分类算法：k-近邻算法、贝叶斯分类、决策树与随机森林、逻辑回归
 - 回归算法：线性回归、岭回归
 - 无监督学习：聚类k-means



- **深度学习是机器学习的一个领域（人工神经网络）发展而来的**
- **深度学习算法试图从数据中学习高级功能，这是深度学习的一个非常独特的部分。因此，减少了为每个问题开发新特征提取器的任务。适合用在难提取特征的图像、语音、自然语言处理领域。**
- **算法代表：神经网络**



- **计算机视觉 (Computer Vision, CV)**

- 图像分类
- 目标检测
- 语义分割
- 实例分割
- 视频分类
- 人体关键点检测
- 场景文字识别
- 目标跟踪任务
- . . .



- **自然语言处理 (Natural Language Processing, NLP)**
 - 智能客服
 - 智能问答系统
 - 文本挖掘
 - 机器翻译与跨语言交流
 - 摘要生成
 - AIGC




- **语言模型：**

- 就是一个用来估计文本概率分布的数学模型，它可以帮助我们了解某个文本序列在自然语言中出现的概率，因此也就能够根据给定的文本，预测下一个最可能出现的单词。
- 语言模型关注的是一段上下文中单词之间的相关性，以保证模型所生成的文本序列是合理的。
- 语言模型被广泛应用于机器翻译、语音识别、文本生成、对话系统等多个NLP领域。

- 假设语料库足够大，句子2曾经在这个语料库中出现过，那么AI会说：句子2更好。
- 当然，概率高的事情不是百分之百正确，这也是强大的大语言模型偶尔也会出错的原因。

几个词

咖哥 一本书 学 零基础 机器学习 写了 

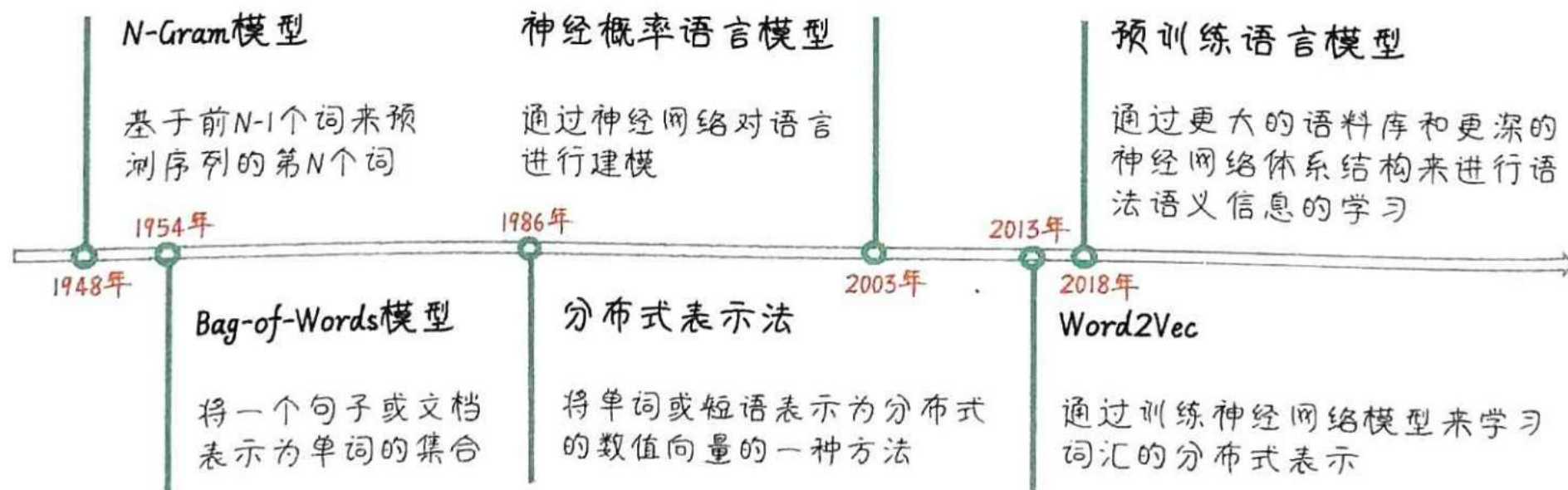
句子1 咖哥零基础学一本书写了机器学习 ✗

句子2 咖哥写了一本书零基础学机器学习 ✓ 更像句子

句子2的概率 > 句子1的概率

句子2正确的概率比较高

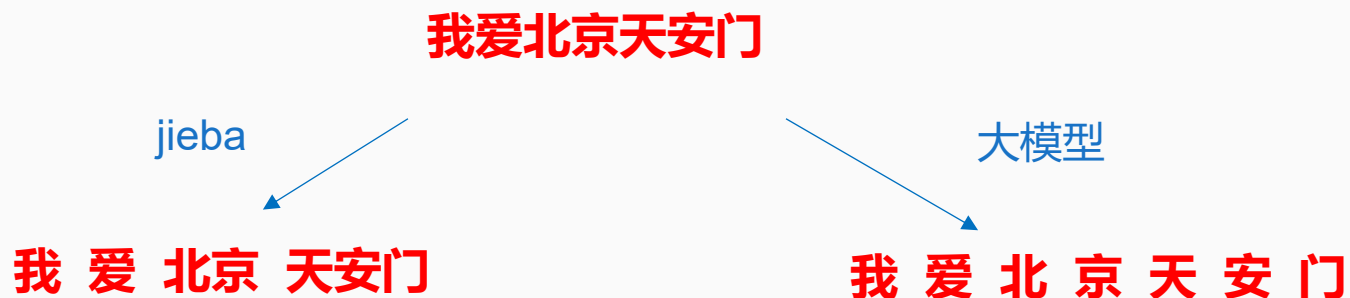
- 上半部分是语言模型技术的发展
- 下半部分是词向量技术的发展



统计语言模型发展的里程碑

- **token: 子词/分词**

- 通过分词工具，把语料，也就是一个个句子，切成了能够被语言模型读取并处理的一个个元素
- 分词工具：英文分词工具（NLTK、spaCy等），中文分词工具（jieba）





- N-Gram：语言模型的雏形
- 在N-Gram模型中，通过**将文本分割成连续的N个词的组合**（即：N-Gram），来近似地**描述词序列的联合概率**。
- 主要用于语言建模、文本生成、语音识别等任务中



- 1、将给定的文本分割成连续的N个词的组合
- 2、统计每个N-Gram在文本中出现的次数，也就是词频
- 3、利用条件概率公式计算，给定前N-1个词时，下一个词出现的概率
- 4、使用这个条件概率来预测下一个词出现的可能性，多次迭代这个过程可以生成句子

```
corpus = [  
    '我喜欢吃苹果',  
    '我喜欢吃香蕉',  
    '她喜欢吃葡萄',  
    '他不喜欢吃香蕉',  
    '他喜欢吃苹果',  
    '她喜欢吃草莓'  
]
```

原始语料库

词频:

```
我: {'喜': 2}  
喜: {'欢': 6}  
欢: {'吃': 6}  
吃: {'苹': 2, '香': 2, '葡': 1, '草': 1}  
苹: {'果': 2}  
香: {'蕉': 2}  
她: {'喜': 2}  
葡: {'萄': 1}  
他: {'不': 1, '喜': 1}  
不: {'喜': 1}  
草: {'莓': 1}
```

计算每个Bigram在语料库中的词频
图中为N=2

计算每个Bigram的出现频率

Bigram概率:

```
我: {'喜': 1.0}  
喜: {'欢': 1.0}  
欢: {'吃': 1.0}  
吃: {'苹': 0.3333, '香': 0.3333, '葡': 0.1667, '草': 0.1667}  
苹: {'果': 1.0}  
香: {'蕉': 1.0}  
她: {'喜': 1.0}  
葡: {'萄': 1.0}  
他: {'不': 0.5, '喜': 0.5}  
不: {'喜': 1.0}  
草: {'莓': 1.0}
```



- **优点：计算简单。**
- **缺点：只考虑它前面的N-1个词，无法捕捉到距离较远的词之间的关系。**

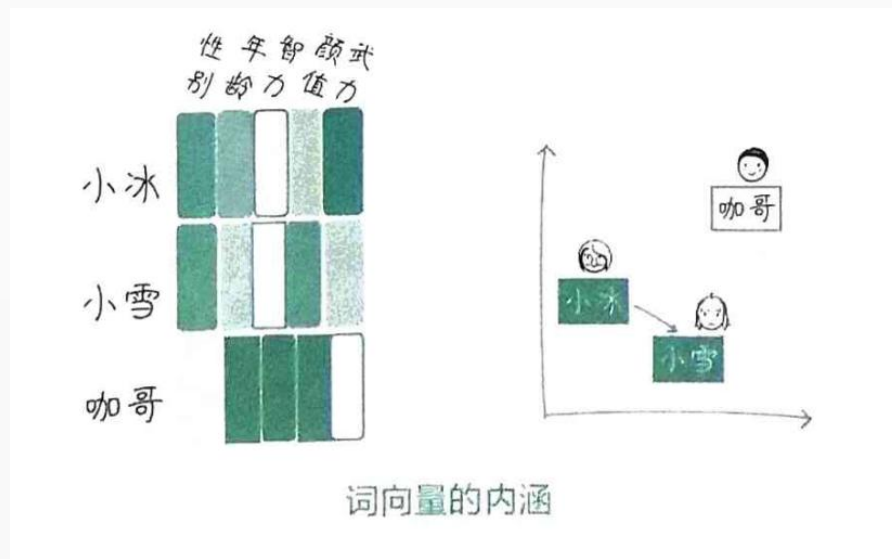


- Bag-of-Words: 将文本中的词看作一个个独立的个体，不考虑它们在句子中出现的顺序，只关心每个词出现的频率。
- 主要用于文本分类、情感分析、信息检索等任务中。

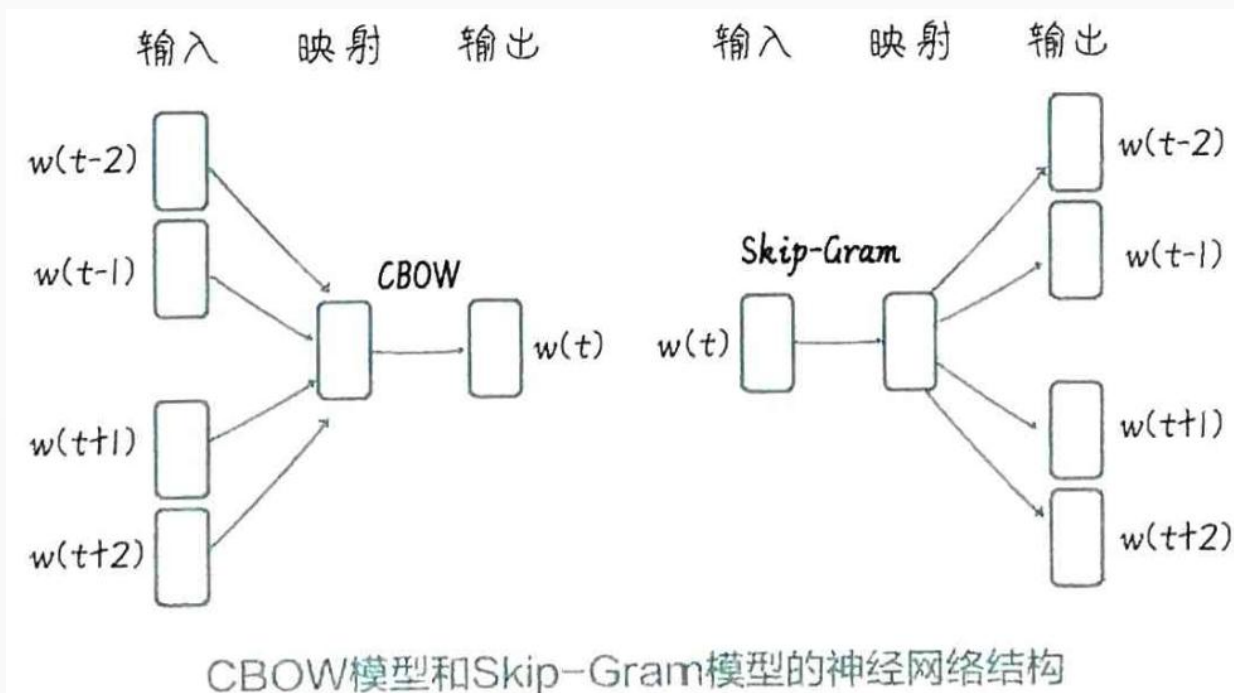
```
sentences = [  
    '小明喜欢吃苹果',  
    '苹果是小白非常非常喜欢的水果'  
]
```

```
{ '喜欢': 1, '小明': 1, '苹果': 1, '吃': 1 }  
{ '非常': 2, '的': 1, '苹果': 1, '小白': 1, '水果': 1, '喜欢': 1, '是': 1 }
```

- 词向量/词嵌入 (Word Embedding)：是一种寻找词和词之间的相似性的NLP技术。
- 词嵌入不是一种语言模型，而是一个技术。用于捕捉词语之间的关系，为下游的NLP任务提供丰富的表示。
- 词嵌入和词向量不完全相同，词嵌入是将词映射到向量空间的过程或方法，通过这个过程或方法来生成词向量（即：一个词对应的实际向量表示）。
- 常用算法：Word2Vec



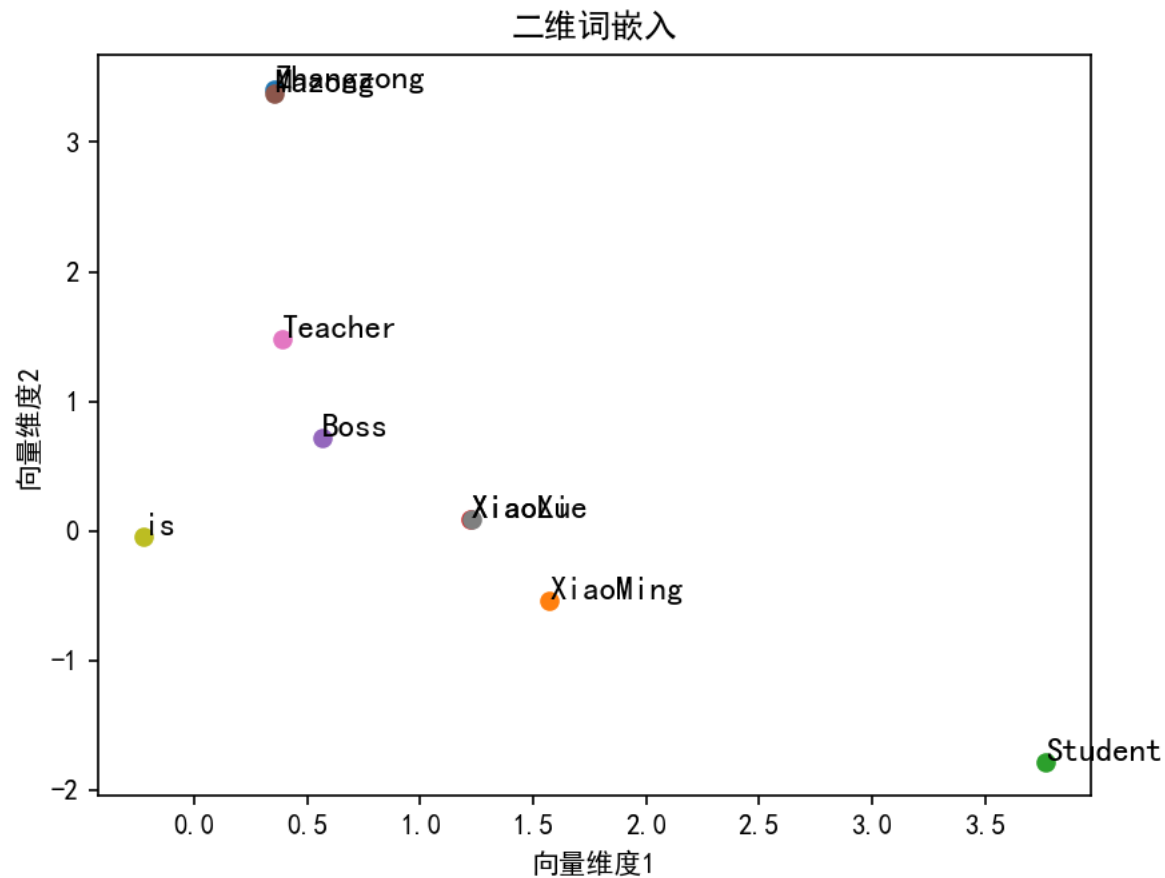
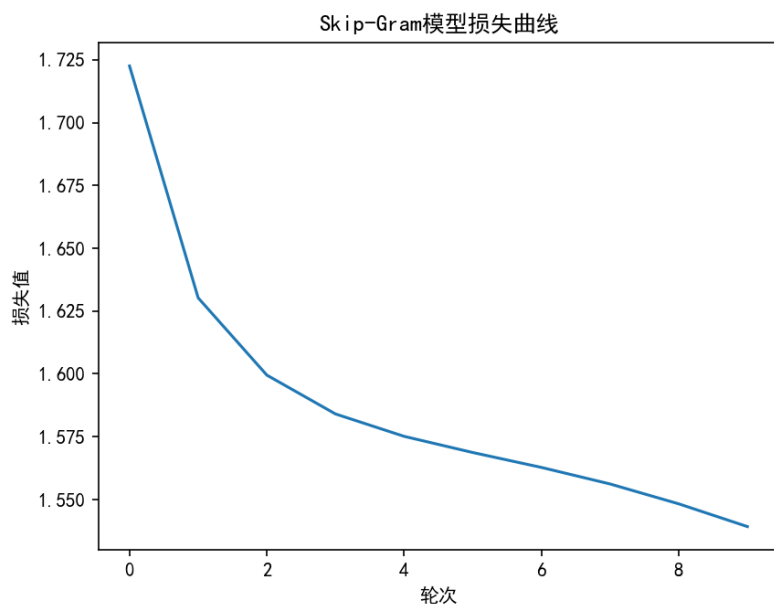
- Word2Vec是通过训练一个神经网络模型来学习词嵌入，模型的任务就是基于给定的上下文词来预测目标词（ CBOW ），或者基于目标词来预测上下文词（ Skip-Gram ）。
- 两种实现方式：
 - CBOW （ Continuous Bag of Words ）：连续词袋模型
 - Skip-Gram：跳字模型



Skip-Gram算法实现结果



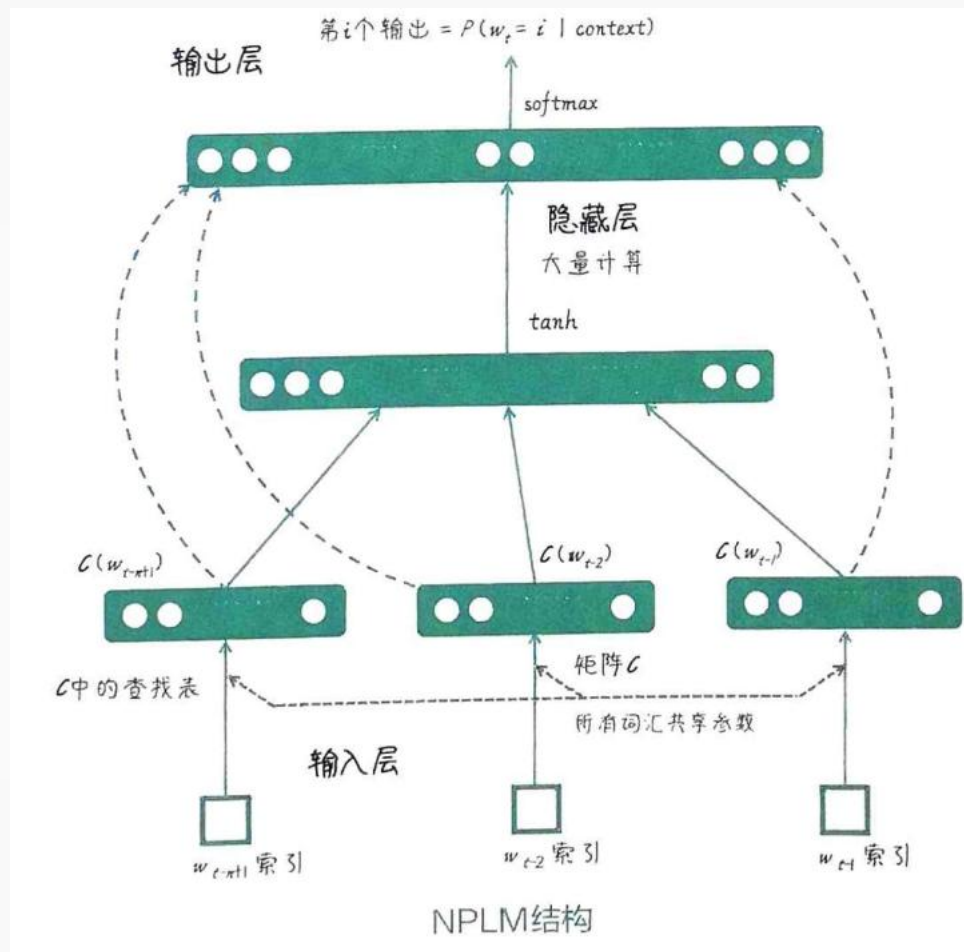
```
sentences = [  
    'XiaoMing is Teacher',  
    'Mazong is Boss',  
    'Zhangzong is Boss',  
    'XiaoLi is Student',  
    'XiaoXue is Student'  
]
```





- **优点：**是一种更先进的分布式表示方法，它通过学习单词在上下文中的共现关系来生成低维、密集的词向量，捕捉单词之间的语义和语法关系，并在向量空间中体现这些关系。
- **局限性：**
 - 词向量的大小是固定的，限制了模型捕捉一词多义的能力。
 - 无法处理未知词汇

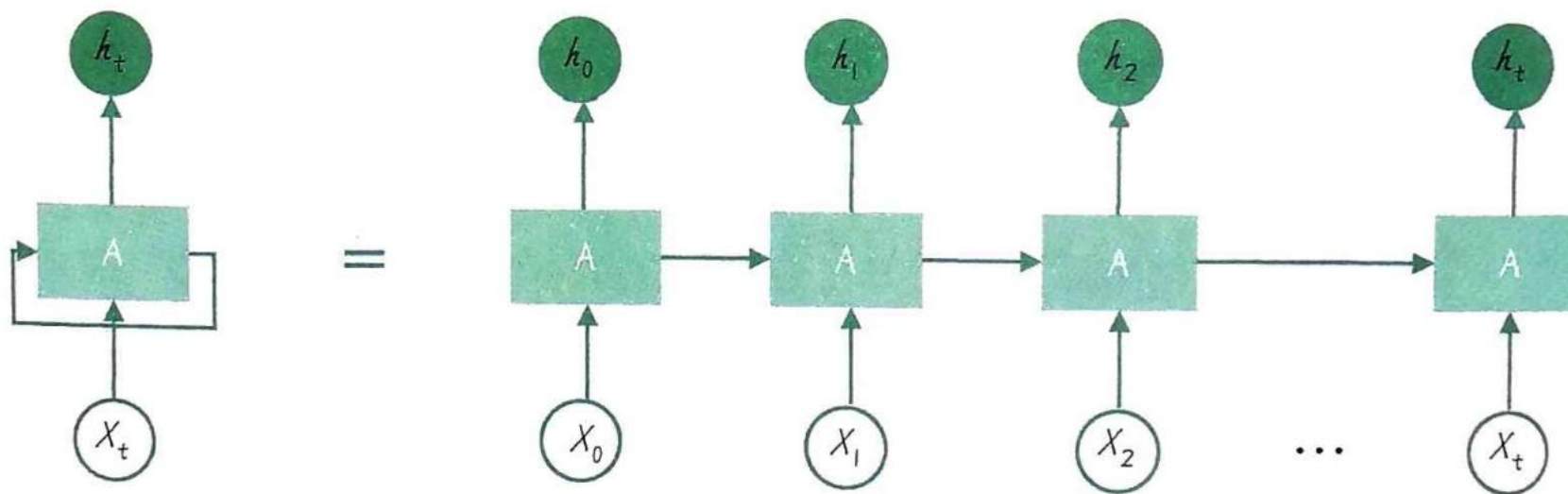
- 神经概率语言模型（NPLM）：将神经网络应用于语言模型，能够更有效的处理稀疏矩阵和长距离依赖问题。
- NPLM的结构包括三个主要部分：
 - 输入层：将词汇映射到连续的向量空间
 - 隐藏层：通过非线性激活函数学习词与词的关系
 - 输出层：通过softmax函数产生下一个词的概率





- **历史意义：在于开创性地把神经网络技术引入NLP领域**
- **优点：**
 - 自动学习复杂的特征表示，减少手工特征工程
 - 可以对大量数据进行高效的处理
 - 具有强大的拟合能力
- **不足之处：**
 - 模型结构简单，表达能力受限
 - 窗口大小/输入序列长度固定，限制了模型处理不同长度上下文的能力
 - 缺乏长距离依赖捕捉
 - 词表固定，无法处理训练集中未出现的词汇

- 循环神经网络 (RNN)：能够很好的处理变长序列的长距离依赖问题，以及稀有词汇和词汇表以外的词汇
- 核心思想：利用“循环”机制，将网络的输出反馈到输入，这样可以在处理数据时保留前面的信息，从而捕捉序列中的长距离依赖关系



RNN基本架构



- **LSTM (长短期记忆网络) 和GRU (门控循环单元) 等：都属于改进型RNN，为了解决训练过程中的梯度消失或者梯度爆炸等问题**
- **不足之处 (在RNN时代)：**
 - 缺乏大规模数据
 - 优化算法发展不足
 - 模型表达能力不足
 - RNN在同时处理输入和输出序列 (既负责编码，又负责解码) 时，容易出现信息损失

- 序列到序列(Sequence to Sequence, Seq2Seq)
- 起源于2014年, 伊利亚·苏茨克维(ChatGPT首席科学家)等人发表的一篇文章
- Seq2Seq架构通过编码器 (Encoder) 和解码器 (Decoder) 来分离对输入和输出序列的处理
- 在编码器和解码器中, 分别嵌入相互独立的RNN, 可以有效的解决编解码过程中的信息损失问题
- Transformer基础架构



编码器-解码器架构和通信模型十分相似

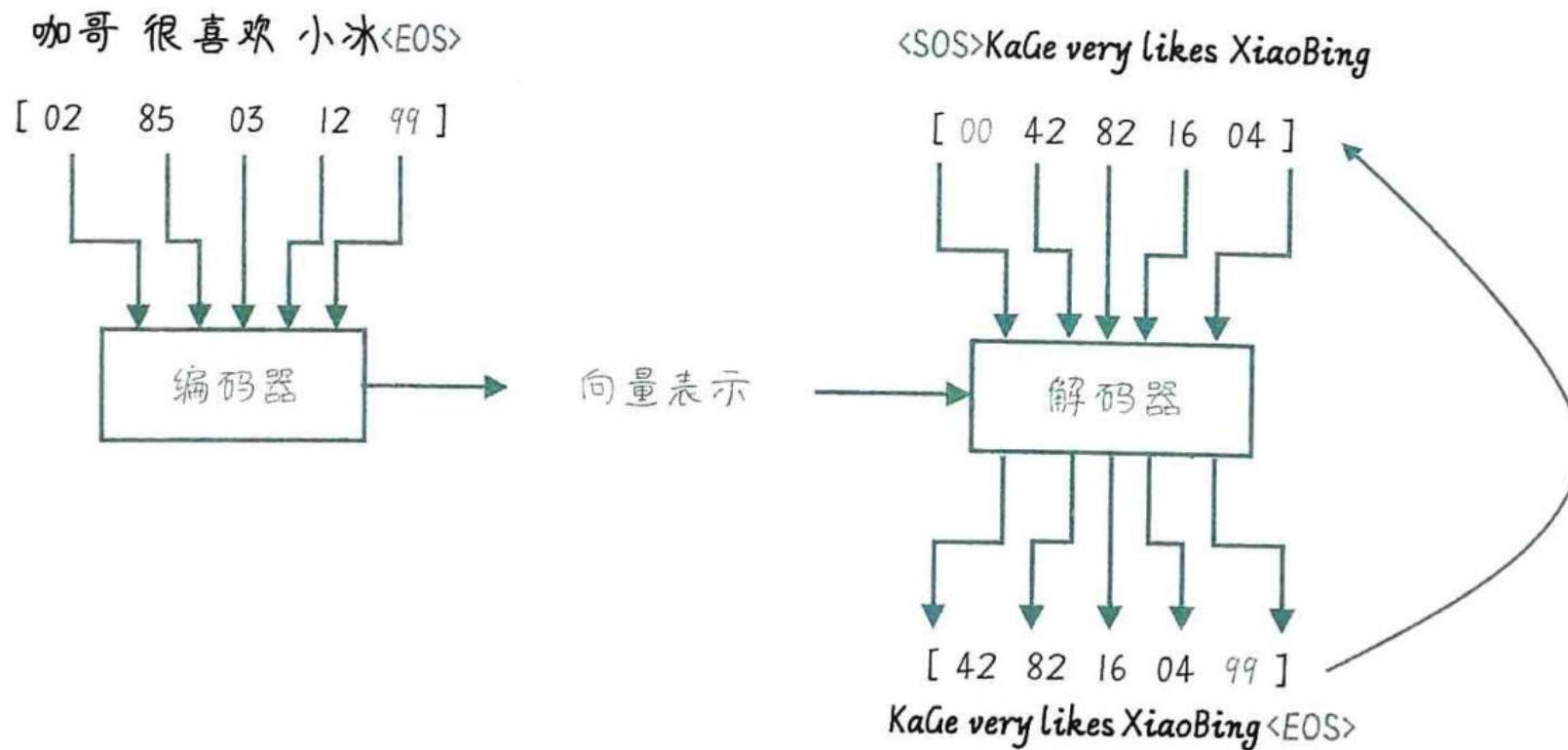


- **编码器 (Encoder)**

- 负责将输入序列（源语言文本）转换为固定大小的向量
- 采用RNN、LSTM或GRU等模型，逐个处理输入序列中的元素（单词或字符）
- 最后生成一个上下文向量，这个向量包含整个输入序列的信息

- **解码器 (Decoder)**

- 负责将编码器生成的上下文向量转换为输出序列（目标语言文本）
- 也是采用RNN、LSTM或GRU等模型，使用来自编码器的上下文向量作为其初始隐藏状态，并逐个生成输出元素



Seq2Seq架构将输入序列转换成向量表示，然后将该向量表示转换成输出序列



- 编码器的输入序列和解码器的输出序列的**长度可以是不同的**，所以更适合用于处理翻译、问答、文本摘要等生成类型的NLP任务
- 编码器和解码器可以采用RNN、LSTM或其他循环神经网络的变体，也可以采用其他形式的神经网络来处理
- 可以使用注意力机制增强模型性能，让解码器在生成输出时关注输入序列的不同部分

- 编码器 (Encoder)

```
# 编码器类
class Encoder(nn.Module): 2 usages
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
    def forward(self, input, hidden):
        embedding = self.embedding(input)
        output, hidden = self.rnn(embedding, hidden)
        return output, hidden
```

- 解码器 (Decoder)

```
class Decoder(nn.Module): 2 usages
    def __init__(self, hidden_size, ouput_size):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(ouput_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size, ouput_size)
    def forward(self, input, hidden):
        embedding = self.embedding(input)
        output, hidden = self.rnn(embedding, hidden)
        output = self.out(output)
        return output, hidden
```

- Seq2Seq架构

```
# 将上述编码器解码器组合成Seq2Seq架构
class Seq2Seq(nn.Module): 2 usages
    def __init__(self, encoder, decoder):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
    def forward(self, enc_input, hidden, dec_input):
        enc_output, enc_hidden = encoder(enc_input, hidden)
        dec_output, dec_hidden = decoder(dec_input, enc_hidden)
        return dec_output
```



- Seq2Seq架构的**核心在于编码器和解码器的设计**
- 基于RNN的Seq2Seq存在一些缺点
 - 难以处理长序列和复杂的上下文相关性
- 改进：向编码器-解码器架构间引入**注意力机制**



- **注意力机制 (Attention Mechanism)** 是深度学习领域中的一种技术，它模仿人类视觉注意力的功能，**使模型能够集中于输入数据中最重要的部分。**
- **关键特点：**
 - 选择性聚焦
 - 上下文建模
 - 可解释性
 - 灵活性
 - 并行处理



- **实现方式:**
 - 点积注意力
 - 缩放点积注意力
 - 自注意力
 - **多头自注意力**: Transformer的一个组成部分
 - 全局注意力
 - 因果注意力
 -

观众	喜欢浪漫	喜欢动作	喜欢恐怖
小明	5	1	1
小白	1	5	1
小花	1	1	5

两组数据之间的关系？

电影	浪漫	动作	恐怖
异形	0.2	4	5
叶问	1	5	1

- 实现步骤:

- 使用点积计算原始权重: 也就是计算x1中每个位置与x2中每个位置之间的相似度得分

```
raw_weight = torch.matmul(x1, x2.transpose(dim0: 0, dim1: 1))
```

- 使用softmax函数对原始权重进行归一化, 让它们的和为1, 得到注意力权重

```
attn_weights = F.softmax(raw_weight, dim=1)
```

- 将注意力权重与x2相乘, 计算加权和, 得到的结果张量形状与x1相同, 可以看做x1关注了x2之后的新x1

```
attn_output = torch.matmul(attn_weights, x2)
```



- 在深度学习模型中，**点积的值可能会变得非常大**，尤其是当特征维度较大时，会导致softmax函数可能会在一个非常陡峭的区域内运行
- 为确保softmax函数在一个较为平缓的区域内工作，在计算注意力权重之前，将原始权重除以一个缩放因子，通常这个**缩放因子是输入特征维度的平方根**

```
# 缩放点积注意力  
raw_weight = raw_weight / (x1.size(-1) ** 0.5)
```



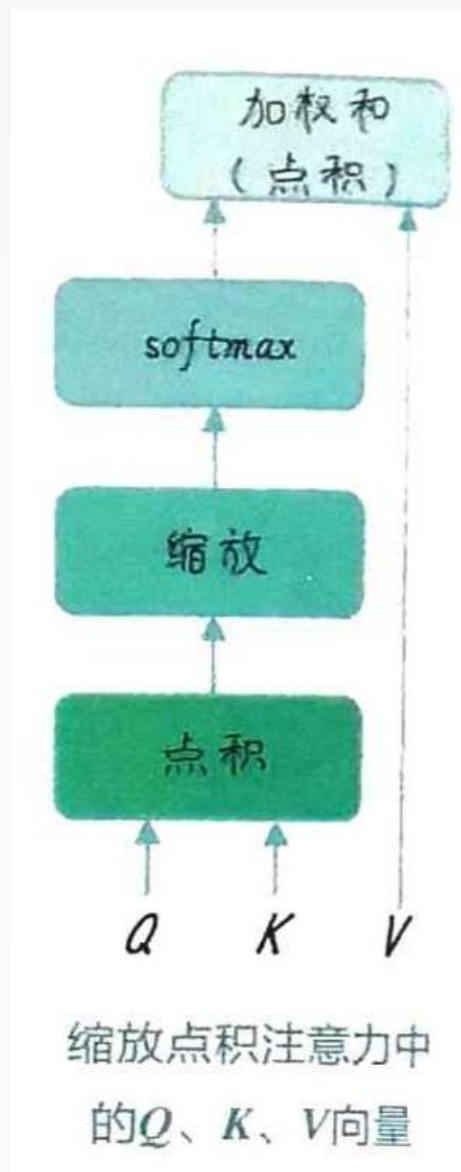
- 在Seq2Seq架构中:
 - **x1**: 是**解码器**在各个step的隐藏状态
 - **x2**: 是**编码器**在各个step的隐藏状态
 - 也就是说, 解码器需要对编码器进行注意

```
class Attention(nn.Module):  
    def __init__(self):  
        super(Attention, self).__init__()  
    def forward(self, decoder_context, encoder_context):  
        scores = torch.matmul(decoder_context, encoder_context.transpose(-2, -1))  
        attn_weight = F.softmax(scores, dim=-1)  
        context = torch.matmul(attn_weight, encoder_context)  
        return context, attn_weight
```

- 在解码器类的初始化部分增加注意力层
- 用前向传播方法在该层上计算注意力上下文向量，以便在解码过程中使用注意力权重

```
# 定义解码器
class DecoderWithAttention(nn.Module): 2 usages
    def __init__(self, hidden_size, output_size):
        super(DecoderWithAttention, self).__init__()
        # self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.attention = Attention()
        self.out = nn.Linear(2 * hidden_size, output_size)
    def forward(self, dec_input, hidden, enc_output):
        embedded = self.embedding(dec_input)
        output, hidden = self.rnn(embedded, hidden)
        context, attn_weights = self.attention(output, enc_output)
        # 将上下文向量与解码器的输出拼接
        output = torch.cat(tensors=(output, context), dim=-1)
        output = self.out(output)
        return output, hidden, attn_weights
```

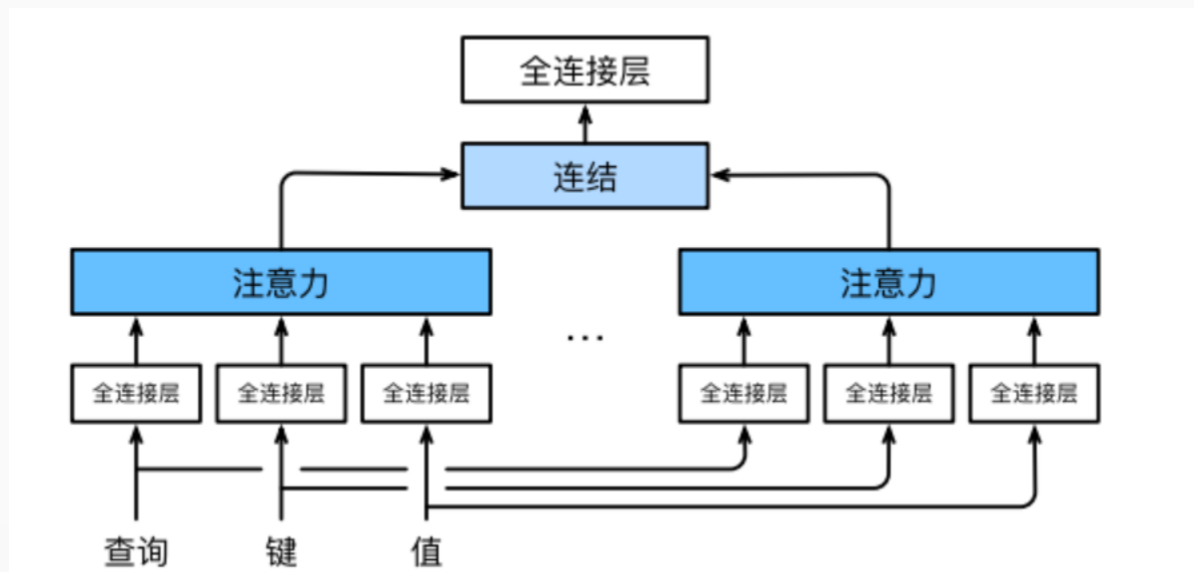
- **Q: 查询 (Query)**
 - 是指当前需要处理的信息
- **K: 键 (Key)**
 - 来自输入序列的一组表示。与Q计算注意力权重
- **V: 值 (Value)**
 - 来自输入序列的一组表示
 - 用于与softmax之后的结果进行计算加权和





- 自注意力就是自己对自己的注意
- 允许模型在同一序列中的不同位置之间建立依赖关系

- 注意力机制的一种扩展，可以帮助模型从不同的表示子空间捕获输入数据的多种特征
- 在计算注意力权重和输出时，会对Q、K、V向量分别进行多次线性变换，从而获得不同的头 (Head)
- 计算过程：
 - 初始化
 - 线性变换
 - 缩放点积注意力
 - 合并





- **优势:**

- 可以在不同的表示子空间中捕捉输入数据的多种特征，从而**提高模型在处理长距离依赖和复杂结构时的性能。**
- 在自然语言处理、计算机视觉等领域的任务中表现较好

- 注意力掩码的作用是避免模型在计算注意力分数时，将不相关的单词考虑进来。
- 掩码操作可以防止模型学习到不必要的信息
- 填充掩码 (Padding Mask)

雕	龙	一	拍	PAD	PAD	PAD
顶	呱	呱	PAD	PAD	PAD	PAD
呀	!	PAD	PAD	PAD	PAD	PAD
顶	呱	呱	PAD	PAD	PAD	PAD
中	文	模	型	全	靠	它

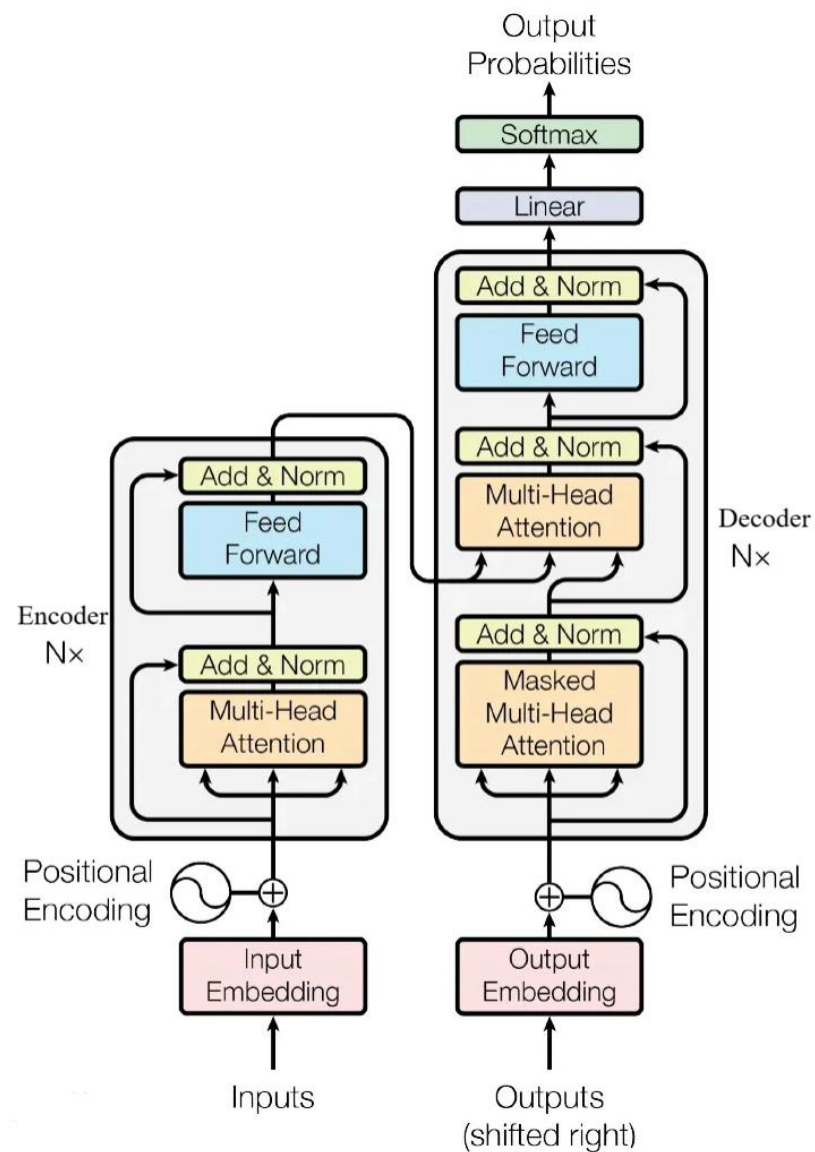
序列的Padding



- Transformer的起源可以追溯到2017年，谷歌大脑（Google Brain）团队的Vaswani等人在论文“Attention is All You Need”《你只需要注意力》中提出的结构。
- **核心是自注意力机制**
- 摒弃了RNN和LSTM中的循环结构
- 采用了全新的**编码器--解码器架构**
- 已成为NLP领域的代表性技术，并在计算机视觉、语音识别等其他AI领域也取得了显著的成绩
- GPT的技术内核

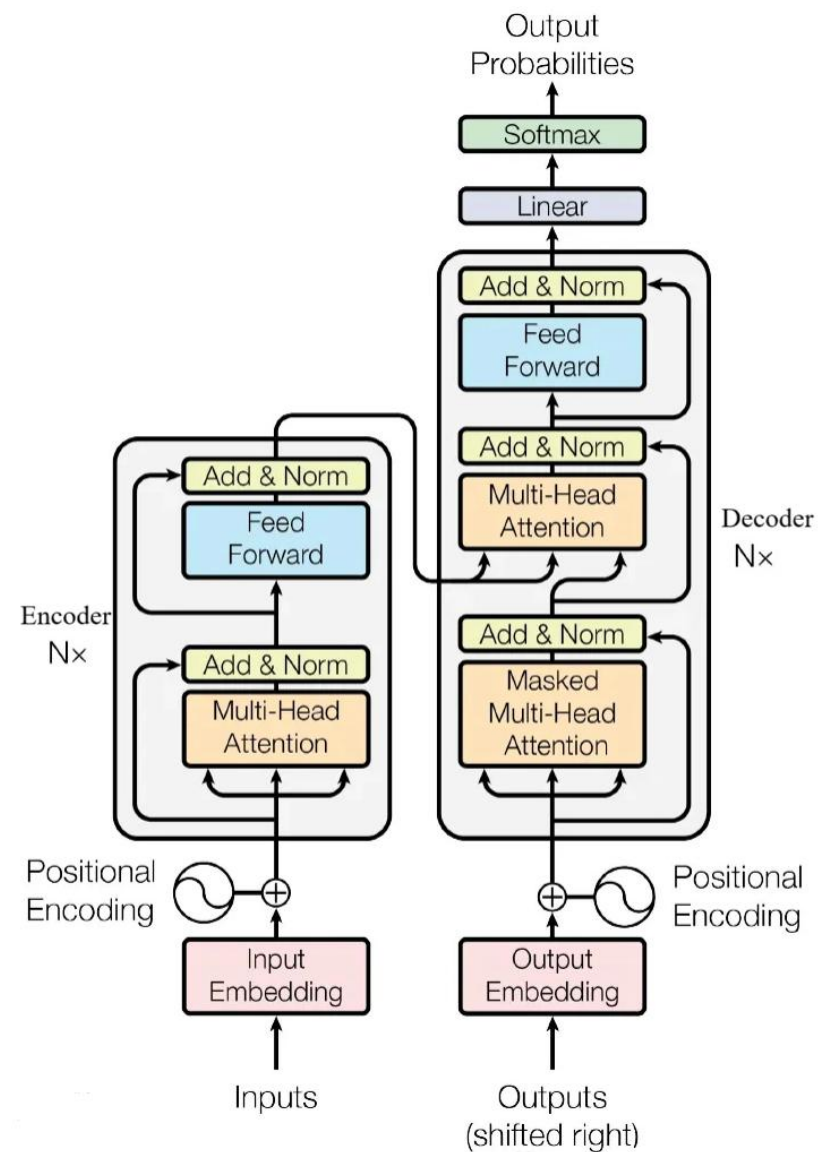
● 编码器：

- 1、把需要处理的文本序列转换为一个输入词嵌入向量 (Input Embedding)
- 2、为词向量添加位置编码 (Positional Encoding)
- 3、词向量和位置编码结合起来进入编码器的第一层，这一层先进行多头自注意力计算 (Multi-Head Attention)
- 4、多头自注意力的输出与原始输入相加 (残差连接 Residual Connection)，然后经过归一化 (Layer Normalization) 处理。这个模块在论文中被简称为Add & Norm。



● 编码器：

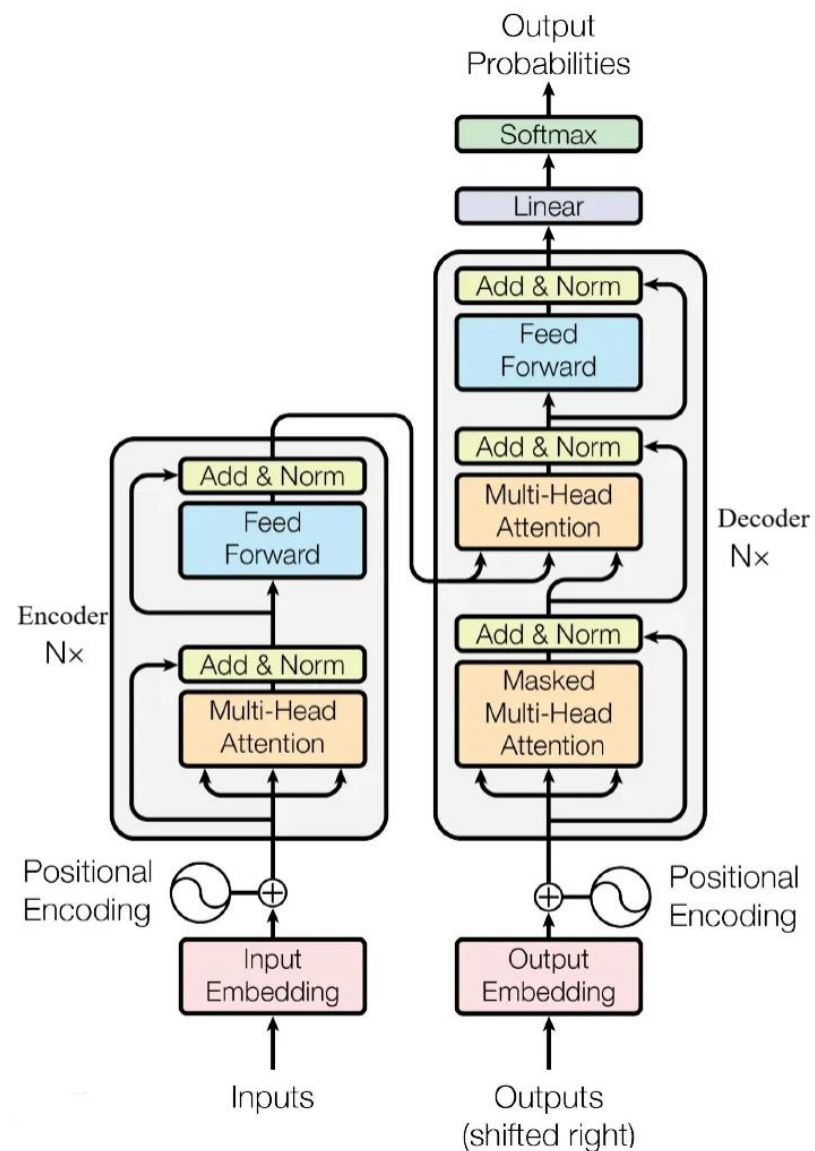
- 5、归一化的结果，经过一个前馈神经网络（Feed Forward），该层是一个包含两个线性层和一个激活函数的简单网络。
- 6、前馈神经网络的输出与自注意力机制的结果再次相加，并进行残差和归一化的处理，最终得到编码器的输出。
- 3-6的过程会执行N次



● 解码器:

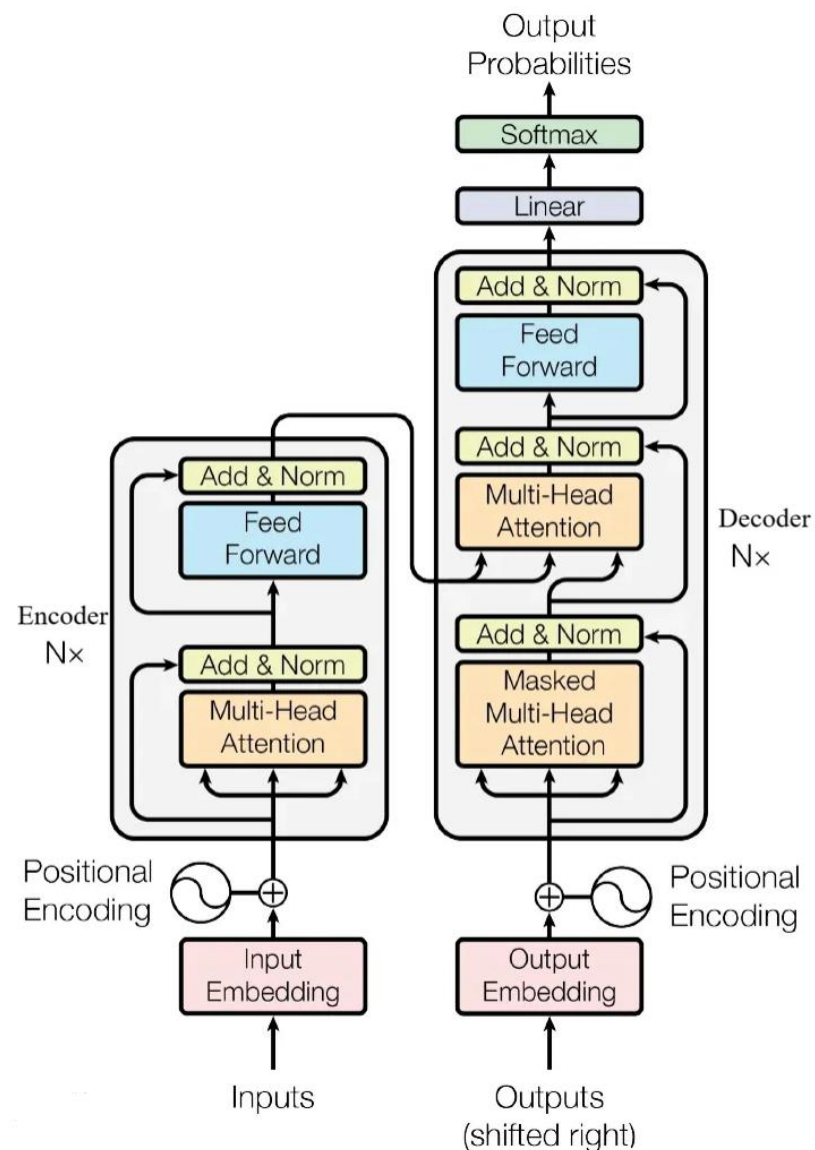
- 1、把目标序列（也称为“输出”）转换为一个输出词嵌入向量（Output Embedding）
- 2、为词向量添加位置编码（Positional Encoding）
- 3、词向量和位置编码结合起来进入解码器的第一层，这一层先进行多头自注意力计算（Multi-Head Attention）
- 4、多头自注意力的输出与原始输出相加（残差连接 Residual Connection），然后经过归一化（Layer Normalization）处理。

Shifted right: 向右移位。因为NLP是根据输入序列预测下一词，所以输出序列需要向右移位



● 解码器:

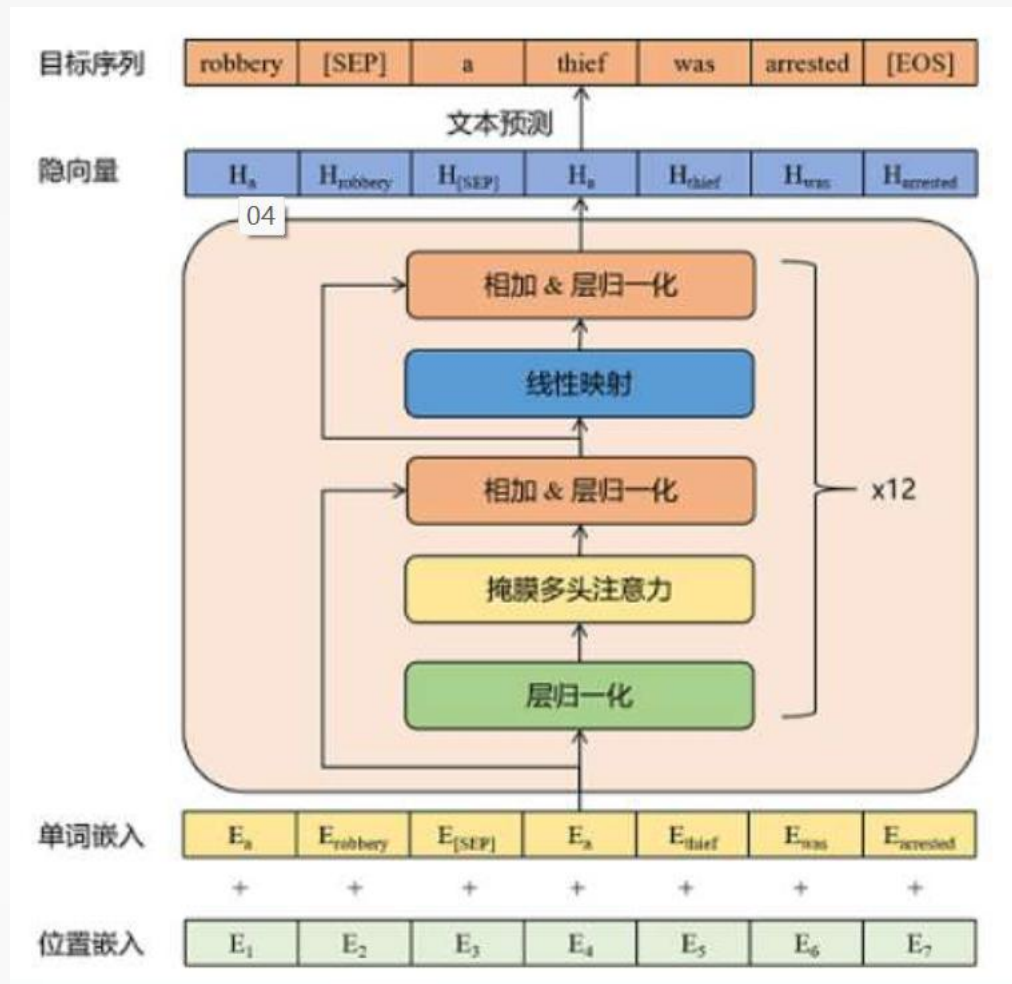
- 5、进行编码器-解码器注意力计算，编码器的输出作为K和V，解码器的自注意力输出作为Q。计算结果进行残差连接和归一化处理。
- 6、归一化的结果，经过一个前馈神经网络（Feed Forward）。
- 7、前馈神经网络的输出与编码器—解码器注意力机制的结果再次相加，并进行残差和归一化的处理。
- 3-7的过程会执行N次
- 8、完成上述所有步骤后，经历一个线性层（Linear）将解码器的输出映射到词表大小的空间，再应用softmax函数得到概率分布。为下游任务提供依据。





- **为什么注意力机制能够大幅提升语言模型性能？**
 - 注意力机制让Transformer能够在不同层次和不同位置捕捉输入序列中的依赖关系
 - 注意力机制使得模型具有强大的表达能力，能够有效处理各种序列到序列任务
 - 由于注意力机制的计算可以高度并行化，Transformer的训练速度也得到了显著提升

- 输入层：词嵌入层+位置嵌入层
- 中间层：Transformer的解码器
Decoder模块*12
- 输出层：LayerNorm归一化+线性全连接层
- 主要参数：详见config/config.json





03/ 项目开发部署



- **数据获取**：收集或网络爬虫获取医患问诊语料库
- **数据处理**：格式转换及向量表示
- **模型构建**：基于GPT模型搭建及参数设置
- **模型训练**：使用云服务器进行模型训练
- **模型评估**：指标指定或人工评估
- **模型上线**：基于界面（Web页面或GUI）的人机交互场景

- 通过多种渠道收集训练数据
- 数据样例：问答方式

数据示例：

胰岛素强化治疗的推荐药有些什么？

预混胰岛素类似物

发育性颈椎管狭窄症的辅助治疗有些什么？

钛板固定

唇畸形的辅助治疗有些什么？

重建口轮匝肌环

严重皮肤软组织损伤的手术治疗有些什么？

确定性手术；简略手术；负压封闭引流(VSD)术



- **目的：将中文文本数据处理成模型能够识别的张量（数字）形式。**
- **处理流程：**
 - 1、读取训练数据，转换成张量，并进行对象的序列化
 - 2、自定义数据集类，用于获取单个样本
 - 3、为训练数据划分批次，以便让模型可以分批次进行训练



- 编写一个函数，将文本进行分词之后，通过词汇表将词转换为对应的id，再转换成向量vector

```
tokenizer = BertTokenizerFast('../vocab/vocab.txt',  
                             sep_token='[SEP]',  
                             pad_token='[PAD]',  
                             cls_token='[CLS]')  
  
sep_id = tokenizer.sep_token_id  
cls_id = tokenizer.cls_token_id  
  
sentences = ['你叫什么名字?', '我叫小明']  
input_ids = [cls_id]  
for s in sentences:  
    input_ids += tokenizer.encode(s, add_special_tokens=False)  
    input_ids += [sep_id]
```

```
['你叫什么名字?', '我叫小明']
```

```
[101, 872, 1373, 784, 720, 1399, 2099, 8043, 102, 2769, 1373, 2207, 3209, 102]
```



- 自定义数据集，将上一步经过处理的数据，存储在torch指定的结构中
- 此类必须继承自torch提供的DataSet类，并重写如下方法：
 - `__init__()`
 - `__len__()`
 - `__getitem__()`

```
class MyDataset(Dataset): 1 usage
    def __init__(self, input_list, max_len):
        super().__init__()
        self.input_list = input_list
        self.max_len = max_len # 每个样本的最大长度

    def __len__(self):
        return len(self.input_list) # 返回样本的个数

    def __getitem__(self, index):
        input_ids = self.input_list[index]
        input_ids = input_ids[:self.max_len]
        input_ids = torch.tensor(input_ids, dtype=torch.long)
        return input_ids
```




- 为训练数据划分批次 (batch_size) , 以便让模型可以分批次进行训练

- 步骤:

- 1、将训练集和验证集的数据加载进来, 存放不同的DataSet中

```
def load_dataset(train_path, valid_path):
```

- 2、对训练集和验证集的数据进行batch划分

```
def get_data_loader(train_path, valid_path):
```

- 3、划分过程中, 对样本进行填充补齐

```
def collate_fn(batch):
```

- 本项目没有预训练模型，通过配置文件直接加载模型

```
def main():  
    if params.pretrained_model:  
        # 加载预训练模型，一般是以.bin为后缀名的文件，本项目没有预训练模型  
        model = GPT2LMHeadModel.from_pretrained(params.pretrained_model)  
    else:  
        model_config = GPT2Config.from_json_file(params.model_config)  
        model = GPT2LMHeadModel(config=model_config)  
    # 将模型加载到设备中CPU or GPU  
    model = model.to(params.device)  
    print(model)
```

- 获取数据
- 遍历每个轮次epoch的数据
- 遍历每个批次batch的数据
- 前向传播
- 计算损失
- 反向传播
- 更新梯度
- 在云服务器上使用GPU进行训练

```
# 每个轮次的训练  
def train_epoch(model, train_dataloader, optimizer, scheduler, epoch):
```

- 使用验证集数据进行模型评估
- 保存损失最小的模型

```
# 模型评估
def evaluate(model, valid_dataloader, epoch):
    # 将模型设置为评估模式，所有参数都不可以进行处理和更新
    model.eval()
    # 记录每个epoch的验证损失
    valid_loss = 0.0
    start_time = datetime.now()
    with torch.no_grad():
        for batch_idx, (input_ids, labels) in enumerate(valid_dataloader):
            # 将输入数据和标签加载到设备中
            input_ids = input_ids.to(params.device)
            labels = labels.to(params.device)
            # 前向传播
            outputs = model(input_ids, labels=labels)
            # 计算损失
            loss = outputs.loss
            # 记录损失
            valid_loss += loss.item()

    # 计算平均损失
    avg_valid_loss = valid_loss / len(valid_dataloader)
    print(f'Epoch [{epoch + 1}/{params.epochs}], Valid Loss: {avg_valid_loss:.4f}')
    end_time = datetime.now()
    print(f'Epoch [{epoch + 1}/{params.epochs}], Valid Time: {end_time - start_time}')
    return avg_valid_loss
```

● 模型应用

- 加载模型和词表
- 将输入数据转换为向量
- 根据输入数据生成答句
- 可以加入历史记录的保存功能

```
# 对输入的对话进行分词
text_ids = tokenizer.encode(text, add_special_tokens=False)
input_ids = [tokenizer.cls_token_id] + text_ids + [tokenizer.sep_token_id]
```

```
for _ in range(params.max_length):
    # 生成回答
    outputs = model(input_ids=input_ids)
    # 获取模型的预测结果
    next_token_logits = outputs.logits[0, -1, :] # 取出最后一个token的预测结果
    # 对预测结果进行softmax, 得到概率分布
    next_token_probs = torch.softmax(next_token_logits, dim=-1)
    # 从概率分布中采样一个token
    next_token_id = torch.multinomial(next_token_probs, num_samples=1)
    # 如果采样的token是[SEP], 则结束生成
    if next_token_id.item() == tokenizer.sep_token_id:
        break
    # 将采样的token添加到回答中
    response.append(next_token_id.item())
    # 将采样的token添加到输入中
    input_ids = torch.cat([input_ids, next_token_id.unsqueeze(0)], dim=-1)
```

- 可以结合GUI界面或者Web页面进行模型的应用

智能健康问诊

流涕不止怎么办

多考虑是属于鼻炎的情况

儿童呼吸道感染的临床表现有什么？

咽部细菌分布；打喷嚏；咳嗽；顽固性咳嗽

鼓膜外伤的症状是什么？

耳溢液；耳漏；耳痛；耳聋；听力减退

发送