

# HW1: 理论部分

2024年9月11日 10:11

备注: S24版GitHub上有某CMU学生分享了自己的全套理论+编程作业, 以下内容的整理结合了我自己的理解查阅、GPT4的解答、以及CMU学生的答案。

## [0 作业概述](#)

## [1 RNN语言模型](#)

### [1.1 问题1: Elman \(即RNN\) 网络模型条件输出问题](#)

[讲义中问题的原文](#)

[Elman 网络介绍](#)

[问题的通俗解释](#)

[解答](#)

[解题思路](#)

[参数选择](#)

[验证](#)

### [1.2 问题2: 双向RNN与自回归语言模型的可实现性探讨](#)

[讲义中问题的原文](#)

[解答](#)

[问题背景:](#)

[问题1.2\(a\) 的解释:](#)

[问题1.2\(b\) 的解释:](#)

[1.2\(b\)概率分布的定义:](#)

[1.2\(a\)和1.2\(b\)的总结:](#)

## [2 transformer语言模型](#)

### [2.1\(a\) 问题1: 乘法注意力 \(multiplicative attention\)](#)

[解答](#)

[理由:](#)

[关于函数类的解释:](#)

我（GPT4）不赞同CMU学生答案的理由：

总结：

2.1(b) 问题2：连接注意力（concatenated attention）

解答

理由：

总结：

2.1(c) 问题3：加性注意力（Additive Attention）

解答

理由：

总结：

问题2.2：自注意力与多头注意力的对称性与多头/单头注意力的等价性问题

解答

3 Sliding Window Attention（滑动窗口注意力）

解答

3.1(a) 公式版的时间复杂度

3.1(b) 公式版的空间复杂度

3.1(c, d, e) 代码版实现、空间复杂度、时间复杂度

代码流程总结：

时间复杂度分析：

空间复杂度分析：

## 0 作业概述

这次作业主要围绕生成式文本模型，具体包括以下几个部分：

### 1. RNN语言模型 (7分)：

- 构建递归神经网络（RNN）用于语言建模，解答与RNN递归方程相关的数值问题，讨论双向RNN能否用于自回归语言模型。

### • Transformer语言模型 (15分)：

- 比较不同类型的注意力机制：缩放点积注意力、乘法注意力、连接注意力和加法注意力，并分析它们是否能够学习不同类别的函数。

- **滑动窗口注意力 (8分):**
  - 探讨滑动窗口注意力的时间和空间复杂度，并编写伪代码以优化滑动窗口注意力的计算效率。
- **编程题：RoPE与GQA (24分):**
  - 实现旋转位置嵌入（RoPE）和分组查询注意力（GQA），并对比它们在训练时间、内存消耗和模型性能上的表现。

# 1 RNN语言模型

## 1.1 问题1：Elman（即RNN）网络模型条件输出问题

### 讲义中问题的原文

1.1 (3 分) 数值问题：考虑一个 RNN（Elman 网络），其输入为  $x_t \in \{0,1\}^2$ ，隐藏向量为  $h_t \in \mathbb{R}^2$ ，输出单元为  $y_t \in \mathbb{R}$ ，其中  $t \in \{1, \dots, T\}$ 。假设递归公式如下所示：

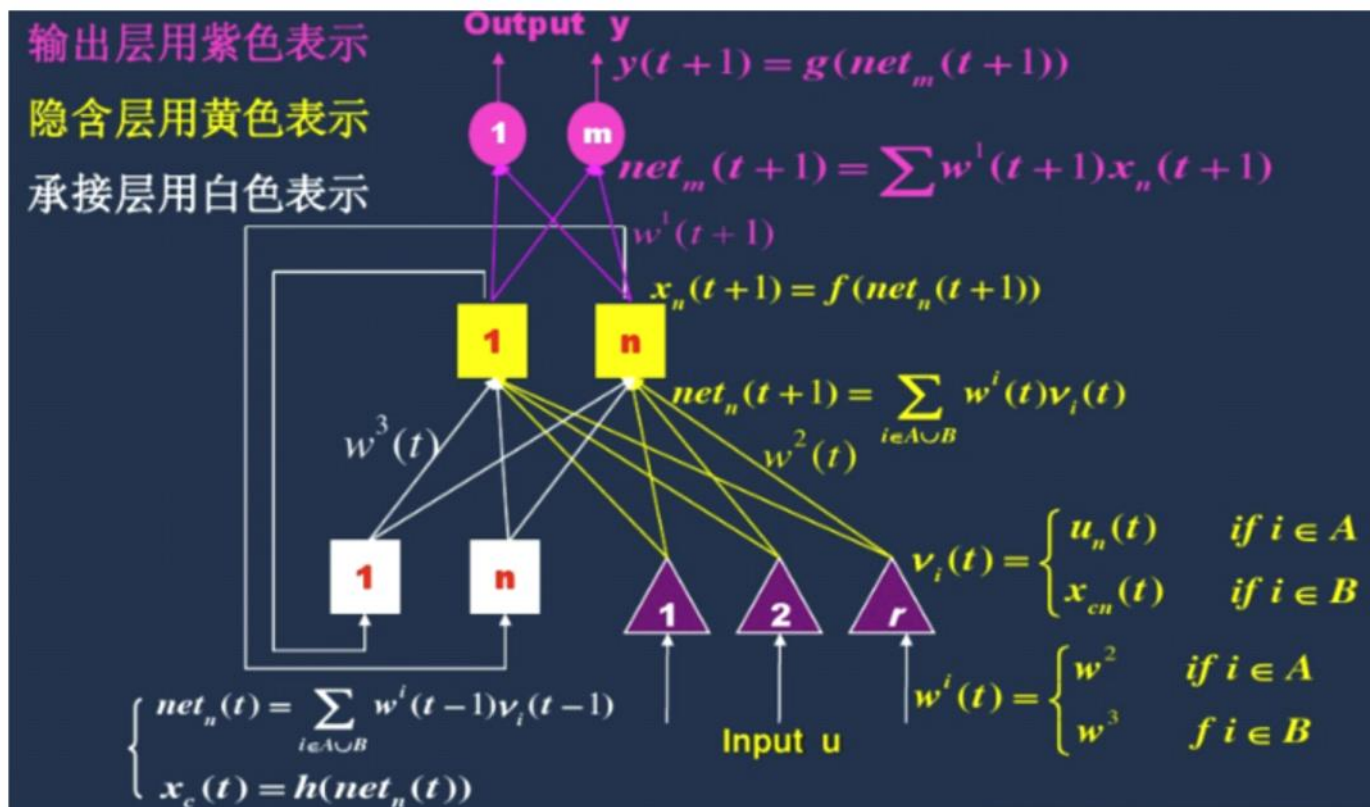
$$h_t = \text{slide}(W_{hh}h_{t-1} + W_{hx}x_t + b_h)$$

$$y_t = \text{slide}(W_{yh}h_t + b_y)$$

其中，激活函数  $\text{slide}(a)$  定义为： $\text{slide}(a) = \min(1, \max(0, a))$ 。请定义参数  $W_{hh} \in \mathbb{R}^{2 \times 2}, W_{hx} \in \mathbb{R}^{2 \times 2}, W_{yh} \in \mathbb{R}^{1 \times 2}, b_h \in \mathbb{R}^2, b_y \in \mathbb{R}$  使得满足以下条件：如果存在  $r, s \leq t$ ，使得  $x_{r,0} = 1$  且  $x_{s,1} = 1$ ，那么  $y_t = 1$ ，否则  $y_t = 0$ 。假设  $h_0 = [0,0]^T$ 。

### Elman 网络介绍

突然冒出个elman网络，查阅了些资料，其实就是我们熟知的最基本的RNN，在此介绍一下Elman网络早年的架构图。只看下图三层网络结构，不要看公式：



Elman网络是一种简单的递归神经网络（RNN），由Jeffrey Elman在1990年提出。它是RNN早期的基本形式之一，具有一个“隐含层”来存储前一时刻的状态信息，从而使网络能够处理序列数据（如时间序列、文本等），并捕捉输入数据中的时序依赖性。

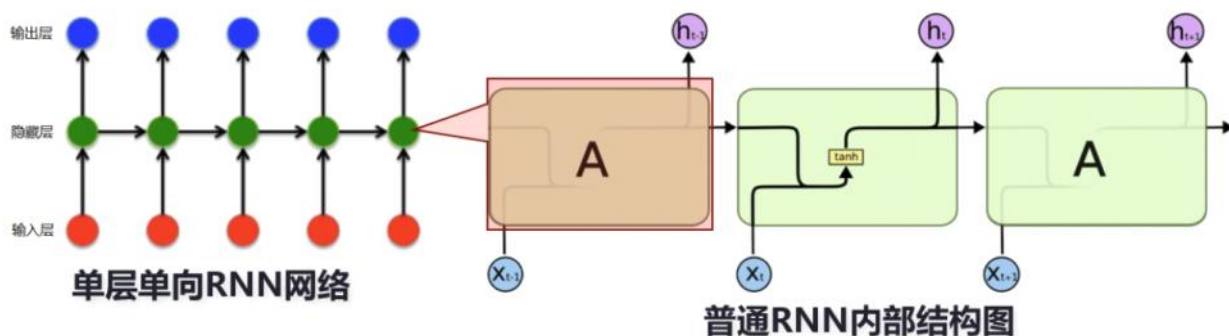
输入（紫色）：代表每个时间步的输入信息。

隐藏层（黄色）：相当于一个全连接

输出（粉色）：最终输出的结果

承接层（白色）：这个就相当于常规RNN中的 $h_t$ ，即缓存的是上一个时间步（即上一次的输入）的隐藏层的输出结果。

贴下当前RNN架构图的绘制方法：



## 问题的通俗解释

这个问题的核心是在构建一个简单的递归神经网络（RNN），该网络需要根据输入序列中的特定模式来输出0或1。具体来说，问题要求我们设计网络的参数，使得当输入序列的某些位置满足特定条件时，网络的输出为1，否则为0。

通俗来说：

### 1. 输入是什么？

- 每个时刻  $t$ ，你会输入一个二维向量  $x_t \in \{0,1\}^2$ ，也就是说，这个向量有两个元素，每个元素都是0或1。

### • 网络要做什么？

- 网络要通过不断处理这些输入，并根据这些输入计算出输出  $y_t$ 。输出的规则是：如果在某些时刻：
  - 输入的第二个元素  $x_{t,1}$  曾经为1，并且
  - 输入的第二个元素  $x_{t,1}$  也曾经为1，那么输出  $y_t$  应该是1，否则输出应该是0。

### • 如何实现？

- 网络有一个“记忆”（隐藏状态  $h_t$ ），它会记住过去发生的事情。我们需要让网络“记住”是否见过第一个元素为1和第二个元素为1的情况。
- 比如，如果在之前的某个时刻  $t$ ，你输入的  $x_{t,0} = 1$ ，那么网络的“记忆”应该更新，记住这一点。同样，如果在某个时刻  $x_{t,1} = 1$ ，网络也要记住。
- 一旦网络“记住”了这两个条件（即第一个元素和第二个元素都见过1），它就输出1；如果这两个条件都没发生或只发生了一个，它就输出0。

## 解答

CMU学生的解答：

Let:

$$W_{hh} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad W_{hx} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad b_h = 0 \quad W_{hy} = \begin{bmatrix} 1 & 1 \end{bmatrix} \quad b_y = -1$$

This simplifies the equations to  $h_t = \text{slide}(h_{t-1} + x_t)$  and  $y_t = \text{slide}([1 \ 1] h_t + b_y)$ .

Given  $h_0 = [0 \ 0]$  and  $h_t = \text{slide}(h_{t-1} + x_t)$ ,  $h_t$  will contain a 1 in position  $i$  if and only if for some  $x_r$  where  $r \leq t$ ,  $x_{r,i} = 1$  for  $i \in \{0, 1\}$ . Additionally once  $h_{t,i}$  becomes 1, all  $h_{v,i}$  for  $v \geq t$  will also equal 1.

Given  $W_{hy} = [1 \ 1]$  and  $b_y = -1$ ,  $y_t = \text{slide}(W_{hy}h_t + b_y)$  can be rewritten as  $y_t = \text{slide}(\sum_{i=0}^1 (h_{t,i}) - 1)$ . Using the matrices given above,  $h_{t,i} \in \{0, 1\}$  where  $i \in \{0, 1\}$ . When  $h_t = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ,  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ , or  $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$   $y_t$  evaluates to 0. If  $h_t = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$  then  $y_t = 1$  satisfying the constraints given.

以下是GPT的解答：

问题给出的RNN的结构为：

1.  $h_t = \text{slide}(W_{hh}h_{t-1} + W_{hx}x_t + b_h)$

$$2. y_t = \text{slide}(W_{yh}h_t + b_y)$$

其中, `slide` 函数的作用是将值限制在  $[0, 1]$  之间, 即:

$$\text{slide}(a) = \min(1, \max(0, a))$$

需要我们定义参数  $W_{hh}, W_{hx}, W_{yh}, b_h, b_y$ , 使得:

- $y_t = 1$ , 当存在  $r, s \leq t$  使得  $x_{r,0} = 1$  并且  $x_{s,1} = 1$
- 否则  $y_t = 0$

假设初始状态  $h_0 = [0, 0]^T$ 。

## 解题思路

### 1. 目标条件分析:

- 我们希望当存在一个时刻  $r$  使得  $x_{r,0} = 1$  (即输入的第二个元素为1) 并且存在另一个时刻  $s$  使得  $x_{s,1} = 1$  (即输入的第二个元素为1) 时, 输出  $y_t = 1$ 。
- 因此我们可以利用隐藏状态  $h_t$  来存储这些信息, 即记录是否见过  $x_{t,0} = 1$  和  $x_{t,1} = 1$ 。
- **隐藏状态  $h_t$  的设计:**
  - 将隐藏状态设计为  $h_t = [h_t^{(0)}, h_t^{(1)}]$ , 其中:
    - $h_t^{(0)}$  用来记录是否见过  $x_{t,0} = 1$ 。
    - $h_t^{(1)}$  用来记录是否见过  $x_{t,1} = 1$ 。
  - 当  $h_t^{(0)} = 1$  时, 说明在当前时刻之前至少有一个  $x_{r,0} = 1$ ;
  - 当  $h_t^{(1)} = 1$  时, 说明在当前时刻之前至少有一个  $x_{s,1} = 1$ 。
- **递归更新  $h_t$ :**
  - 对于  $W_{hx}$ , 当  $x_{t,0} = 1$  时, 需要更新  $h_t^{(0)} = 1$ ;
  - 对于  $W_{hx}$ , 当  $x_{t,1} = 1$  时, 需要更新  $h_t^{(1)} = 1$ ;
  - 这些信息将通过递归公式传递, 因此设计  $W_{hh}$  来使之前的状态能够保持。
- **输出  $y_t$  的设计:**
  - 当  $h_t^{(0)} = 1$  且  $h_t^{(1)} = 1$  时, 输出  $y_t = 1$ , 否则  $y_t = 0$ 。

## 参数选择

我们可以选择以下参数来实现这个目标:

$$1. W_{hx} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

这样当  $x_{t,0} = 1$  时, 隐藏状态  $h_t^{(0)}$  更新为1; 当  $x_{t,1} = 1$  时, 隐藏状态  $h_t^{(1)}$  更新为1。

$$2. W_{hh} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

这可以确保上一时刻的隐藏状态会传递到当前时刻。

$$3. b_h = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

无需额外的偏置项。

4.  $W_{yh} = \begin{pmatrix} 1 & 1 \end{pmatrix}$

输出  $y_t$  取决于隐藏状态的两个分量的和，当且仅当两个分量都是1时，才能满足  $y_t = 1$ 。

5.  $b_y = -1$

这样只有当两个分量的和为2时，才能使得输出  $y_t$  通过 `slide` 函数达到1，否则输出为0。

## 验证

我们需要验证：当且仅当  $x_{r,0} = 1$  和  $x_{s,1} = 1$  存在时，输出  $y_t = 1$ ，否则  $y_t = 0$ 。

我们从初始状态  $h_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$  开始，逐步推导隐藏状态  $h_t$  和输出  $y_t$  的计算过程。

**步骤 1:** 当  $x_t = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  时

- 输入  $x_t$  的第一个元素为 1，第二个元素为 0。
- 递归更新  $h_t$ :

$$\begin{aligned} h_t &= \text{slide}(W_{hh}h_{t-1} + W_{hx}x_t + b_h) \\ h_t &= \text{slide}\left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} h_{t-1}^{(0)} \\ h_{t-1}^{(1)} \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) \\ h_t &= \text{slide}\left(\begin{pmatrix} h_{t-1}^{(0)} \\ h_{t-1}^{(1)} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) = \text{slide}\left(\begin{pmatrix} h_{t-1}^{(0)} + 1 \\ h_{t-1}^{(1)} \end{pmatrix}\right) \\ h_t &= \begin{pmatrix} 1 \\ h_{t-1}^{(1)} \end{pmatrix} \end{aligned}$$

因此，此时  $h_t^{(0)} = 1$ ，表示见到了  $x_{t,0} = 1$ 。

**步骤 2:** 当  $x_t = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  时

- 输入  $x_t$  的第一个元素为 0，第二个元素为 1。
- 递归更新  $h_t$ :

$$\begin{aligned} h_t &= \text{slide}(W_{hh}h_{t-1} + W_{hx}x_t + b_h) \\ h_t &= \text{slide}\left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} h_{t-1}^{(0)} \\ h_{t-1}^{(1)} \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) \\ h_t &= \text{slide}\left(\begin{pmatrix} h_{t-1}^{(0)} \\ h_{t-1}^{(1)} + 1 \end{pmatrix}\right) \\ h_t &= \begin{pmatrix} h_{t-1}^{(0)} \\ 1 \end{pmatrix} \end{aligned}$$

此时  $h_t^{(1)} = 1$ ，表示见到了  $x_{t,1} = 1$ 。

**步骤 3:** 输出  $y_t$  的计算

- 输出层的公式为：

$$y_t = \text{slide}(W_{yh}h_t + b_y)$$

其中  $W_{yh} = \begin{pmatrix} 1 & 1 \end{pmatrix}$  和  $b_y = -1$ 。

所以当  $h_t = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$  时：

$$y_t = \text{slide}(1 \times 1 + 1 \times 1 - 1) = \text{slide}(1) = 1$$

即输出为 1。

当  $h_t = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  或  $h_t = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  时：

$$y_t = \text{slide}(1 \times 1 + 1 \times 0 - 1) = \text{slide}(0) = 0$$

或

$$y_t = \text{slide}(1 \times 0 + 1 \times 1 - 1) = \text{slide}(0) = 0$$

即输出为 0。

### 验证总结：

通过上面的推导，我们可以验证以下结果：

- 当  $h_t^{(0)} = 1$  且  $h_t^{(1)} = 1$ ，即同时见到了  $x_{t,0} = 1$  和  $x_{t,1} = 1$  时，输出  $y_t = 1$ 。
- 如果仅有一个分量为 1，或者两个分量都为 0，输出  $y_t = 0$ 。

因此，所选的参数符合题目要求，成功实现了题目中要求的条件。

## 1.2 问题2：双向RNN与自回归语言模型的可实现性探讨

### 讲义中问题的原文

自回归语言模型定义了一个序列  $x_{1:T}$  上的概率分布形式如下：

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$$

(a) (2 分) 简答：假设我们给定一个输入  $x_{1:T}$ ，并且定义一个双向 RNN，如下所示：

$$f_t = \sigma(W_f f_{t-1} + W_{fx} x_t + b_f), \forall t \in \{1, \dots, T\}$$

$$g_t = \sigma(W_g g_{t+1} + W_{gx} x_t + b_g), \forall t \in \{1, \dots, T\}$$

$$h_t = \sigma(W_h f_t + W_h g_t + b_h), \forall t \in \{1, \dots, T\}$$

注意  $f_t$  从左侧建立上下文， $g_t$  从右侧建立上下文， $h_t$  结合了两者的。我们是否可以定义形式为  $p(x_{1:T}) = \prod_{t=1}^T p(x_t | h_{t-1})$  的自回归语言模型？如果可以，定义这个概率分布；如果不可以，请解释原因。

(b) (2 分) 简答：假设  $\text{BiRNN}(x_{1:t-1})$  计算子序列  $x_{1:t-1}$  上的双向 RNN，并返回  $h_{t-1}$ 。我们是否可以定义形式为  $p(x_{1:T}) = \prod_{t=1}^T p(x_t | \text{BiRNN}(x_{1:t-1}))$  的自回归语言模型？如果可以，定义这个概率分布；如果不可以，请解释原因。



## 解答

CMU学生的解答：

1.2 (a) :

No because an autoregressive language model defines the probability of the current input as being dependent on all previous. However,  $h_t$  depends on all inputs future and past. Hence,  $\prod_{t=1}^T p(x_t | h_{t-1}) \neq \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$ .

1.2 (b) :

Yes because  $\text{BiRNN}(\mathbf{x}_{1:t-1})$  contains info on all the  $\mathbf{x}$  from time 1 to time  $t-1$ . Thus it is possible to represent  $\prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$  as  $\prod_{t=1}^T p(x_t | \text{BiRNN}(\mathbf{x}_{1:t-1}))$ .  
The probability distribution model is:  $p(x_t | \text{BiRNN}(\mathbf{x}_{1:t-1})) = p(x_t | h_{t-1})$  where  $h_i$  is dependent on  $x_0, \dots, x_{t-1} \forall i \in 1 \dots (t-1)$ .

以下是GPT解答：

**1.2问题**涉及到自回归语言模型以及双向RNN的结构设计。问题的核心是探讨如何在双向RNN的基础上定义自回归语言模型，或者说能否通过某些改动来实现这种模型。下面我将对问题进行通俗解释。

### 问题背景：

首先，什么是**自回归语言模型**？它是通过对前面已经看到的序列来预测下一个单词或字符。具体来说，自回归模型的核心是使用已经看到的序列来计算当前时刻的条件概率：

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$$

这意味着：我们在生成序列时，每个时间步  $t$  的输出  $x_t$  都依赖于之前的所有输入  $x_1, \dots, x_{t-1}$ 。

### 问题1.2(a) 的解释：

- **双向RNN (BiRNN)** 是一种扩展的RNN，它同时考虑序列的**前后文信息**，即它的两个隐藏状态：
  - $f_t$ ：从左到右处理，构建从左侧传来的上下文。
  - $g_t$ ：从右到左处理，构建从右侧传来的上下文。
- 现在问题问：**能否使用这种双向RNN来定义一个自回归语言模型？**

通俗地说，自回归语言模型要求在生成  $x_t$  时，不能利用未来的信息（即  $t+1, t+2, \dots$ ）。然而，双向RNN的

结构同时利用了前向和后向的隐藏状态，即它利用了未来的上下文信息。因此，这与自回归语言模型的要求不符。

**结论：**不可以直接使用双向RNN定义自回归语言模型，因为双向RNN的设计依赖未来的信息，而自回归模型只依赖过去的信息。

### 问题1.2(b) 的解释：

- 在这个问题中，假设我们使用一个双向RNN，但只让它处理到时间步  $t - 1$  的输入（即不考虑未来的输入）。
- 现在问题问：**如果我们只使用时间步  $t - 1$  的双向RNN隐藏状态来生成  $x_t$** ，是否可以定义自回归语言模型？

这个问题实际上是在问：**是否可以利用双向RNN的隐藏状态（只处理到  $t - 1$ ）来进行自回归预测？**

通俗地说，如果我们在生成当前的  $x_t$  时，只使用之前时间步  $x_1, \dots, x_{t-1}$  的信息，那么这是符合自回归模型要求的。因为它只使用了过去的信息，符合自回归模型不使用未来信息的条件。

**结论：**可以使用双向RNN处理到时间步  $t - 1$  的隐藏状态来定义自回归语言模型，因为此时没有使用未来的输入序列。

### 1.2(b) 概率分布的定义：

- 自回归语言模型的基本公式为：

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t | x_1, \dots, x_{t-1})$$

这意味着每个时刻  $x_t$  的输出只依赖于之前时刻的输入。

- 在这个问题中，双向RNN的隐藏状态  $h_{t-1}$ （包括  $f_{t-1}$  和  $g_{t-1}$ ）能够捕捉之前的输入序列信息。因此，我们可以用隐藏状态  $h_{t-1}$  来代替  $x_1, \dots, x_{t-1}$  作为当前时刻  $x_t$  的条件：

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t | h_{t-1}) = \prod_{t=1}^T p(x_t | f_{t-1}, g_{t-1})$$

其中， $f_{t-1}$  是从左到右的隐藏状态， $g_{t-1}$  是从右到左的隐藏状态。

### 最终结论：

我们可以使用双向RNN处理到时间步  $t - 1$  的隐藏状态  $h_{t-1}$  来定义自回归语言模型，其概率分布为：

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t | h_{t-1}) = \prod_{t=1}^T p(x_t | f_{t-1}, g_{t-1})$$

该公式表明，序列的生成是基于前一时刻的隐藏状态，符合自回归语言模型的要求。

### 1.2(a)和1.2(b)的总结：

- 1.2(a)：**不能直接使用双向RNN定义自回归语言模型，因为它依赖未来的信息。
- 1.2(b)：**如果我们只使用双向RNN处理到时间步  $t - 1$  的信息来生成  $x_t$ ，则可以定义自回归语言模型，因为它只依赖过去的信息。

## 2 transformer语言模型

Transformer 使用缩放点积注意力 (scaled-dot-product attention) :

$$s_{t,j} = \frac{k_j^T q_t}{\sqrt{|k|}}, \forall j, t$$

$$a_t = \text{softmax}(s_t), \forall t$$

其中, 值 (values)、查询 (queries) 和键 (keys) 分别由以下公式给出:  $v_j = W_v^T x_j$ ,  $q_j = W_q^T x_j$ , 以及  $k_j = W_k^T x_j$  对于所有  $j$ 。这里,  $v_j, q_j, k_j \in \mathbb{R}^{d_k}$ 。

### 2.1(a) 问题1: 乘法注意力 (multiplicative attention)

乘法注意力将注意力权重定义为:

$$\tilde{s}_{t,j} = \frac{k_j^T W_s q_t}{\sqrt{|k|}}, \forall j, t$$

其中,  $W_s \in \mathbb{R}^{d_k \times d_k}$  是一个参数矩阵。一个使用乘法注意力的 Transformer 能否学习到与使用更简单的缩放点积注意力不同的函数类? 简要说明你的理由。

### 解答

CMU学生的解答: (GPT4并不认同该答案)

No because the multiplicative attention model can be thought of as a transformed version of the scaled dot-product attention model. In particular, if  $W_s q_t$  is thought of as a linear transformation of  $q_t$ . Then the equation for  $\tilde{s}_{t,j}$  becomes almost identical to that of  $s_{t,j}$ . Instead of using the scalar dot product between  $k_j^T$  and  $q_t$ , the multiplicative model uses the scalar dot product between  $k_j^T$  and the transformation of  $q_t$  by the matrix  $W_s$ .

GPT的解答:

**结论: 是的, 使用乘法注意力的 Transformer 能够学习到与缩放点积注意力不同的函数类。**

**理由:**

1. **乘法注意力的额外自由度:**

- 在 **乘法注意力** 中, 查询向量  $q_t$  经过了一个可训练的线性变换  $W_s$ , 这使得相似性度量具有更高的灵活性。这个矩阵  $W_s$  是额外的参数, 能够对查询进行方向和尺度上的调整。
- 相较之下, **缩放点积注意力** 中的查询  $q_t$  和键  $k_j$  直接进行点积运算, 并通过  $\frac{1}{\sqrt{|k|}}$  进行缩放, 但没有进一步的变换。这种相似性度量是固定的, 仅依赖于查询和键向量的方向性。

- **扩展的函数类：**

- **缩放点积注意力** 的函数类是基于直接的点积计算，即：

$$s_{t,j} = \frac{k_j^T q_t}{\sqrt{|k|}}$$

这是一个简单的相似性度量函数，计算的是查询和键之间的夹角或相对方向。

- **乘法注意力** 的函数类更广泛，因为查询  $q_t$  在与键  $k_j$  点积之前，先经过了线性变换  $W_s$ 。这意味着它不仅可以表示点积相似性，还可以通过学习  $W_s$  来调整查询向量，从而学习到更复杂的相似性关系。

$$\tilde{s}_{t,j} = \frac{k_j^T W_s q_t}{\sqrt{|k|}}$$

因此，乘法注意力能够学习到比缩放点积注意力更加灵活和复杂的相似性函数，能够捕捉到不同的模式。

## 关于函数类的解释：

- **函数类 (function class)** 是指一组具有相似结构或行为的函数的集合。它描述了从输入到输出的映射关系。在机器学习中，函数类是由模型的结构、参数以及计算方法所决定的。
  - **在注意力机制中**，函数类表示的是用于计算查询  $q_t$  和键  $k_j$  之间相似性得分的函数集合。例如，缩放点积注意力的函数类包括直接通过点积计算相似性的函数，而乘法注意力的函数类由于引入了线性变换，涵盖了更多的相似性度量方式。

## 我 (GPT4) 不赞同CMU学生答案的理由：

某人在截图中的观点认为：“**乘法注意力无法学习到与缩放点积注意力不同的函数类**”，因为它仅仅是对缩放点积注意力的线性变换版本。这个观点过于局限，忽视了**线性变换**  $W_s$  的可训练性和灵活性。

- **为什么不认同？**
  - **可训练的  $W_s$** ：乘法注意力通过引入矩阵  $W_s$  对查询  $q_t$  进行线性变换，使得相似性度量不再局限于简单的点积，而是能够学习和表达更多复杂的关系。这就意味着，乘法注意力可以表示出缩放点积注意力无法表达的相似性度量。
  - **不同的函数类**：虽然在表面形式上，乘法注意力的点积结构与缩放点积注意力相似，但由于矩阵  $W_s$  的存在，乘法注意力的函数类要比缩放点积注意力的函数类更广。因此，**乘法注意力能够学习到与缩放点积注意力不同的函数类**。

## 总结：

- **乘法注意力** 引入了额外的线性变换  $W_s$ ，使得它能够表示比缩放点积注意力更复杂的相似性函数，因此它们学习的函数类是不同的。
- 某人的观点忽略了乘法注意力中的  $W_s$  的可训练性，这也是我不认同其结论的原因。

## 2.1(b) 问题2：连接注意力 (concatenated attention)

连接注意力将注意力权重定义为：

$$\hat{s}_{t,j} = w_s^T [k_j; q_t], \forall j, t$$

其中,  $w_s \in \mathbb{R}^{2d_k}$  是一个参数向量, 且  $[a; b]$  表示向量  $a$  和  $b$  的连接。是否存在参数  $w_s$ , 使得  $\hat{s}_{t,j}$  能近似等于两个向量  $k_j$  和  $q_t$  之间的角度  $\theta$ , 或  $\cos(\theta)$ ? 简要说明理由 (不需要正式证明)。

## 解答

CMU学生：

There does not exist parameters such that  $s_{t,j}$  will approximate the angle  $\theta$ . This is because there is no way for a scalar dot product to approximate the non-linear cosine function.

GPT4解答：

**结论：不存在**这样的固定参数  $w_s$ , 使得  $\hat{s}_{t,j}$  能够在查询  $q_t$  和键  $k_j$  变化的情况下, 始终近似表示它们之间的角度  $\theta$  或余弦相似度  $\cos(\theta)$ 。

**理由：**

1. **连接操作与固定参数的局限性：**

- 在连接注意力中,  $[k_j; q_t]$  是通过将两个向量  $k_j \in \mathbb{R}^{d_k}$  和  $q_t \in \mathbb{R}^{d_k}$  拼接起来形成一个  $2d_k$  维的向量：

$$[k_j; q_t] = \begin{pmatrix} k_j \\ q_t \end{pmatrix} \in \mathbb{R}^{2d_k}$$

然后使用固定的权重向量  $w_s$  对该拼接向量进行线性变换：

$$\hat{s}_{t,j} = w_s^T [k_j; q_t]$$

这个线性变换只能计算出一个**固定的线性组合**, 它无法直接表示两个向量之间的非线性关系, 例如角度  $\theta$  或余弦相似度  $\cos(\theta)$ , 因为它们依赖于向量的相对方向和长度。

- 余弦相似度的非线性特性：**

- 余弦相似度  $\cos(\theta)$  的公式为：

$$\cos(\theta) = \frac{k_j^T q_t}{\|k_j\| \|q_t\|}$$

这是一个非线性函数, 它不仅涉及向量  $k_j$  和  $q_t$  的点积, 还需要计算它们的长度  $\|k_j\|$  和  $\|q_t\|$ 。而连接注意力中的线性变换  $w_s^T [k_j; q_t]$  并没有这种非线性处理能力, 因此无法准确表示或近似余弦相似度。

- 输入的变化与固定参数的关系：**

- 在注意力机制中，查询  $q_t$  和键  $k_j$  是随每次输入变化的，而  $w_s$  是一个**固定的参数向量**。由于  $w_s$  不会随输入变化，因此它无法动态地调整来适应每个不同的  $q_t$  和  $k_j$ ，从而计算出它们之间的角度  $\theta$  或余弦相似度。
- 点积与余弦相似度的不同：**
  - 点积 ( $k_j^T q_t$ ) 与余弦相似度 ( $\cos(\theta)$ ) 之间有一个关键区别：点积是一个线性度量，而余弦相似度是一个非线性度量，涉及向量的方向和大小。固定的线性变换  $w_s^T [k_j; q_t]$  无法表示这种非线性关系。
  - 因此，**即便通过合适的权重组合**， $w_s$  也不能完全捕捉到  $\cos(\theta)$  这样复杂的非线性相似度。

## 总结：

- 不存在**固定的参数  $w_s$ ，可以使得连接注意力的输出  $\hat{s}_{t,j}$  近似等于两个向量  $k_j$  和  $q_t$  之间的角度  $\theta$  或余弦相似度  $\cos(\theta)$ 。
- 连接注意力中的线性变换  $w_s^T [k_j; q_t]$  只能表示线性关系，而无法捕捉两个向量间的非线性相似性，例如角度或余弦相似度。

## 2.1(c) 问题3：加性注意力 (Additive Attention)

加性注意力将注意力权重定义为：

$$\hat{s}_{t,j} = w_s^T \tanh(W_s[k_j; q_t]), \forall j, t$$

其中， $w_s \in \mathbb{R}^{d_s}$  和  $W_s \in \mathbb{R}^{d_s \times 2d_k}$  是参数，且  $d_s$  是超参数。是否存在参数  $w_s, W_s$ ，使得  $\hat{s}_{t,j}$  近似等于向量  $k_j$  和  $q_t$  之间的角度  $\theta$  或  $\cos(\theta)$ ？简要说明理由（不需要正式证明）。

## 解答

CMU学生：

Yes,  $\tanh(W_s[k_j; q_t])$  can be used to approximate  $\cos(\theta)$  since  $w_s^T \tanh(W_s[k_j; q_t])$  can be thought of as a layer in a neural network. Since a hidden layer in a neural network can be thought of as a universal function approximator, it can be trained to approximate  $\cos(\theta)$  where  $\theta$  is the angle between  $k_j$  and  $q_t$ .

GPT解答：

**结论：**存在这样的参数  $w_s$  和  $W_s$ ，可以使加性注意力中的输出  $\hat{s}_{t,j}$  近似表示  $k_j$  和  $q_t$  之间的余弦相似度  $\cos(\theta)$ ，这是因为加性注意力引入了非线性激活函数  $\tanh$  和可学习的线性变换，具备足够的表示能力来近似复杂的非线性关系。

## 理由：

### 1. 非线性激活函数的作用：

- 加性注意力中的公式使用了  $\tanh$  作为非线性激活函数。 $\tanh$  函数能够将线性组合的输入（在这里是  $W_s[k_j; q_t]$ ）转化为非线性输出，具备近似任意非线性函数的能力，包括余弦相似度这样的非线性关系。
- **神经网络的近似能力：**
  - 神经网络的一个基本性质是**通用函数逼近器**，即通过足够的参数和非线性激活函数，神经网络可以近似任意的连续函数。这里的  $w_s^T \tanh(W_s[k_j; q_t])$  可以看作是一个简单的神经网络结构，它能够通过训练来逼近  $\cos(\theta)$ ，即查询  $q_t$  和键  $k_j$  之间的余弦相似度。
- **线性变换与非线性结合的灵活性：**
  - $W_s$  是一个线性变换矩阵，它可以对输入  $[k_j; q_t]$  进行初步的线性映射。结合非线性激活函数  $\tanh$ ，这提供了足够的自由度来表示复杂的相似性度量（如余弦相似度）。
  - 参数  $w_s$  的线性组合进一步将这些非线性输出组合在一起。因此，经过适当的训练，参数  $w_s$  和  $W_s$  完全可以学习到近似  $\cos(\theta)$  的映射。

## 总结：

- **存在**这样的参数  $w_s$  和  $W_s$ ，使得加性注意力的输出  $\hat{s}_{t,j}$  可以近似表示查询  $q_t$  和键  $k_j$  之间的角度或余弦相似度。这是因为加性注意力结合了非线性激活函数  $\tanh$  和线性变换，具备近似复杂非线性关系的能力。

## 问题2.2：自注意力与多头注意力的对称性与多头/单头注意力的等价性问题

自注意力（Self-attention）通常通过矩阵乘法来计算。这里我们考虑没有因果注意力掩码的多头注意力（multi-headed attention）。

设：

$$\begin{aligned} X &= [x_1, \dots, x_N]^T \\ V^{(i)} &= XW_v^{(i)} \\ K^{(i)} &= XW_k^{(i)} \\ Q^{(i)} &= XW_q^{(i)} \\ S^{(i)} &= Q^{(i)}(K^{(i)})^T / \sqrt{d_k} \\ A^{(i)} &= \text{softmax}(S^{(i)}) \\ X'^{(i)} &= A^{(i)}V^{(i)} \\ X' &= \text{concat}(X'^{(1)}, \dots, X'^{(h)}) \end{aligned}$$

其中,  $N$  是序列长度,  $h$  是注意力头的数量, 每一行与  $i$  相关的定义适用于所有  $i \in \{1, \dots, h\}$ 。

(a) (3分) 简答: 注意力矩阵  $A^{(i)}$  是否总是对称的? 如果是, 请证明它是对称的。如果不是, 请描述一个确保它对称的条件。

(b) (4分) 简答: 假设我们有两个注意力头  $h = 2$ , 令  $d_k = d_m/h$ , 并且我们有一个单一输入  $X$ 。令  $X'$  为在参数  $W_v^{(1)}, W_k^{(1)}, W_q^{(1)}, W_v^{(2)}, W_k^{(2)}, W_q^{(2)} \in R^{d_m \times d_k}$  上执行的多头注意力的输出。现在, 假设我们使用这些相同的参数, 并沿行拼接得到新的参数:

$$W'_v = \text{concat}(W_v^{(1)}, W_v^{(2)}), W'_k = \text{concat}(W_k^{(1)}, W_k^{(2)}), W'_q = \text{concat}(W_q^{(1)}, W_q^{(2)}) \in R^{d_m \times d_m}$$

令  $X''$  为在参数  $W'_v, W'_k, W'_q$  上执行的单头注意力的输出。在这种情况下,  $X'' = X'$  吗? 请证明你的答案。

## 解答

CMU学生:

2.2 (a) :

The attention matrix is not always symmetric. One way to ensure  $A^{(i)}$  is always symmetric is to enforce the condition  $W_k^{(i)} = W_q^{(i)}$ . This condition would cause  $K^{(i)}$  to equal  $Q^{(i)}$ . Hence  $Q^{(i)}(K^{(i)})^T$  would be equivalent to  $Q^{(i)}(Q^{(i)})^T$  which is symmetric.

2.2 (b) :

No, because when calculating  $S^{(i)}$  the matrices  $Q^{(i)}K^{(i)}$  is scaled by  $\sqrt{d_k}$ . When there are two heads  $d_k = d_m/2$ . When there is one head  $d_k = d_m$ , thus  $S \neq S'$  where  $S$  is calculated using multi-headed attention and  $S'$  is calculated using single-headed attention. This means that the attention matrix between the multi-headed and single-headed attention models won't be the same which means  $X'$  and  $X''$  are not the same.

GPT解答:

**问题2.2 (a) 的解答:**

注意力矩阵  $A^{(i)}$  并不总是对称的。具体来说,  $A^{(i)}$  的计算方式是通过:

$$A^{(i)} = \text{softmax}(S^{(i)})$$

其中,  $S^{(i)} = Q^{(i)}(K^{(i)})^T / \sqrt{d_k}$ 。

由于  $Q^{(i)}$  和  $K^{(i)}$  是通过不同的权重矩阵  $W_q^{(i)}$  和  $W_k^{(i)}$  从输入  $X$  中生成的, 因此  $Q^{(i)} \neq K^{(i)}$  是常见的情况。因此,  $S^{(i)}$  通常不会是对称的矩阵。通过 softmax 操作后,  $A^{(i)}$  也不会是对称的。

**确保对称的条件:**



要确保注意力矩阵  $A^{(i)}$  是对称的，需要使得  $Q^{(i)} = K^{(i)}$ 。换句话说，只有当查询矩阵  $Q^{(i)}$  和键矩阵  $K^{(i)}$  完全相同时，注意力矩阵  $S^{(i)}$  才会对称，而这要求  $W_q^{(i)} = W_k^{(i)}$ 。在这种情况下， $S^{(i)} = Q^{(i)}(Q^{(i)})^T / \sqrt{d_k}$  是对称的，从而  $A^{(i)}$  也会对称。

总结：

- $A^{(i)}$  通常不是对称的。
- 当  $W_q^{(i)} = W_k^{(i)}$  时， $A^{(i)}$  才会是对称的。

### 问题2.2 (b) 的解答：

问题的核心是在比较两个不同的注意力机制输出是否相等：一种是多头注意力（multi-headed attention），另一种是将注意力头的参数沿行拼接并作为单头注意力（single-headed attention）的输出。

给定的情况是：

- 多头注意力使用的参数为  $W_v^{(1)}, W_k^{(1)}, W_q^{(1)}$  和  $W_v^{(2)}, W_k^{(2)}, W_q^{(2)}$ 。
- 单头注意力则使用拼接后的参数  $W'_v = \text{concat}(W_v^{(1)}, W_v^{(2)})$ ， $W'_k = \text{concat}(W_k^{(1)}, W_k^{(2)})$ ，和  $W'_q = \text{concat}(W_q^{(1)}, W_q^{(2)})$ 。

问题：  $X'' = X'$  吗？

答案：

$X'' \neq X'$ 。

解释：

#### 1. 多头注意力的特性：

在多头注意力机制中，输入  $X$  被分别投影到不同的查询、键和值空间中（每个注意力头的维度为  $d_k$ ），计算出相应的注意力矩阵  $A^{(i)}$  并作用在值向量上。这些不同的注意力头可以捕捉输入序列中的不同模式。最终的输出  $X'$  是将各个头的输出  $X'^{(i)}$  进行拼接后得到的。

#### 2. 单头注意力的特性：

在单头注意力机制中，输入  $X$  是通过一个拼接后的参数矩阵  $W'_v, W'_k, W'_q$  投影到一个维度更大的查询、键和值空间中。这意味着单头注意力只执行一次注意力计算，无法像多头注意力那样对输入序列进行多角度的模式捕捉。

#### 3. 拼接的影响：

虽然多头注意力的参数可以通过拼接形式转换为单头注意力的参数，但由于单头注意力只进行一次注意力计算，而多头注意力在每个头上进行独立的注意力计算并拼接结果，因此单头注意力的输出  $X''$  与多头注意力的输出  $X'$  不同。

换句话说，单头注意力将所有信息合并到一个注意力头中，而多头注意力可以独立地处理多个注意力头，捕捉不同的上下文关系。因此，拼接后的单头注意力无法完全重现多头注意力的效果。

总结：

- $X'' \neq X'$ ，因为多头注意力和单头注意力的计算方式不同，拼接参数并不能保证输出相等。

### 3 Sliding Window Attention (滑动窗口注意力)

滑动窗口注意力的最简单定义方法是将因果掩码  $M$  设置为仅包含  $\frac{1}{2}w + 1$  个标记，其中最右侧的窗口元素为当前标记（即在对角线上）。然后我们的注意力计算为：

$$X' = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

例如，如果序列长度为  $N = 6$ ，窗口大小为  $w = 4$ ，则掩码矩阵  $M$  为：

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty & -\infty & -\infty \\ 0 & 0 & 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & 0 & 0 & -\infty & -\infty \\ -\infty & -\infty & 0 & 0 & 0 & -\infty \\ -\infty & -\infty & -\infty & 0 & 0 & 0 \end{bmatrix}$$

(a) (1 分) 简答：如果我们按照公式中的矩阵乘法实现滑动窗口，时间复杂度在  $N$  和  $w$  的情况下是什么？（对于此问题及后续问题，假设两个矩阵  $X \in \mathbb{R}^{m \times n}$  和  $Y \in \mathbb{R}^{n \times p}$  相乘的计算成本为  $O(mnp)$ ）。

(b) (1 分) 简答：如果我们按照公式中的矩阵乘法实现滑动窗口，空间复杂度在  $N$  和  $w$  的情况下是什么？

(c) (4 分) 伪代码：编写伪代码或数学表达式，用于一个函数，它接收查询、键、值和窗口大小  $w$ ，并计算  $X'$ ：

SCALED DOT PRODUCT ATTENTION( $Q, K, V, d_k, w$ )

你的伪代码/数学表达式必须具有比上述天真矩阵乘法方法更低的渐进计算复杂度。你的解决方案可以并且应该包括 for 循环。假设可以访问一个函数 `softmax(x)`，该函数对向量  $x$  应用 softmax，以及一个函数 `tensor(·)`，用于构造任意形状的向量、矩阵、张量。

(d) (1 分) 简答：你的伪代码的空间复杂度是多少（以  $N$  和  $w$  表示）？

(e) (1 分) 简答：你的伪代码的时间复杂度是多少（以  $N$  和  $w$  表示）？

通过这部分问题，要求理解滑动窗口注意力的实现及其计算复杂度，重点在于如何在序列上应用局部注意力窗口，并分析其效率。

## 解答

CMU 学生的 a~e 答案：

$O(N^2)$   
assuming  $d_k$   
constant.

$N$   
assuming  $d_m$   
constant.

备注：关于严格按照公式计算的空间复杂度，这里CMU学生应该错了，空间复杂度应该是 $N^2$ ，因为要存储 $Q \cdot K$ 矩阵的结果维度也就是 $N^2$ 。

**Input:**  $Q, K, V, d_k, w$

**Output:**  $\text{softmax}((QK^T/\sqrt{d_k}) + M)V$

```
1:  $QK \leftarrow \text{tensor}(N \times (\frac{w}{2} + 1))$ 
2: for  $i = 1$  to  $N$  do
3:   for  $j = \max(0, i - (\frac{w}{2} + 1))$  to  $i$  do
4:      $QK_{\{i, i-j\}} \leftarrow (Q_{\{i, :\}} \cdot K_{\{j, :\}}) / \sqrt{d_k}$ 
5:  $QK \leftarrow \text{softmax}(QK)$ 
6:  $QKV \leftarrow \text{tensor}(N \times d_k)$ 
7: for  $i = 1$  to  $N$  do
8:   for  $j = 1$  to  $d_k$  do
9:      $idx0 \leftarrow \max(0, i - (\frac{w}{2} + 1))$ 
10:     $QKV_{i,j} \leftarrow QK \cdot V_{\{idx0:i, j\}}$ 
return  $(QKV)$ 
```

$Nd_k \rightarrow N$   
assuming  
constant  $d_k$ .

$O(Nw)$   
assuming  
constant  $d_k$ .

GPT解答：

### 3.1(a) 公式版的时间复杂度

**问题：**如果我们严格按照公式  $X' = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$  实现滑动窗口注意力，时间复杂度是多少？

解答:

1. 计算  $QK^T/\sqrt{d_k}$ :

- $Q \in \mathbb{R}^{N \times d_k}$ ,  $K \in \mathbb{R}^{N \times d_k}$ , 计算  $QK^T$  需要  $N^2$  次点积, 每个点积的复杂度是  $O(d_k)$ 。
- 时间复杂度:  $O(N^2 d_k)$ 。

• 加上掩码矩阵  $M$ :

- $M$  是  $N \times N$  矩阵, 将它与  $QK^T$  相加的时间复杂度是  $O(N^2)$ 。

• 计算 softmax:

- Softmax 应用在  $N \times N$  矩阵上, 因此时间复杂度是  $O(N^2)$ 。

• 计算  $AV$ :

- $A \in \mathbb{R}^{N \times N}$ ,  $V \in \mathbb{R}^{N \times d_k}$ , 矩阵乘法的时间复杂度是  $O(N^2 d_k)$ 。

总的时间复杂度:

$$O(N^2 d_k) + O(N^2) = O(N^2 d_k)$$

如果  $d_k$  是常量, 则时间复杂度简化为:

$$O(N^2)$$

---

### 3.1(b) 公式版的空间复杂度

问题: 如果我们严格按照公式实现滑动窗口注意力, 空间复杂度是多少?

解答:

1. 存储  $Q$ 、 $K$ 、 $V$ :

- $Q$ 、 $K$ 、 $V$  的大小分别是  $N \times d_k$ , 所以需要的存储空间是  $O(Nd_k)$ 。

• 存储  $QK^T$ :

- $QK^T$  是一个  $N \times N$  的矩阵, 因此需要  $O(N^2)$  的空间。

• 存储 softmax 的中间结果和  $A$ :

- softmax 操作需要存储  $A \in \mathbb{R}^{N \times N}$ , 即  $O(N^2)$ 。

• 存储  $X'$ :

- $X' \in \mathbb{R}^{N \times d_k}$ , 因此存储  $X'$  需要  $O(Nd_k)$ 。

总的空间复杂度:

$$O(N^2) + O(Nd_k) = O(N^2)$$

在  $d_k$  是常量的情况下, 空间复杂度简化为:

$$O(N^2)$$

### 3.1(c, d, e) 代码版实现、空间复杂度、时间复杂度

**问题：**编写伪代码，优化滑动窗口注意力计算，使其具有比天真矩阵乘法方法更低的渐进计算复杂度。

**解答：**

优化滑动窗口注意力的关键在于减少每个位置与序列中每个元素的交互，而仅与滑动窗口内的元素进行交互，从而将复杂度从  $O(N^2)$  降到  $O(Nw)$ 。

代码如下：

```
import torch
import torch.nn.functional as F

def sliding_window_causal_attention(Q, K, V, d_k, w):
    """
    实现滑动窗口的因果注意力机制
    Q, K, V: 输入的查询、键和值张量，形状为 (N, d_k)
    d_k: 每个向量的维度
    w: 滑动窗口大小
    返回: 经过滑动窗口因果注意力机制处理后的输出 x_prime
    """
    N = Q.size(0) # 序列长度
    X_prime = torch.zeros_like(Q) # 初始化输出矩阵

    for i in range(N):
        # 只向前看，因此窗口从 max(0, i - w // 2) 到 i
        window_start = max(0, i - w // 2)
        window_end = i # 只看当前位置及之前的位置

        # 提取键和值的窗口部分
        K_window = K[window_start:window_end+1] # 窗口内的键
        V_window = V[window_start:window_end+1] # 窗口内的值

        # 计算 Q[i] 与 K_window 的点积，并除以 sqrt(d_k)
        scores = torch.matmul(Q[i], K_window.T) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))

        # 创建因果掩码，确保当前位置只关注前面的元素
        mask = torch.full_like(scores, float('-inf')) # 初始化为 -inf
        mask[i - window_start:] = 0 # 当前位置及其前面的部分设置为有效

        # 计算 softmax 并应用掩码
        attention_weights = F.softmax(scores + mask, dim=-1)

        # 计算加权的输出
        X_prime[i] = torch.matmul(attention_weights, V_window)
```

```

        return X_prime

# 测试
N = 6 # 序列长度
d_k = 4 # 向量维度
w = 4 # 滑动窗口大小

# 随机生成查询、键和值
Q = torch.rand(N, d_k)
K = torch.rand(N, d_k)
V = torch.rand(N, d_k)

output = sliding_window_causal_attention(Q, K, V, d_k, w)
print(output)

# 我测试的运行结果:
tensor([[0.0323, 0.9104, 0.7727, 0.9223],
        [0.3455, 0.5752, 0.7489, 0.8813],
        [0.3887, 0.2673, 0.6308, 0.3600],
        [0.8112, 0.2752, 0.5635, 0.5418],
        [0.2759, 0.4999, 0.2751, 0.8016],
        [0.9672, 0.7985, 0.7158, 0.2517]])

```

## 代码流程总结:

1. **初始化**: 为输出矩阵 `x_prime` 分配与查询向量  $Q$  相同形状的零矩阵, 用于存储最终的注意力结果。
2. **滑动窗口定义**: 对每个序列位置  $i$ , 确定滑动窗口的范围, 该窗口的起点为  $\max(0, i - w // 2)$ , 终点为  $i$ 。该窗口确保当前元素  $i$  只能关注自己和前面  $w/2$  个元素。
3. **提取窗口内的键和值**: 从序列中提取滑动窗口范围内的键  $K$  和值  $V$ , 用于后续的计算。
4. **计算点积注意力分数**: 对查询向量  $Q[i]$  与窗口内的键  $K$  进行点积计算, 并且进行缩放操作 (除以  $\sqrt{d_k}$ )。
5. **应用因果掩码**: 生成掩码矩阵, 屏蔽窗口外的元素, 确保当前元素  $i$  只关注自己及前面  $w/2$  个元素。
6. **Softmax归一化**: 对注意力分数应用 softmax 操作, 得到注意力权重。
7. **加权求和**: 使用注意力权重对窗口内的值  $V$  进行加权求和, 计算最终的注意力输出。
8. **返回输出**: 最终返回经过滑动窗口注意力机制处理后的输出矩阵  $X'$ 。

## 时间复杂度分析:

对于每个序列位置  $i$ , 计算点积、掩码应用、Softmax以及加权求和的操作都限制在一个大小为  $w/2$  的窗口内。因此, 对于序列长度  $N$  和窗口大小  $w$ , 代码的整体时间复杂度为:

- **每个位置的时间复杂度**: 在窗口  $w$  内进行矩阵点积和加权求和, 每次计算的时间复杂度为  $O(wd_k)$ 。

- **整体时间复杂度**：对于整个序列，每个序列位置的计算都限制在窗口大小  $w$  内，因此总的时间复杂度为：

$$O(Nwd_k)$$

其中  $N$  是序列长度， $w$  是窗口大小， $d_k$  是查询和键的向量维度。

## 空间复杂度分析：

- **存储键、查询和值**：每个键、查询和值的大小为  $N \times d_k$ ，因此存储  $Q$ 、 $K$  和  $V$  的空间复杂度为  $O(Nd_k)$ 。
- **存储掩码矩阵**：每个位置的掩码矩阵大小为  $w/2$ ，但并不需要存储全局的  $N \times N$  掩码矩阵，因此掩码的空间复杂度是  $O(w)$ 。
- **存储中间结果**：每个位置的注意力分数和权重只涉及窗口内的元素，大小为  $O(w)$ 。
- **总空间复杂度**：代码的空间复杂度主要由存储  $Q$ 、 $K$ 、 $V$  以及中间计算的结果决定，最终为：

$$O(Nd_k + w)$$

其中  $N$  是序列长度， $d_k$  是向量维度， $w$  是窗口大小。