Chapter 2: Working with Text

2024年11月2日 16:59

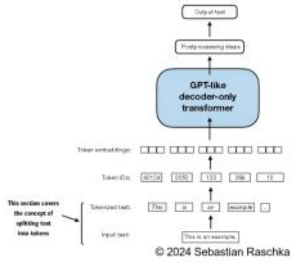
2.1 word embeddings

将单词转换成固定维度向量的方法

将每个单词表示为一个向量，使得相似的词语在向量空间中也比较接近。这种方法将词语的语义信息转换为数值格式，便于模型进行计算和分析。

2.2 Tokenizing text

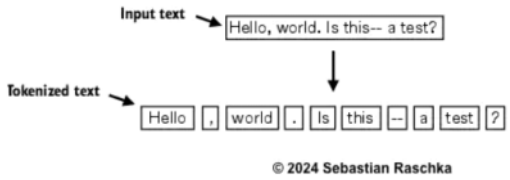
文本的“分词”或“标记化”过程,将文本拆分成基本的分析单元（token），以便模型理解和处理。



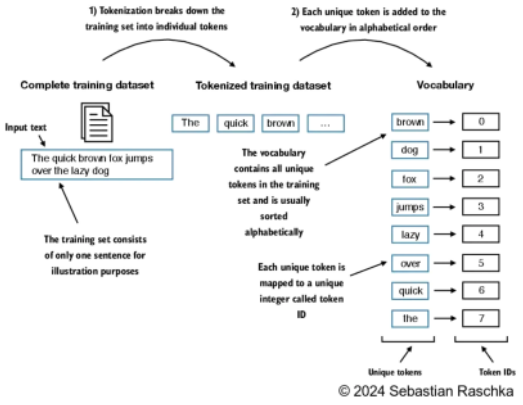
1. 导包
2. 下载想处理的数据raw text

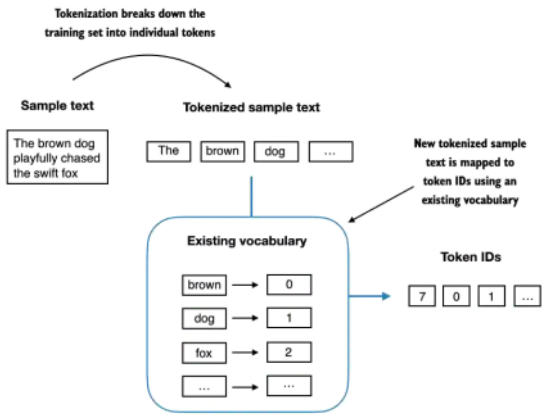
1.删除空格字符串

2.可以按逗号（,）和句号（.）等其他标点符号进行分割。

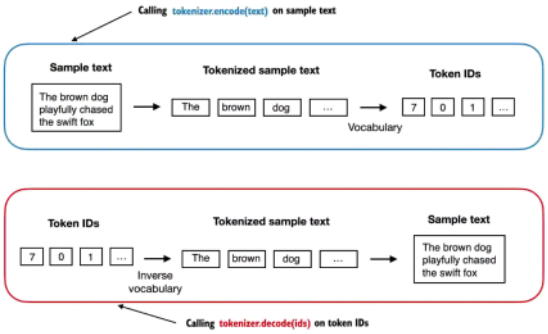


2.3 Converting tokens into token IDs





建立一个包含所有词汇的词汇表，并按照字典序排序，得到token-integer对



2.4 Adding special context tokens

It's useful to add some "special" tokens for unknown words and to denote the end of a text 特殊的token标志文本的结束/额外的文本内容

特殊标记

1.[BOS] (Beginning of Sequence)：表示文本的开始，帮助模型理解输入的起始位置。

2.[EOS] (End of Sequence)：文本的结束。常用于将多个不相关的文本连接在一起，例如不同的维基百科文章或书籍。它指示模型何时停止生成文本。

3.[PAD] (Padding)：在使用批处理（batch）训练时，输入文本的长度可能不同。使用填充标记可以将较短的文本扩展到与最长文本相同的长度，从而形成统一的输入形状。这在训练时很重要，因为深度学习模型通常需要固定大小的输入。

4.[UNK] (Unknown)：词汇表中未包含的单词。在某些分词器中，当遇到无法识别的词时，会使用此标记。

GPT-2只使用一个特定的标记 <|endoftext|>

类似于 [EOS] 标记，用于指示文本的结束。

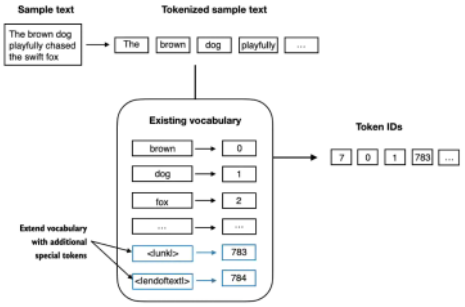
也用作填充标记，因为在训练过程中，使用掩码（mask）来处理批量输入时，模型不会关注填充的标记，因此填充的具体内容并不重要。

- 某个token没出现在词汇表中："<|unk|>"来表示未知的token
- "<|endoftext|>" 表示文本的结束，或者训练集中多篇文章的分割点

应用

在独立文本之间使用 <|endoftext|> 标记：

该标记可以帮助模型理解何时开始处理新的文本段落，从而更好地管理上下文。



enumerate函数：

主要用于在迭代列表（或其他可迭代对象）时，同时获取元素的索引和值。

enumerate(iterable, start=0)

- **iterable**：一个可迭代对象，例如列表、元组、字符串等。
- **start**：可选参数，指定索引的起始值，默认是 0。

可用于**创建词汇表**：

- all_tokens 是一个包含所有唯一标记的列表。
- 使用 enumerate(all_tokens)，为 all_tokens 中的每个标记生成一个索引。

2.5 BytePair encoding - 处理不在词汇表中的单词

BPE 分词器

GPT-2 采用字节对编码（BPE）分词器，而不是使用 [UNK] 标记处理未在词汇表中的单词。BPE 将单词分解为子词单元，从而更灵活地处理新词或组合词。

BPE通过将不常见的词分解成更小的子词单元或字符，使得模型能够理解这些未知的词汇。

例如，如果 GPT-2 的词汇表中没有 "unfamiliarword" 这个词，它可能会将其分解为 ["unfam", "iliar", "word"]，或者根据其训练的 BPE 合并规则进行其他分解。

1.原始BPE分词器

2.OpenAI 的开源 tiktoken 库来实现 BPE 分词器。核心算法用 Rust 语言实现，提高计算性能。在大规模文本上处理速度更快，约5倍

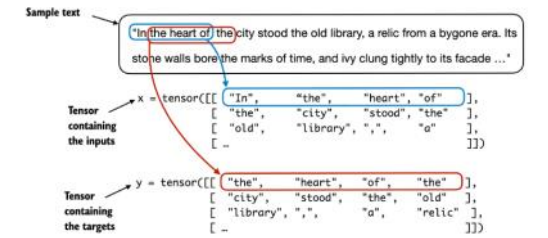
其他不同的BPE：

1. Using BPE from `tiktoken`

Using the original BPE implementation used in GPT-2
Using the BPE via Hugging Face transformers
A quick performance benchmark

2.6 滑动窗口进行数据采样

一次移动一个位置



1. PyTorch 的 torch.tensor：**张量**是 PyTorch 中的基本数据结构，类似于 NumPy 的数组，但提供了 GPU 加速和自动求导的能力。张量可以是任意维度的数组（0维标量、1维向量、2维矩阵，甚至更高维度）
   torch.tensor(input_chunk) 将输入列表转换为张量，使得后续的模型训练可以高效地使用这些数据。
   通过将数据存储为张量，可以利用 PyTorch 的高效计算能力和 GPU 加速。
2. stride 是指在使用滑动窗口方法时，窗口每次移动的步幅大小。它决定了在文本分块时每个新块的起始位置。
   适当的 stride 值可以在保持上下文信息和减少训练样本数量之间找到平衡。
3. epoch 是指模型在整个训练数据集上完整训练一次的过程。在一个 epoch 中，模型会遍历所有的训练样本。训练过程通常包括多个 epoch，以便模型逐渐调整参数以适应训练数据。在每个 epoch 结束时，通常会在验证集上评估模型性能，以监测过拟合和调整学习率等超参数
4. **batch**（批次）是指一次性送入模型进行训练的样本集合。每个批次可以包含多个样本，通常由用户在设置超参数时指定（例如，batch_size）
5. #shuffle：布尔值，决定是否在每个 epoch 开始时随机打乱数据，默认值为 True。
   #drop_last：布尔值，决定在批次不足时是否丢弃最后一个批次，默认值为 True。
6. stride很小，overlap很多可能会导致过拟合

   正向传播：对于每个批次，模型首先进行正向传播计算输出。
   计算损失：根据模型输出和真实标签计算损失。
   反向传播：通过反向传播算法计算梯度，并根据批次中所有样本的损失平均值来更新模型参数。

- 小批次（如 1 到 32）通常会使模型更频繁地更新参数，可能帮助模型更好地跳出局部最优解。
- 大批次（如 64 到 256）会减少训练的波动，可能导致更稳定的训练过程，但也可能使模型陷入局部最优解。

2.7 Creating token embeddings

嵌入层（embedding layer）实现.
嵌入层是神经网络中的一种特殊层，用于将离散的标记（例如词汇表中的ID）映射到连续的向量空间。
每个标记都有一个对应的向量表示，这些向量可以捕捉到标记之间的语义关系。

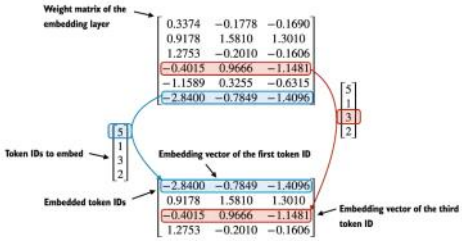
- **创建嵌入矩阵**：初始化一个嵌入层。
- **输入标记 ID**：准备要转换的标记 ID。
- **查找嵌入向量**：从嵌入层中获取对应的向量。
- **优化**：在训练过程中更新嵌入层的权重。

嵌入层的工作原理类似于一热编码加上全连接层的矩阵乘法。嵌入层的权重在模型训练过程中可以通过反向传播进行优化，这意味着它可以学习到不同标记之间的相对关系和语义信息。

grad_fn=<EmbeddingBackward0> 是 PyTorch 中用于跟踪计算图的一部分
grad_fn 是一个属性，用于记录一个张量的创建过程，特别是它是通过哪个操作生成的。这个属性在自动求导过程中很重要，因为它帮助 PyTorch 知道如何计算梯度。

<EmbeddingBackward0> 的含义

- 嵌入层是用于将离散的标记 ID 转换为连续向量表示的层。
  Backward：指示这个操作可以进行反向传播（backpropagation），即可以计算梯度以更新模型参数。
  0：这是一个索引，表示这是嵌入层的第一个（或唯一的）反向传播操作的实例。这个索引可以在有多个操作的情况下帮助区分不同的计算图节点。
- **自动求导**：当你在模型中调用 .backward() 方法进行反向传播时，PyTorch 将使用 grad_fn 来沿着计算图向后传播梯度。这确保了所有在前向传播中使用的参数都能正确更新。
- **调试**：查看 grad_fn 可以帮助你理解某个张量是如何生成的，尤其在调试模型时，这可以提供有价值的信息。



2.8 Encoding word positions

嵌入层将ID变为相同的向量表示，没有考虑词语位置

Positional embeddings + token embedding vector = input embedding

BONUS:Understanding the Difference Between Embedding Layers and Linear Layers

2024年11月4日 18:15

在 PyTorch 中，嵌入层（embedding layers）和线性层（linear layers）在功能上可以实现相似的效果，即通过矩阵乘法将输入转换为输出，但嵌入层在计算效率方面更具优势。

- 将 One-Hot 编码的输入与嵌入层的权重矩阵相乘时，由于大多数元素都是 0，乘法结果几乎等同于查找操作。这意味着只有索引为 1 的元素对最终结果有贡献，其余的乘法（与 0 的乘法）不会影响结果。
- 因此，矩阵乘法的操作实际上只是从嵌入矩阵中查找相应的嵌入向量。
- 尽管矩阵乘法在数学上是等价的，但如果嵌入矩阵非常大，那么这种做法可能会变得低效。因为在执行乘法时，系统需要处理大量与 0 相乘的计算，这些计算是多余的。
- 嵌入层通过直接查找嵌入向量来实现同样的功能，这样就避免了不必要的乘法运算，从而提高了计算效率。

3. 嵌入层 vs. 线性层

特性	嵌入层	线性层
输入	整数索引（类别 ID）	连续稠密向量
核心操作	查表操作	矩阵乘法
应用场景	NLP（单词嵌入、分类嵌入）	全连接网络、分类器、回归器
优化方式	优化嵌入矩阵	优化权重矩阵和偏置
参数矩阵	索引直接指向嵌入矩阵的一部分	全部权重矩阵参与计算
计算效率	高效（查表操作，无矩阵运算）	依赖矩阵运算

- 嵌入层更适合处理离散类别数据，通过查找操作将类别映射为连续的稠密向量，常用于 NLP。
- 线性层是标准的全连接层，处理稠密向量，通过矩阵运算对数据进行线性变换，适用于大多数神经网络任务。

One-Hot 编码是一种常见的特征表示方法，尤其是在处理分类任务或自然语言处理时。它将一个分类变量（如类别标签或词汇表中的单词）表示为一个稀疏向量，其中只有一个元素为 1，其余元素为 0。

One-Hot 编码的关键特性

- **稀疏性**：向量中大部分元素为 0，仅有一个元素为 1。
- **独特性**：每个类别对应一个唯一的向量位置。
- **无序性**：它不引入类别之间的大小或顺序信息。

使用场景

1. **分类模型输入**：在 NLP 或图像分类中，将类别标签转换为 One-Hot 编码后输入模型。
2. **Embedding 层输入**：通常，One-Hot 编码是嵌入层（nn.Embedding）输入的基础。
3. **损失计算**：在多类别分类中，CrossEntropyLoss 不需要显式的 One-Hot 编码，但你可以用它计算逐类别损失。

PyTorch 中的注意事项

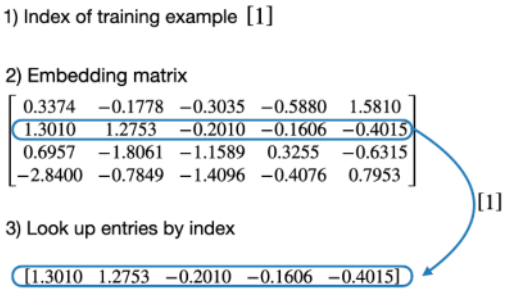
1. **one_hot 输出的类型**：输出张量通常是整型，可能需要使用 .float() 方法将其转换为浮点数以适配某些模型。
2. **不直接作为网络输入**：One-Hot 编码通常直接传递给嵌入层或映射为更低维的特征，减少内存消耗和计算开销。

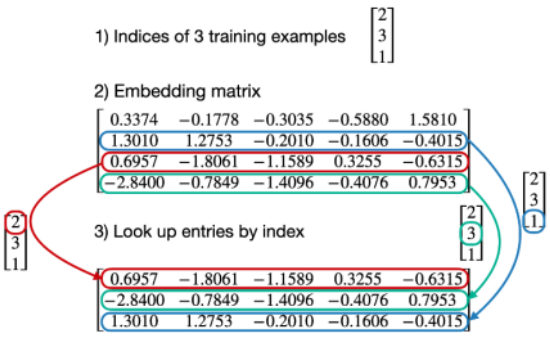
随机种子（Random Seed）：

- torch.manual_seed(123) 这一行代码设置了随机种子为 123。随机种子的作用是确保在多次运行相同代码时，生成的随机数序列是相同的。这在训练机器学习模型时非常重要，因为模型的初始化和数据的随机分割通常会影响最终结果。
- 通过设置随机种子，可以使得实验的结果在不同的运行中保持一致，从而方便调试和比较不同模型的性能。

权重初始化：

- 在嵌入层中，权重矩阵的行数等于类别的数量，列数等于嵌入维度。每一行对应一个类别的嵌入向量。
- 这些嵌入向量在模型训练开始时通常会被初始化为小的随机值。这是因为随机初始化可以帮助模型打破对称性，促进更好的学习。





- **One-Hot 编码的特性**：
  - One-Hot 编码是一种表示离散类别的方式，其中每个类别用一个向量表示，该向量的长度等于类别总数。在这个向量中，只有一个元素为 1（表示该类别的索引），其余元素均为 0。
  - 例如，类别 3 的 One-Hot 编码在类别总数为 5 时表示为 [0, 0, 1, 0, 0]。
- **矩阵乘法与查找**：
  - 在将 One-Hot 编码的输入与嵌入层的权重矩阵相乘时，由于大多数元素都是 0，乘法结果几乎等同于查找操作。这意味着只有索引为 1 的元素对最终结果有贡献，其余的乘法（与 0 的乘法）不会影响结果。
  - 因此，矩阵乘法的操作实际上只是从嵌入矩阵中查找相应的嵌入向量。
- **效率问题**：
  - 尽管矩阵乘法在数学上是等价的，但如果嵌入矩阵非常大，那么这种做法可能会变得低效。因为在执行乘法时，系统需要处理大量与 0 相乘的计算，这些计算是多余的。
  - 嵌入层通过直接查找嵌入向量来实现同样的功能，这样就避免了不必要的乘法运算，从而提高了计算效率。

Chapter 3: Coding Attention Mechanisms

2024年11月7日 16:37

3.1 长序列建模的问题

逐字逐句翻译文本不可行，因为不同目标语言的语法结构差异很大。
在 Transformer 模型出现之前，基于 RNN 的编码器-解码器架构 （encoder-decoder RNNs）用于机器翻译任务
编码器使用hidden state（一种神经网络的中间层）来处理源语言的tokens，生成整个input序列的压缩表示

3.2 注意力机制中的数据依赖捕捉

- 通过注意力机制，生成文本的解码器 选择性访问 编码器中的输入标记，特定的token在生成特定的输出时有不同的significance

Transformer的自注意力机制 突破了传统 RNN 的限制，它允许序列中的每个位置能够关注序列中其他位置的标记，赋予每个标记对序列中其他标记的重要性评分。这一评分决定了在生成当前标记时，哪些输入标记更重要。

3.3 使用自注意力关注输入的不同部分

3.3.1 一个没有可训练权重的简单自注意力机制

步骤 1：计算未经归一化的注意力得分

未经归一化的注意力权重被称为“注意力得分” "attention scores"，而归一化后的注意力得分（和为 1）则被称为“注意力权重” "attention weights"。

计算谁的context vector就把谁当做query

步骤2：归一化attention score w 使其总和为1

1. 直接计算
2. **Softmax** ：
   - 更好的处理极值
   - 训练时 更好的梯度性质

   输入值过大或过小时，可能会遭遇溢出（overflow）或下溢（underflow）的问题，导致数值不稳定。
3. PyTorch 内置的 softmax 实现

步骤3：计算上下文向量 (2) = sum (( ) * attention weights)

3.3.2 对所有的输入token计算注意力权重

推广到所有输入序列的token：

在self-attention中，首先需要计算注意力分数（attention scores），然后对其进行归一化处理，得到总和为1的注意力权重。这些注意力权重随后会用于通过输入的加权求和生成上下文向量（context vectors）

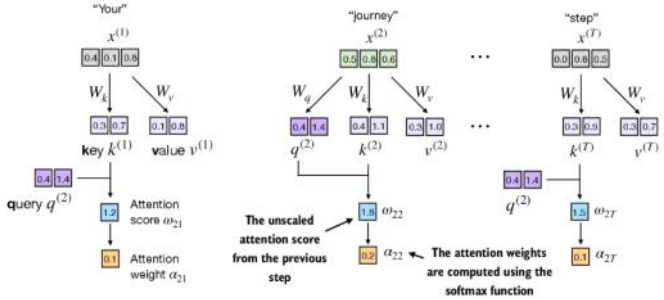
3.4 可训练权重的自注意力机制

3.4.1 一步步计算注意力权重

缩放点积注意力（scaled dot-product attention）。其核心思路类似于之前提到的注意力机制： 我们希望针对特定的输入元素，通过对输入向量的加权求和来计算上下文向量。为此，我们需要计算注意力权重。
与之前介绍的基础注意力机制相比，这种自注意力机制只有些许差异，最显著的变化是引入了训练用的**权重矩阵**，这些矩阵在模型训练过程中不断更新。这些可训练的权重矩阵非常关键，因为模型（特别是模型中的注意力模块attention module ）可以学习生成“优质”的上下文向量。

查询向量 Query vector: ( )= ( )
键向量 Key vector: ( )= ( )
值向量 Value vector: ( )= ( )

第二步：将得到的q依次和每个k相乘得到注意力评分：K*Q的转置



第三步： 归一化得到权重，不同之处在于现在我们将注意力得分除以嵌入维度 d_k 的平方根 ----√dk (i.e., d_k**0.5)来进行缩放 ：

- **第四步：** 计算对于q2的上下文向量： 注意力权重*value矩阵

3.4.2 实现紧凑的 SelfAttention 类

3.5 使用Causal Attention隐藏mask 未来单词

- 对角线以上的注意力权重会被屏蔽（mask），从而确保在计算当前输入的上下文向量时，模型不会使用到未来的 token。

3.5.1 应用casual attention mask（因果注意力掩码）

- 确保了**每个新单词的预测只依赖于前面的单词。**
- 为此，对于给定的每个 token，我们会屏蔽掉未来的 token（即在输入文本中位于当前 token 之后的 token）。
- **可以使用 PyTorch 的 tril 函数来创建一个掩码矩阵，该矩阵的对角线及以下元素为 1，对角线以上的元素为 0。**
- 将注意力权重乘以掩码，屏蔽对角线以上的部分

- 如果在 softmax 之后应用掩码操作，会破坏 softmax 创建的概率分布，因为 softmax 的结果应该总和为 1。掩码会改变这一分布，需要重新归一化，可能会导致计算复杂度增加并产生非预期的效果。
- 下面来归一化注意力权重

3.5.2 用 Dropout 屏蔽额外的注意力权重

- 在训练过程中，我们还可以应用 Dropout 以减少过拟合。
- 可以在多个位置应用 Dropout：
  - 比如，在计算注意力权重后，或在将注意力权重与值向量相乘之后（计算上下文向量）。
  - 在这里，我们将在计算注意力权重后应用 Dropout，因为这种做法更为常见。
- 在本例中，我们使用 50% 的 Dropout 率，这意味着随机屏蔽掉一半的注意力权重（在训练 GPT 模型时，我们通常使用较低的 Dropout 率，如 0.1 或 0.2）

dropout**只用于训练，不用于推理**

3.6 将single-head attention扩展到multi-head attention

3.6.1 堆叠多个单头注意力层

stack 多个single-head attention 可以得到multi-head attention

- 多头注意力的核心思想是让**注意力机制并行运行多次 在不同的学习过的线性投影上**。这样可以让模型在不同的位置上同时关注来自不同表示子空间的信息。

3.6.2 使用权重拆分实现多头注意力

- 虽然以上实现是一个直观且完整的多头注意力机制实现（通过包装单头注意力的CausalAttention 实现），我们可以编写一个独立的 MultiHeadAttention 类来实现相同的功能。
- 在这个独立的 MultiHeadAttention 类中，我们不将单个注意力头进行串联，而是：
  - 创建一个单独的 W_query、W_key 和 W_value 权重矩阵。
  - 然后将这些权重矩阵划分成每个注意力头的单独矩阵。

在上面的 MultiHeadAttention 类中，我们添加了一个线性投影层（self.out_proj）。这只是一个不会改变维度的线性变换。在大型语言模型（LLM）实现中使用这样的投影层是一种常见的惯例，但它并非绝对必要（近期研究表明，去掉这个层不会影响建模性能，详见本章末尾的进一步阅读部分）。

Understanding PyTorch Buffers

2024年11月9日 18:18

理解 PyTorch 缓冲区

本质上，PyTorch 缓冲区是与 PyTorch 模块或模型相关联的张量属性，它们与参数类似，但不同之处在于缓冲区不会在训练过程中更新。

在使用 GPU 计算时，PyTorch 缓冲区尤其有用，因为它们需要和模型参数一起在设备之间传输（例如从 CPU 到 GPU）。与参数不同的是，缓冲区不需要计算梯度，但它们仍然需要在正确的设备上，以确保所有计算的准确性。

在第 3 章中，我们通过 self.register_buffer 使用了 PyTorch 缓冲区。书中对此进行了简要解释，但其概念和用途可能并不直观，因此本代码笔记提供了更详细的解释，并附有实际示例。

示例：不使用缓冲区

假设我们有以下代码，这是基于第 3 章的代码。此版本已被修改以不包含缓冲区，展示了 LLM 中使用的因果自注意力机制的实现。

Comparing Efficient Multi-Head Attention Implementations

2024年11月18日 14:53

1) CausalAttention MHA wrapper class from chapter 3

2) The multi-head attention class from chapter 3

3) An alternative multi-head attention with combined weights

- `MultiHeadAttentionCombinedQKV` 类与第 3 章使用的 `MultiHeadAttention` 类的主要区别在于： `MultiHeadAttentionCombinedQKV` 使用一个单独的权重矩阵：`self.qkv = nn.Linear(d_in, 3 * d_out, bias=qkv_bias)` 而不是分别使用三个独立的权重矩阵：
  - `self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)`
  - `self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)`
  - `self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)`
- 在这种实现中，`self.qkv` 将 `self.W_query`、`self.W_key` 和 `self.W_value` 的权重矩阵组合为一个矩阵，从而能够在一个步骤中同时计算 查询 (Query)、键 (Key) 和 值 (Value)。
- Using `q, k, v = qkv.unbind(0)`, we obtain the individual query, key, and value tensors, which are then used similarly to the query, key, and value tensors in the `MultiHeadAttention` class in chapter 3

4) Multi-head attention with Einsum

5) Multi-head attention with PyTorch's scaled dot product attention and FlashAttention

6) PyTorch's scaled dot product attention without FlashAttention¶

7) Using PyTorch's torch.nn.MultiheadAttention

8) Using PyTorch's torch.nn.MultiheadAttention with `scaled_dot_product_attention`

9) Using PyTorch's FlexAttention

Chapter 4: Implementing a GPT model from Scratch To Generate Text

2024年11月10日 8:31

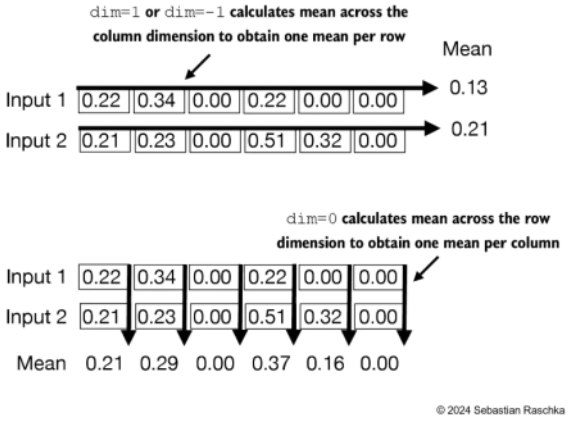
4.1 编写大型语言模型（LLM）架构

- 第 1 章介绍了像 GPT 和 Llama 这样的模型，它们逐词生成文本，基于最初的 Transformer 架构中的解码器部分。因此，这些 LLM 通常被称为“类似解码器”的 LLM。与传统深度学习模型相比，LLM 规模更大，主要体现在庞大的参数数量上，而非代码量。在构建 LLM 的架构时，我们会发现许多部分是重复的。

- vocab_size：词汇量大小为 50,257 个词，由第 2 章中讨论的 BPE 分词器支持。
- context_length：模型的最大输入 token 数量，通过第 2 章中讲到的positional embeddings实现。
- emb_dim：输入 token 的嵌入维度，将每个输入 token 转换为 768 维向量。
- n_heads：多头注意力机制multi-head attention中的注意力头数量，由第 3 章实现。
- n_layers：模型中 Transformer 块的数量，我们将在接下来的部分中实现。
- drop_rate：dropout 概率，即在训练期间丢弃 10% 的隐藏单元来缓解过拟合，第 3 章中已介绍。
- qkv_bias：决定多头注意力机制中的线性层是否应包括偏置向量以计算查询（Q）、键（K）和值（V）张量。我们会关闭此选项，这是现代 LLM 的常见做法；不过，我们会在第 5 章中重新加载 OpenAI 预训练的 GPT-2 权重时重新探讨这一点。

4.2 使用层归一化 标准化 激活值

- 层归一化（Layer Normalization，简称 LayerNorm，由 Ba 等人提出于 2016 年）通过将神经网络层的激活值中心化到均值为 0，方差为 1 的分布来实现标准化。这种方法有助于稳定训练过程并加速权重的有效收敛。
- 层归一化在 Transformer 块中的多头注意力模块前后均会应用，我们稍后将实现这一功能；在最终输出层之前也会应用层归一化。



- 标准化过程 包括 减去均值 并 除以方差的平方根（标准差），从而将输入调整为均值为 0 和方差为 1 的分布-标准正态分布。

缩放与偏移 Scale and shift

- 在执行标准化（-mean 除以标准差）的同时，LayerNorm 还添加了两个可训练参数：缩放参数（scale）和偏移参数（shift）。初始缩放值为 1，偏移值为 0，这对初始结果没有影响；
- 然而，这些缩放和偏移参数可以在训练期间自动调整，以改进模型的性能。模型可以根据数据特点学习合适的缩放和偏移，以更好地处理输入数据。
- 此外，在计算方差的平方根之前，我们会加上一个很小的值（eps），以避免方差为 0 时的除零错误。

有偏方差Biased variance

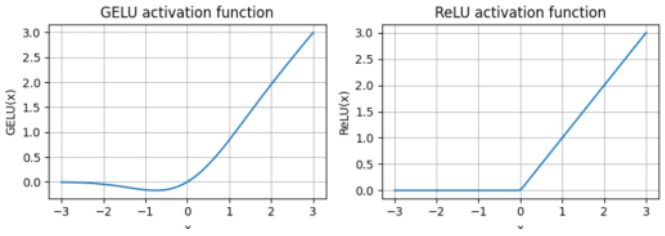
- 在上面的方差计算中，设置 unbiased=False 意味着我们使用公式 Σ (− ¯)2 Σi(xi−x¯)2n 来计算方差，其中 n 是样本数量（在此例中为number of feature 或列数）。此公式未使用 Bessel's correction 贝塞尔校正（分母使用 n-1），因此提供的是方差的有偏估计。
- 对于嵌入维度较大的 LLM 来说，使用 n 和 n-1 之间的差异很小。
- 然而，由于 GPT-2 在训练过程中使用了Biased variance，因此我们也在层归一化中采用了这种设置，以便与后续章节中加载的pretrained weights兼容。
- 让我们实际试用一下 LayerNorm 的实现。

4.3 实现具有 GELU 激活的前馈网络

- 在本节中，我们实现一个用于 LLM 中 Transformer 块的小型神经网络子模块。
- 从activation function开始
- 在深度学习中，ReLU（整流线性单元Rectified Linear Unit）激活函数因其简单和在各种神经网络架构中的有效性而被广泛使用。
- 然而，在 LLM（大型语言模型）中，除了传统的 ReLU 之外，还常用到其他激活函数，其中两个显著的例子是 GELU（**高斯误差线性单元Gaussian Error Linear Unit**）和 SwiGLU（Swish 门控线性单元Swish-Gated Linear Unit）。
- GELU 和 SwiGLU 是更复杂的平滑激活函数，分别结合了高斯和 sigmoid 门控线性单元，通常在深度学习模型中提供更好的性能，而不像 ReLU 只是简单的分段线性函数。

GELU的定义与实现

- GELU 函数（由 Hendrycks 和 Gimpel 提出于 2016 年）有多种实现方式，其精确定义为：GELU(x)=x· Φ(x)， 其中，Φ(x) 是标准高斯分布的累积分布函数。
- 在实际应用中，为了节省计算成本，常用以下近似实现 GELU( )≈0.5· · (1+tanh[2 − − − √· (+0.044715· 3)])GELU(x)≈0.5· x· (1+tanh [2 π · (x+ 0.044715· x3)]) 原始的 GPT-2 模型也使用了这种近似实现。



- ReLU 是一种分段线性函数，对于正数直接输出输入值，对于负数输出 0。相比之下，GELU 是一种平滑的非线性函数，在负数输入上仍然有非零梯度（大约 -0.75 时为零），其行为与 ReLU 相似但更平滑。

4.4 添加快捷连接shortcut connections

- 接下来，让我们讨论快捷连接（shortcut connections）的概念，也称为跳跃连接或残差连接（skip or residual connections）。
- 最初，快捷连接是为深度网络（尤其是计算机视觉中的残差网络）提出的，目的是缓解梯度消失问题mitigate vanishing gradient problems。在深层网络中，梯度随着层数的增加逐渐衰减，从而影响模型的学习能力。通过创建shortcut connections，**网络为梯度提供了一条更短的路径，使其可以在网络中更容易地传播。**
- 实现这种连接的方式是将某一层的输出添加到后面某层的输出上，通常会跳过中间的一层或多层。这样做可以在网络中提供直接的梯度流通路径。
- 这种设计可以通过以下一个简单的网络示例进行说明。

4.5 在 Transformer 块中连接attention和linear layers

- 在这一节中，我们将前面的概念结合成一个所谓的 Transformer 块。
- 一个 Transformer 块结合了上一章中的multi-head attention module、linear layers，以及前面部分实现的feed forward neural network。
- 此外，Transformer 块还使用了 dropout 和shortcut connection。

4.6 编写 GPT 模型代码

- 我们已经接近目标了：现在将 Transformer 块插入本章开头构建的架构中，以获得可用的 GPT 架构。
- 请注意，Transformer 块会重复多次；在最小的 124M 参数 GPT-2 模型中，我们重复这个块 12 次。

4.7 生成文本

- 像我们上面实现的 GPT 模型这样的大型语言模型（LLM）用于逐词生成文本。

- 在**贪心解码greedy decoding** 中，模型在每一步选择具有最高概率的单词（或 token）作为下一个输出（最高的对数值对应于最高的概率，因此我们实际上甚至不需要显式计算 softmax 函数）。
- 在下一章中，我们将实现一个更高级的 generate_text 函数。
- 下图展示了 GPT 模型在给定输入上下文后，生成下一个词 token 的过程。

FLOPS Analysis

- FLOPs (Floating Point Operations Per Second) 每秒浮点运算次数, 通过计算神经网络模型执行的浮点运算次数来衡量其计算复杂度。高 FLOPs 通常表示模型计算和能耗强度较高。

Chapter 5: Pretraining on Unlabeled Data

2024年11月13日 9:16

在本章中，我们将实现一个训练循环 training loop 以及基本的模型评估代码，以便对大型语言模型 (LLM) 进行预训练。最后，我们还将从 OpenAI 加载公开的预训练权重并将其导入到我们的模型中。

5.1 生成式文本模型的评估

- 首先，我们简要回顾使用上一章中的代码初始化 GPT 模型。然后，我们讨论用于**评估 LLM 的基本评价指标**，最后将这些指标应用到训练和验证数据集上。

5.1.2 计算文本生成损失：交叉熵cross-entropy和困惑度perplexity

- 假设我们有一个 inputs 张量，包含 2 个训练样本（行）的 token ID。
- 对应于 inputs，targets 包含了我们希望模型生成的目标 token ID。

我们关注目标索引对应的tokens概率。目标是将这些概率值尽量增大，尽可能接近1。在数学优化中，通常更容易最大化 **概率分数的对数 logarithm of the probability score** ，而不是直接最大化概率。

- 目标是通过优化模型权重，让average log probability越大越好
- 因为取了log，最大的possibility是0，我们目前离0 很远
- 在深度学习中，我们不直接最大化平均对数概率，而是最小化负的平均对数概率。例如，不是最大化-10.7722让他接近0，而是最小化10.7722让他接近0
- 在这种情况下，取负后的对数值（如10.7722）被称为**交叉熵损失cross-entropy**。
- **Perplexity只是交叉熵损失的指数形式**，被认为更具解释性，它表示模型在每一步不确定的 有效词汇大小 effective vocabulary size that the model is uncertain about at each step（下面的例子中，是48725个单词或tokens）。
- 换句话说，perplexity衡量模型预测的概率分布与数据集中实际分布的匹配度。类似于损失值，较低的困惑度表明模型预测更接近真实分布。

5.1.3 计算训练集和验证集的损失training and validation set losses

5.2 Training an LLM

- 从上面的结果来看，模型在训练开始时生成了无法理解的词语字符串，而在训练接近尾声时，已经能够生成基本符合语法的句子。
- 然而，通过观察训练集和验证集的损失值，我们可以看出模型开始出现**过拟合**。
- 如果检查模型在训练后期生成的部分文本，会发现其中一些内容是直接从训练集中逐字复制的——模型仅仅是记住了训练数据。
- 稍后我们将介绍一些解码策略，可以在一定程度上减轻mitigate这种记忆化现象。
- 需要注意的是，之所以会发生过拟合，是因为我们使用的训练集非常小，且多次重复迭代。

5.3 Decoding strategies to control randomness

- generate_text_simple 函数 依次上层一个token/word，依据最大可能性得分

- 介绍两种被称为解码策略的概念，用于修改 generate_text_simple 函数：温度缩放 和 Top-k 采样 **temperature scaling and top-k sampling。**
- 这些策略可以让模型控制生成文本的随机性和多样性。

5.3.1 Temperature scaling-- logits 除以一个大于 0 的数值temperature。

- 在之前，我们总是使用 torch.argmax 选取概率最高的 token 作为下一个 token。
- 为了增加变化，我们可以使用 torch.multinomial(probs, num_samples=1) 从概率分布中进行采样。
- 在这里，每个索引被选中的概率对应于输入张量中的概率。

- 为了避免总是选择最可能的 token，我们可以通过 torch.multinomial(probas, num_samples=1) 从 softmax 分布中进行采样，决定下一个 token。
- 为了便于说明，假设我们使用原始的 softmax 概率对下一个 token 进行 1000 次采样：

- 温度缩放 temperature scaling的概念来控制分布和选择过程，指的是将 **logits 除以一个大于 0 的数值temperature**。
- temperature大于 1 时，在应用 softmax 后，token 概率会更均匀分布；temperature小于 1 时，在应用 softmax 后，概率分布会更自信（更陡峭或尖峰）--会一直选择。

5.3.2 Top-k sampling

- 为了能够在提高输出多样性（higher temperature）的同时减少生成不合逻辑句子的可能性，我们可以将采样限制在前 k 个最可能的 token 中：

5.3.3 修改Modify文本生成函数

5.4 Loading and saving model weights in PyTorch

- 通常，我们会使用自适应优化器adaptive optimizers，如 Adam 或 AdamW，而不是常规的 SGD 来训练大型语言模型（LLM）。
- adaptive optimizers会为每个模型权重存储额外的参数，因此在计划稍后继续预训练时，保存这些参数也是有意义的

5.5 从 OpenAI 加载预训练权重

生成 deterministic确定的文本：

1. top_k=None + 不使用temperature
2. Top-k=1

Chapter 6: Finetuning for Text Classification

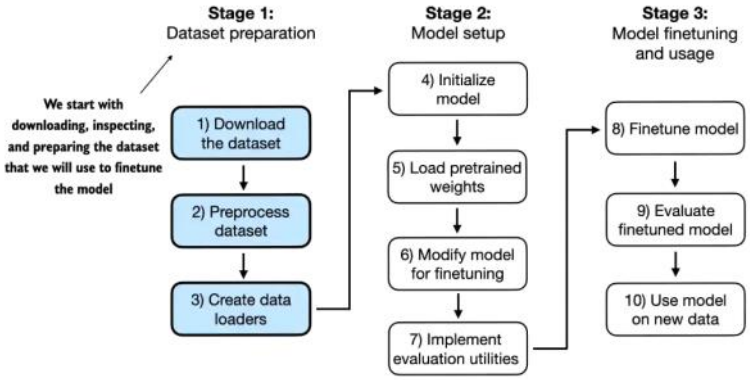
2024年11月14日 12:38

6.1 不同类别的微调

- 最常见的微调语言模型的方法是指令微调和分类微调。instruction-finetuning and classification finetuning

- 例如，在分类微调中，我们有一组特定的类标签（例如“垃圾邮件”和“非垃圾邮件”），模型可以输出这些类别。
- 分类微调的模型只能预测在训练过程中见过的类别（例如“垃圾邮件spam”或“非垃圾邮件”），而指令微调的模型通常能够执行多种任务。
- 我们可以将分类微调的模型看作是非常专门化的模型；实际上，创建一个专门化模型比创建一个在许多不同任务上表现良好的通用模型要容易得多。

6.2 Preparing the dataset



6.3 Creating data loaders

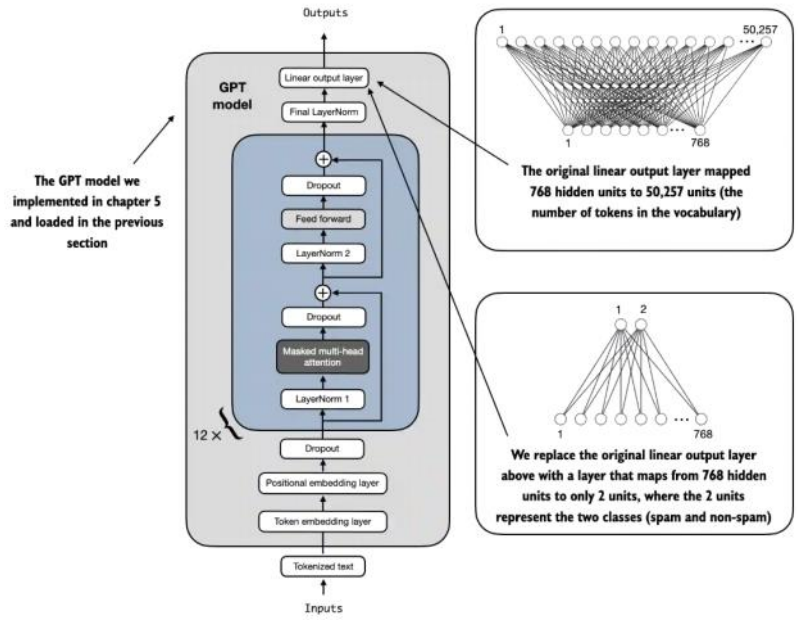
- 请注意，这些短信的长度不一致；如果我们想在一个批次中组合多个训练样本，则需要：
  - 将所有消息截断到数据集或批次中最短消息的长度，
  - 或将所有消息填充到数据集或批次中最长消息的长度。 我们选择了选项2，将所有消息填充到数据集中最长消息的长度。 pad all messages to the longest message

- 我们也将验证集和测试集填充到与最长训练序列相同的长度。
- 请注意，验证集和测试集中的样本如果比最长的训练样本还长，则会在 SpamDataset 代码中通过 encoded_text[:self.max_length] 进行截断。
- 此行为完全是可选的，即使在验证集和测试集中将 max_length 设置为 None 也可以正常工作。

6.4 使用预训练权重初始化模型

- 在本节中，我们初始化了上一章使用的预训练模型。

6.5 Adding a classification head



- 目标是替换并微调输出层。
- 为了实现这一点，我们首先冻结模型，即将所有层设置为不可训练状态。
- 然后，我们替换输出层（model.out_head），它原本将层的输入映射到50,257个维度（即词汇表的大小）。
- 由于我们微调模型进行二分类（预测2个类别，“垃圾邮件”和“非垃圾邮件”），我们可以按如下方式替换输出层，该层默认是可训练的。
- 注意，我们使用 BASE_CONFIG["emb_dim"]（在"gpt2-small (124M)"模型中等于768）以使下面的代码更具通用性。
- 技术上来说，只训练输出层就足够了。
- 然而，正如我在《微调大型语言模型》中发现的那样，实验表明微调额外的层可以显著提高性能。
- 因此，我们还使最后一个transformer块和连接最后一个transformer块到输出层的最终LayerNorm模块变为可训练的。

- 在第3章中，我们讨论了注意力机制，该机制将每个输入token与其他输入token连接起来。
- 在第3章中，我们还介绍了GPT类模型中使用的因果注意力掩码causal attention mask ；这种因果掩码使得当前token只能关注当前及之前的token位置。

- 基于这种因果注意力机制，第4个（最后一个）token包含了所有token中最多的信息，因为它是唯一包含所有其他token信息的token。
- 因此，我们对这个最后一个token特别感兴趣，我们将微调它用于垃圾邮件分类任务。

6.6 Calculating the classification loss and accuracy

- 以看到，预测准确率不是很高，因为我们尚未微调模型。
- 在我们开始微调（训练）之前，我们首先必须定义在训练过程中希望优化的损失函数。
- 目标是最大化模型的垃圾邮件分类准确率；然而，分类准确率不是一个可微分的函数。
- 因此，我们通过最小化交叉熵损失cross-entropy loss来代替最大化分类准确率（可以在我免费提供的深度学习导论课程的第 8 讲中了解更多有关此主题的内容）。
- 这里的 `calc_loss_batch` 函数与第 5 章中的相同，只是我们只关注优化最后一个 `token model(input_batch)[:, -1, :]` 而不是所有 `tokens model(input_batch)`
- 在下一节中，我们将训练模型以改进损失值，从而提高分类准确率。

6.7 Finetuning the model on supervised data

- 在本节中，我们定义并使用训练函数来提高模型的分类准确率。
- 下面的 `train_classifier_simple` 函数实际上与我们在第 5 章中用于预训练模型的 `train_model_simple` 函数几乎相同。
- 唯一的两个不同之处在于：
  - 我们现在跟踪看到的训练样本数（`examples_seen`），而不是看到的 tokens 数量。
  - 每个 epoch 结束后计算准确率，而不是每个 epoch 结束后打印示例文本。

- 上面，根据下降的曲线斜率，可以看出模型学习效果良好。
- 此外，训练损失和验证损失非常接近，这表明模型没有明显地过拟合训练数据。
- 同样地，我们可以在下方绘制准确率图。

- 从上方的准确率图可以看到，模型在第4和第5个周期后达到了相对较高的训练和验证准确率。
- 不过需要注意的是，我们在之前的训练函数中指定了eval_iter=5，这意味着我们只是估算了训练和验证集的性能。
- 我们可以如下计算整个数据集上的训练、验证和测试集性能

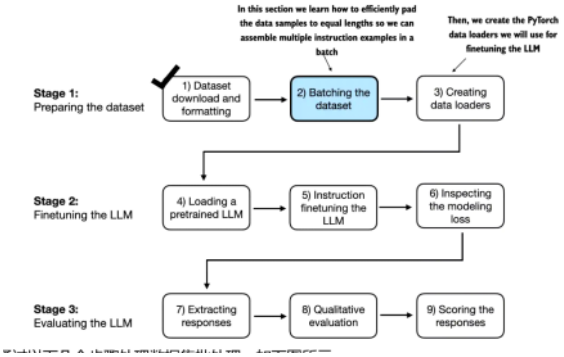
- 可以看到，训练集和验证集的性能几乎是相同的。
- 然而，根据略低的测试集性能，可以看出模型在一定程度上过拟合了训练数据，以及用于调整一些超参数（例如学习率）的验证数据。
- 这是正常现象，可以通过增加模型的dropout率（drop_rate）或优化器设置中的weight_decay来进一步减小这种差距。

6.8 Using the LLM as a spam classifier

Chapter 7: Finetuning To Follow Instructions

2024年11月18日 22:02

7.1 指令微调简介



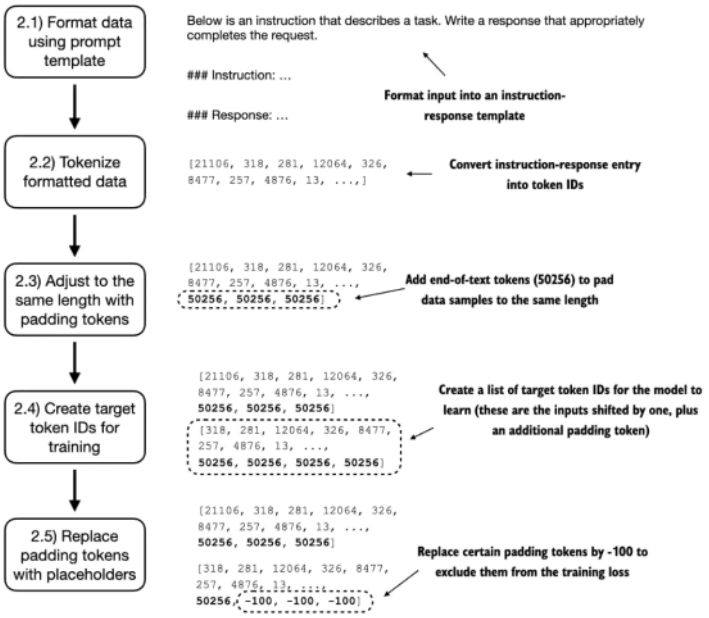
- 在第5章中，我们看到，对LLM（大语言模型）的预训练涉及一种逐字生成的训练过程。
- 因此，预训练的LLM擅长文本补全，但并不擅长遵循指令。

7.2 准备用于监督指令微调的数据集

- 我们将使用为本章准备的一个指令数据集。

- 指令微调通常被称为“监督指令微调”supervised instruction finetuning，因为它涉及对一个显式提供输入-输出对的数据集进行训练。
- 将这些条目格式化为LLM的输入有多种方式；
- 这章我们用Alpaca-style prompt formatting，它是original prompt template for instruction finetuning
- 在下一节准备PyTorch数据加载器之前，我们将数据集分为训练集、验证集和测试集。

7.3 将数据组织为训练批次



- 与第6章类似，我们希望在一个批次中收集多个训练样本以加速训练；这需要将所有输入填充到相似的长度。
- 同样类似于上一章，我们使用<|endoftext|>作为填充标记。

在第6章中，我们将数据集中所有样本填充到相同长度。

- 在这里，我们采用了一种更复杂的方法，开发了一个自定义的“collate”函数，可以传递给数据加载器。
- 这个自定义collate函数将每个批次中的训练样本填充到相同长度（但不同批次可以具有不同长度）。
- 在上面，我们只返回了LLM的输入；然而，对于LLM的训练，我们还需要target values
- 类似于LLM的预训练，目标是将输入向右移动1个位置，这样LLM就可以学习预测下一个标记。

我们引入一个ignore_index值，用于将所有填充标记ID替换为一个新值；ignore_index的目的是在损失函数中忽略填充值

这意味着我们将对应于50256的标记ID替换为-100，但是第一个不会换，如下所示：

- 可以看到，这3个训练样本的最终损失与我们从2个训练样本计算的损失相同，这意味着交叉熵损失函数忽略了带有标签-100的训练样本。
- 默认情况下，PyTorch的cross_entropy(..., ignore_index=-100)设置会忽略标签为-100的样本。
- 使用这个-100的ignore_index，我们可以忽略用于填充训练样本以达到相同长度的额外文本结束标记（padding token）。
- 然而，我们不希望忽略第一个文本结束标记（50256）的实例，因为它可以帮助向LLM指示响应何时完成。
- 在实践中，通常也会屏蔽与指令对应的目标标记ID

7.4 为instruction dataset创建data loaders

- 在本节中，我们将使用 InstructionDataset 类和 custom_collate_fn 函数来实例化训练集、验证集和测试集的数据加载器。

- 上述 custom_collate_fn 函数的另一个改进是：我们现在直接将数据移动到目标设备（例如 GPU），而不是在主训练循环中完成这一步操作。
- 这样可以提高效率，因为在将 custom_collate_fn 作为数据加载器的一部分时，这一步可以作为后台进程执行。
- 使用 Python 的 functools 标准库中的 partial 函数，我们创建了一个新的函数，将原函数的 device 参数预先填充。

7.5 加载预训练 LLM

- 从结果可以看出，模型还不具备遵循指令的能力；它虽然生成了一个“Response”部分，但只是简单地重复了原始输入句子以及指令。

7.6 在指令数据上微调 LLM

- 从上述输出可以看出，模型训练效果良好，从减少的训练损失和验证损失值可以看出来。
- 此外，从每个 epoch 结束后打印的响应文本可以看出，模型能够正确遵循指令，例如将输入句子“厨师每天做饭”（The chef cooks the meal every day.）转换为被动语态“这顿饭每天由厨师烹饪”（The meal is cooked every day by the chef.）。（我们将在后面的章节中正式格式化并评估这些响应。）
- 最后，让我们看一下训练和验证损失曲线。

7.7 Extracting and saving responses

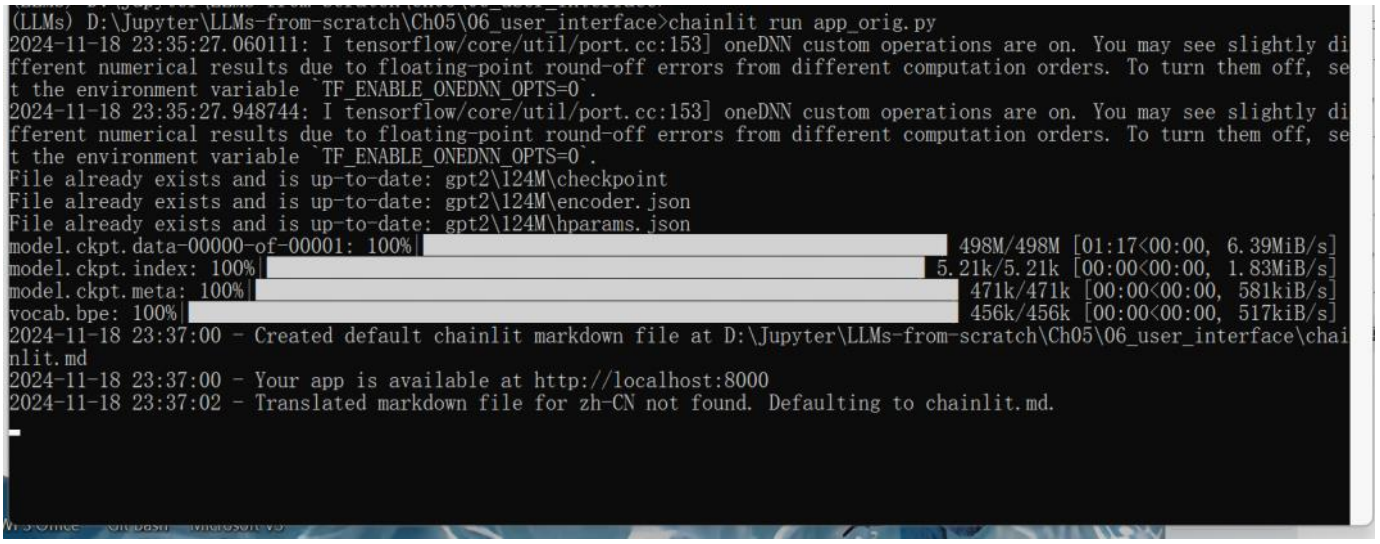
- 在本节中，我们将保存测试集的响应，以便在下一节进行评分，同时保存一份模型的副本以供将来使用。
- 正如我们从测试集的指令、给定的响应以及模型的响应中可以看到的，模型的表现相对较好。
- 对于第一条和最后一条指令，模型的回答显然是正确的。
- 第二个答案也很接近；模型给出的答案是“积云”而不是“雷雨云”（不过需要注意的是，积云可以发展成雷雨云，而雷雨云能够产生雷暴）。
- 最重要的是，我们可以看到，模型的评估并不像上一章那样简单，只需计算正确的垃圾邮件/非垃圾邮件标签的百分比来获得分类准确率。
- 在实际应用中，指令微调的LLM（如聊天机器人）通常通过多种方法进行评估：
  - 短答案和多选基准，如MMLU（“衡量大规模多任务语言理解”，测试模型的知识（ https://arxiv.org/abs/2009.03300 ）；
  - 与其他LLM的人工偏好比较，如LMSYS聊天机器人竞技场（https://arena.lmsys.org ）；
  - 自动化对话基准，其中使用另一个LLM（如GPT-4）来评估响应，如AlpacaEval（ https://tatsu-lab.github.io/alpaca_eval/ ）。
- 在下一节中，我们将采用类似于AlpacaEval的方法，使用另一个LLM来评估我们的模型响应；不过，我们将使用自己的测试集，而不是公开可用的基准数据集。
- 为此，我们将模型响应添加到test_data字典中，并将其保存为“instruction-data-with-response.json”文件进行记录，以便在需要时加载并在单独的Python会话中进行分析。

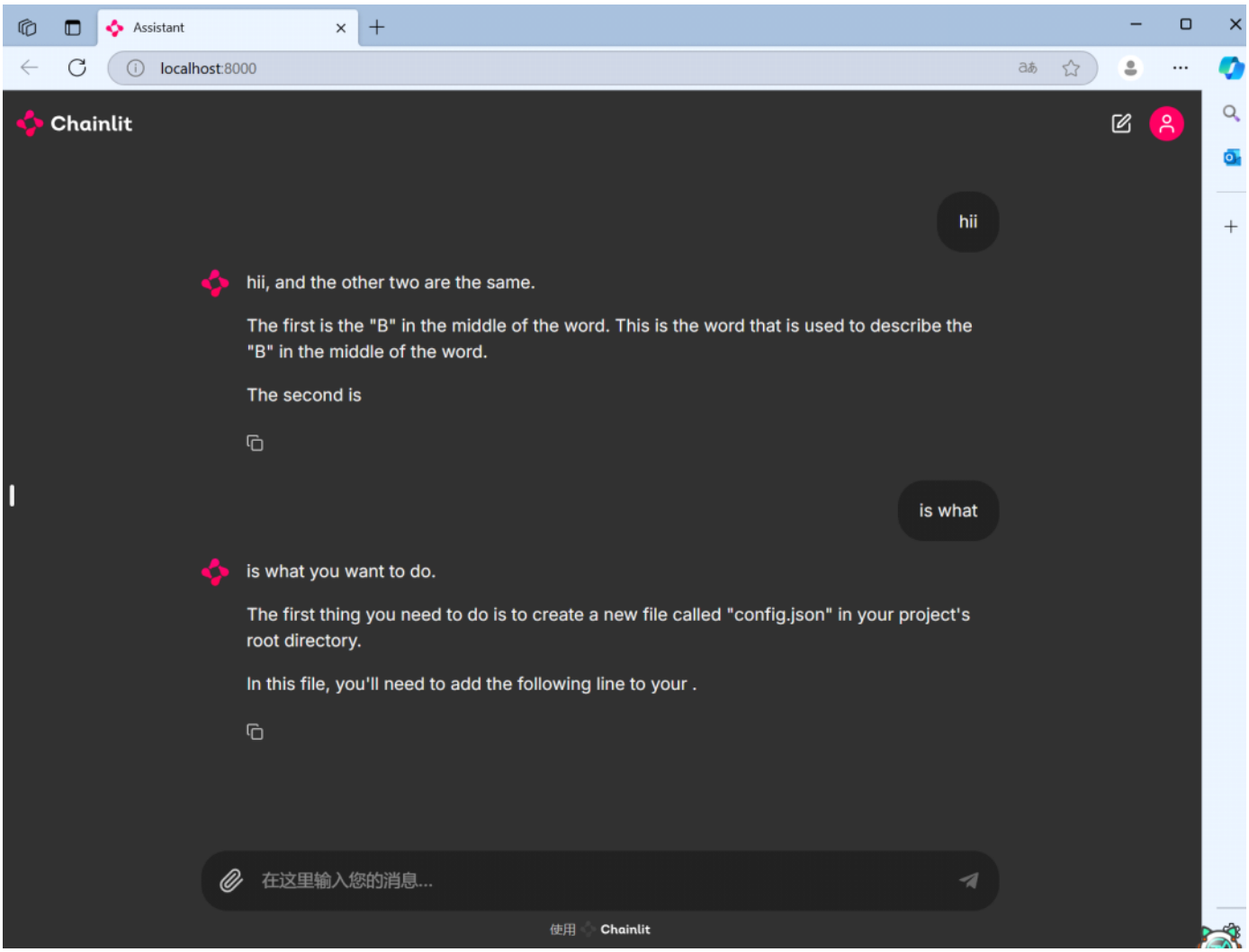
7.8 评估微调的LLM

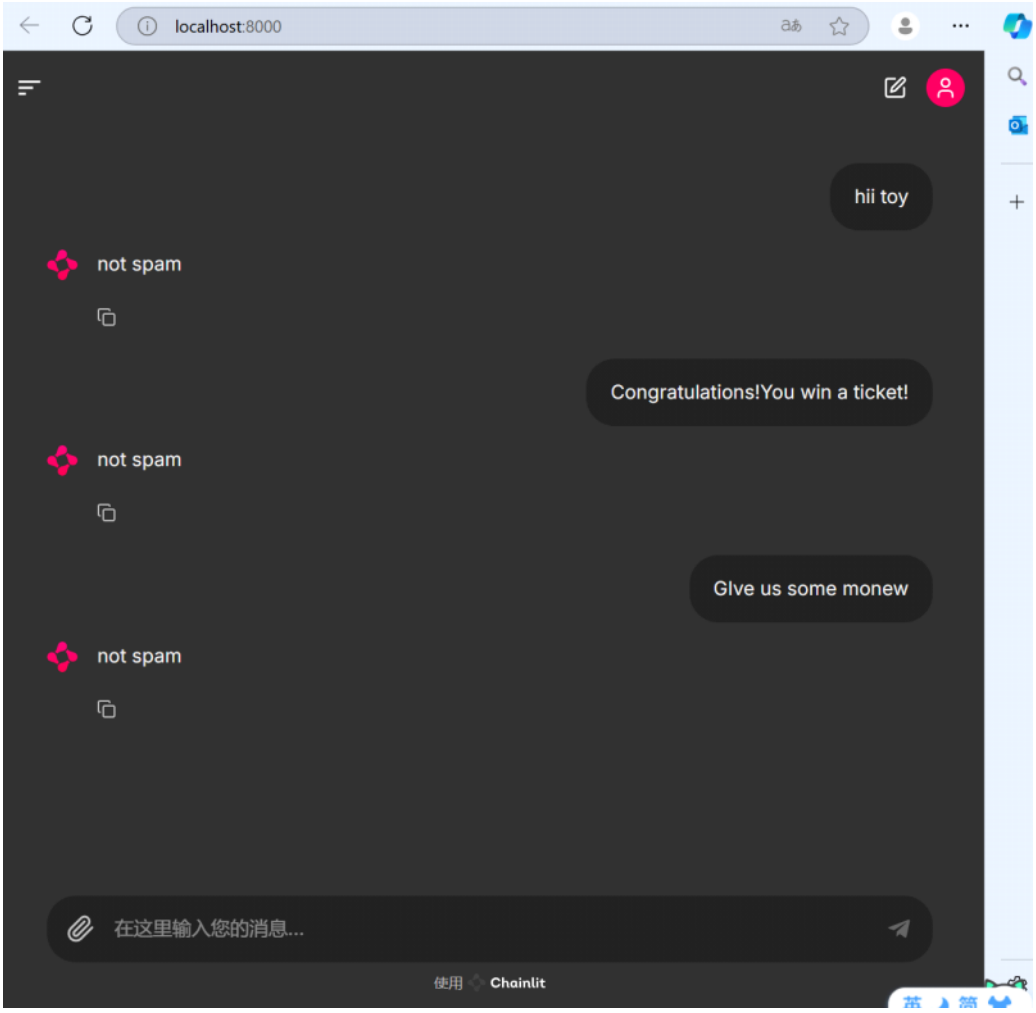
- 在本节中，我们使用另一个更大的LLM自动化评估微调后的LLM响应。
- 特别地，我们使用Meta AI的一个微调指令的8亿参数Llama 3模型，它可以通过ollama（https://ollama.com ）本地运行。（如果您更喜欢使用像GPT-4这样的更强大的LLM通过OpenAI API，请参见llm-instruction-eval-openai.ipynb笔记本）。
- Ollama是一个高效运行LLM的应用程序，它是一个封装在llama.cpp（ https://github.com/ggerganov/llama.cpp ）中的工具，后者实现了用纯C/C++编写的LLM，以最大化效率。
- 需要注意的是，它是一个用于生成文本（推理）的LLM工具，而不是用于训练或微调LLM。
- 在运行下面的代码之前，请访问https://ollama.com 并按照说明安装ollama（例如，点击“下载”按钮并下载适合您的操作系统的ollama应用程序）。

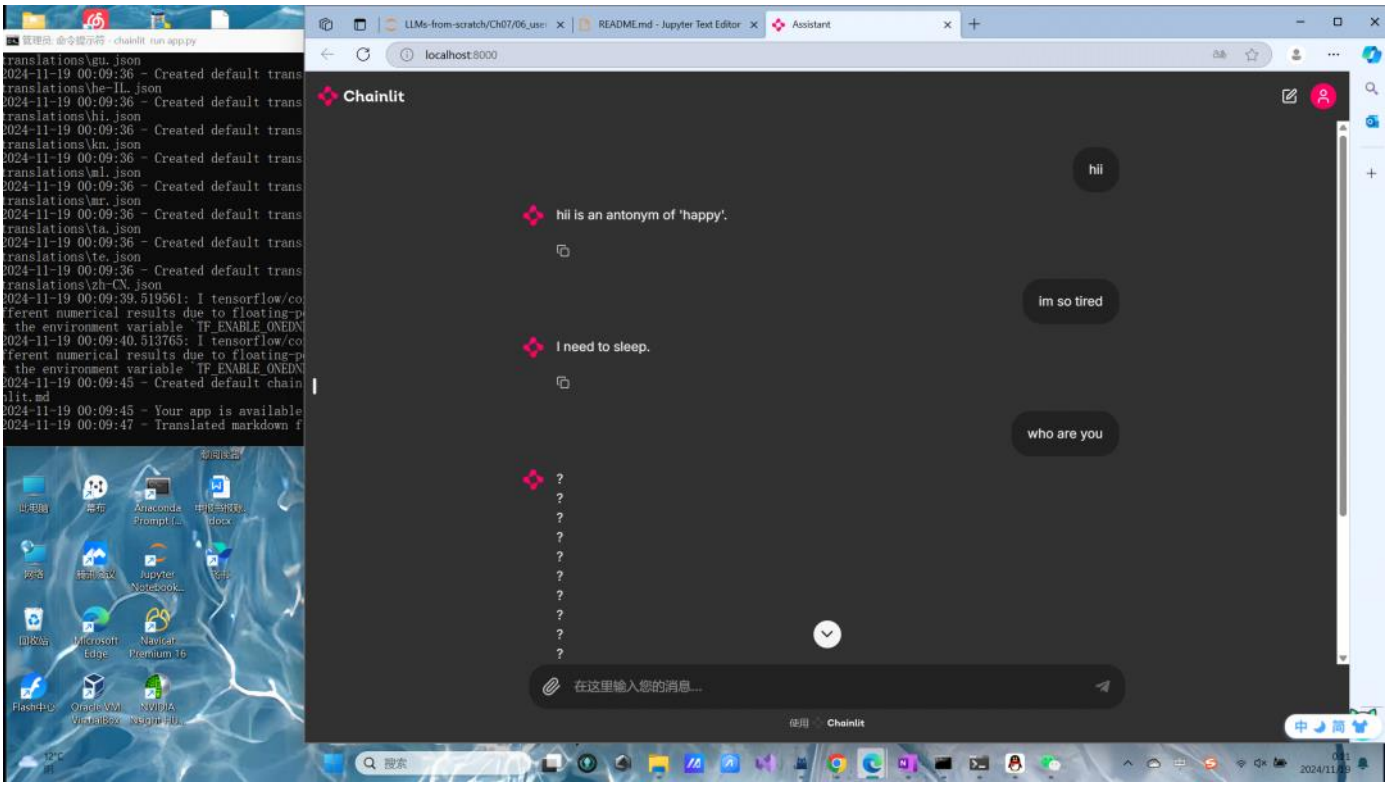
UI server运行截图

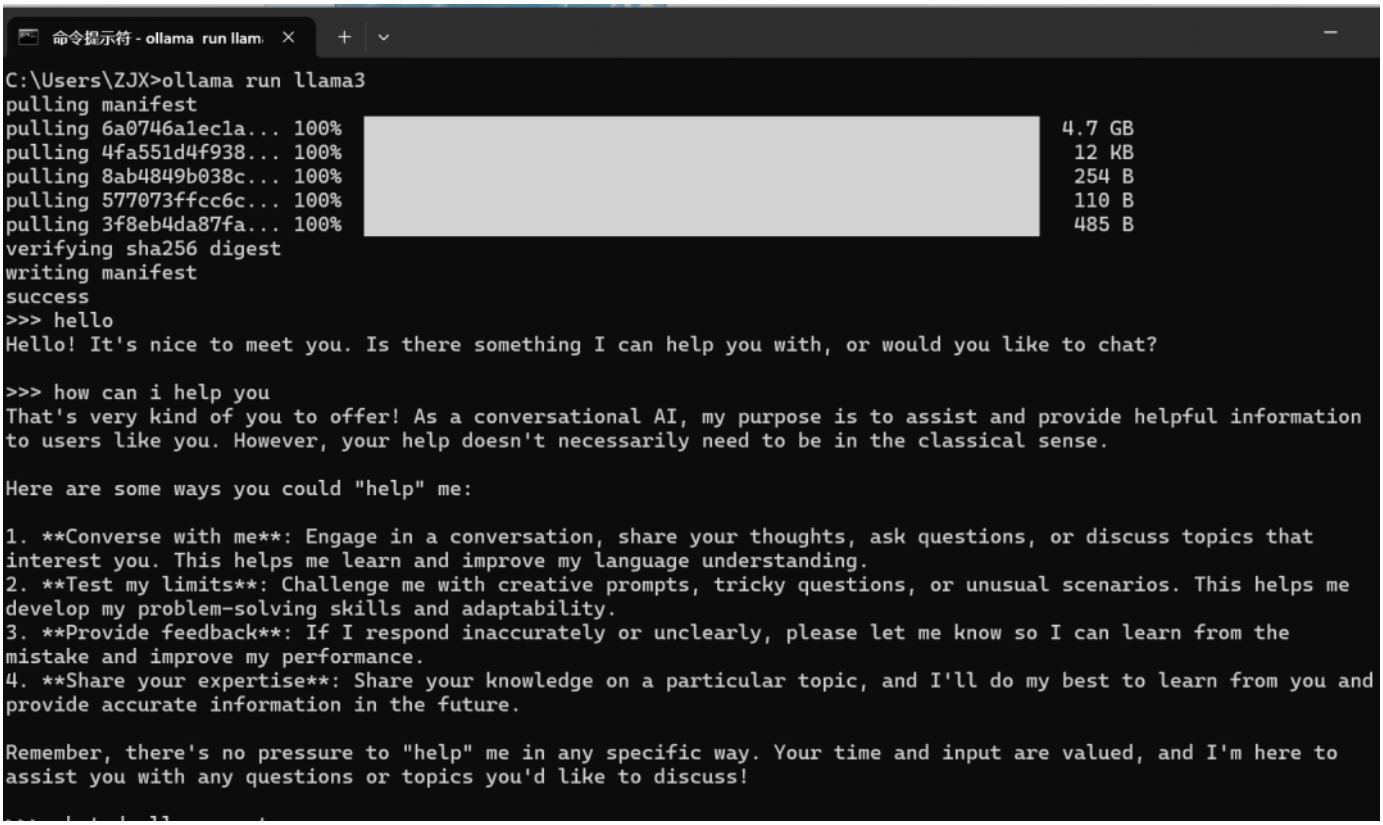
2024年11月18日 23:39






localhost:8000
hii toy
not spam
Congratulations!You win a ticket!
not spam
Give us some monew
not spam
在这里输入您的消息...
使用 Chainlit


localhost:8000
Chainlit
hii
hii is an antonym of 'happy'.
im so tired
I need to sleep.
who are you
在这里输入您的消息...
使用 Chainlit



Ollama run llama3

knowledge

2024年11月18日 19:23

cfg 是一个包含模型配置的字典（dictionary）。它通常用于存储模型的超参数和相关设置，使代码可以根据不同配置进行调整。这些配置参数定义了模型的关键特性，例如词汇量大小、嵌入维度、上下文长度、dropout 率等。
以下是 cfg 中可能包含的键及其对应的配置说明：

- vocab_size：词汇表的大小，定义模型可以使用的词汇数量。
- emb_dim：嵌入维度，即每个输入 token 被映射成的向量维度大小。
- context_length：上下文长度，即模型的输入序列中最大 token 数量。
- drop_rate：dropout 率，表示在训练时丢弃的单元比例，用于防止过拟合。
- n_layers：模型中 Transformer 块的数量。

这些超参数使模型能够根据不同的任务需求或硬件资源灵活调整，而无需修改模型的核心代码。

- layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())：
  - 定义一个简单的神经网络层序列 layer。
  - 使用 nn.Sequential，可以将多个层按照顺序组合成一个模块。这里包括两个层：
    - nn.Linear(5, 6)：一个线性（全连接）层，将输入从 5 维映射到 6 维。nn.Linear 接受输入大小和输出大小两个参数，并在此基础上生成权重矩阵和偏置向量。
    - nn.ReLU()：一个激活函数层，应用 ReLU（修正线性单元）激活函数，将所有负值截断为 0，保留正值不变。这是一种常见的激活函数，有助于神经网络学习非线性关系。
- out = layer(batch_example)：
  - 将 batch_example 输入到定义的 layer 层序列中。
  - 数据会先经过线性层 nn.Linear(5, 6)，得到一个 6 维的输出，然后再通过 ReLU 激活层 nn.ReLU()。最终结果存储在 out 中。

梯度在深度学习和机器学习中是用来指导模型优化的。梯度表示损失函数对模型参数的变化率，即当模型的某个参数变化时，损失函数如何随之变化。具体来说，梯度的计算和作用如下：

1. **计算损失的变化方向**：梯度指出了损失函数增大或减小的方向。在训练模型时，梯度下降算法会使用这个信息，沿着梯度相反的方向更新参数，以逐步减少损失函数的值。
2. **更新参数**：在每次训练迭代中，计算出的梯度用于调整模型的参数。通过不断更新这些参数，模型逐渐学习到数据中的模式，从而提高预测的准确性。通常，这些更新是通过反向传播算法实现的，即先计算输出与目标的误差，再从输出层逐层向前传播该误差并计算各层参数的梯度。
3. **控制学习速率**：梯度的大小会影响参数更新的幅度。通常，为了避免过大的参数更新引发模型不稳定的问题，会使用“学习率”来调节更新步长。梯度大时，学习率可帮助减小更新量，而梯度小的情况下，学习率确保更新量不至于太小。

总之，梯度在模型训练中起到指导方向和调整步长的作用，帮助模型更好地拟合数据，提高模型的预测能力。

- logits每个单词都有
- logits 是模型生成的输出张量，其中包含了每个 token 在词汇表中每个单词的分数（未经过 softmax 转换的概率）。该张量的形状通常为 (batch_size, num_tokens, vocab_size)。
- targets 是目标张量，包含了模型期望生成的每个 token 的真实词汇索引，其形状通常为 (batch_size, num_tokens)。
- cross_entropy 函数期望输入为 (N, C) 的 logits 张量（其中 N 是样本总数，C 是类别数）和形状为 (N,) 的目标张量 targets。在这里，通过展平操作，我们将 batch_size * num_tokens 作为总样本数，将 vocab_size 作为类别数，以便与 cross_entropy 函数的输入要求一致。

logits 是神经网络输出的一个张量，它包含了模型对每个输入位置（例如每个 token 或单词）在所有可能类别中的得分。对于生成式语言模型（如 GPT 系列），logits 的形状通常为 (batch_size, num_tokens, vocab_size)，其中：

- batch_size：批次大小，即每次输入的句子数量。
- num_tokens：每个输入句子的 token 数。
- vocab_size：词汇表大小，表示模型可以输出的所有可能词的数量。

logits 的作用

1. **生成概率分布**：logits 是在没有归一化的状态下表示的分数（或称为“未标准化的对数概率”），这些分数还没有经过 softmax 函数的处理。通过对 logits 应用 softmax，我们可以得到每个位置上所有词的概率分布，即模型对每个词的预测概率。
2. **选择最可能的输出**：在生成文本时，模型会在 logits 中找到概率最高的词，并将其作为输出 token 的预测结果。可以使用贪婪解码（选择最高概率的词）或其他解码方法（如 beam search）来生成一系列连续的词。
3. **计算损失**：在训练过程中，logits 与真实标签（目标 token）之间的交叉熵损失（cross-entropy loss）用于指导模型优化权重，使得模型生成的词更接近目标文本。例如，假设目标 token 是“apple”，而 logits 经过 softmax 后对“apple”词的预测概率越高，损失就越小，模型就学习得更好。

calc_loss_loader 和 Cross-Entropy Loss 的关系

- calc_loss_loader 中的 loss 是通过另一个函数 calc_loss_batch 计算的，通常情况下 calc_loss_batch 内部会调用 cross-entropy loss 来计算单个批次的损失。
- Cross-Entropy Loss：用于分类任务中，衡量模型的预测概率分布与真实标签分布之间的差异。其目标是使模型预测接近真实标签的概率最大化。
- 在 calc_loss_loader 中，cross-entropy loss 作用于每个批次，然后所有批次的 cross-entropy loss 平均值就是这个函数返回的结果，因此该函数本质上是计算整个数据集的平均 cross-entropy loss。

总结

这个 calc_loss_loader 函数在整体上并没有改变 cross-entropy loss 的本质，而是通过累加和平均实现了在数据集（或部分数据集）上的平均交叉熵损失。

temperature scaling

logits 除以一个大于 0 的数值temperature

点积简介

点积是一种在向量间进行计算的操作，本质上是“逐元素相乘并求和”。在NLP和深度学习中，点积通常用于计算两个向量的相似度。数值越大，通常表示向量的相似度越高（在特征空间中更接近）。

假设有两个向量$a=[a1,a2,\cdots,an]$ 和$b=[b1,b2,\cdots,bn]$，它们的点积定义为：

$$a\cdot b=a1\cdot b1+a2\cdot b2+\cdots+an\cdot bn$$

代码实现

在上面的代码中，点积的计算方法有两种，一种是手动计算，另一种是用 torch.dot 函数。

1. **手动实现点积**：

```
res = 0
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
```

   - res 初始化为 0，用于累加点积的结果。
   - for idx, element in enumerate(inputs[0]) 遍历 inputs[0] 向量的每个元素。
   - 每次循环中，将 inputs[0][idx] 和 query[idx] 相乘，并将结果累加到 res 中。

2. **使用 torch.dot 计算点积**:
   print(torch.dot(inputs[0], query))
   - torch.dot(inputs[0], query) 是 PyTorch 中计算两个向量点积的便捷方法。这个方法可以直接计算出两个相同长度向量的点积。

输出结果

两个方法的输出结果应该相同，因为它们计算的是同一个点积。这里展示了两种方法——既可以理解点积的工作原理，也可以看到如何使用 PyTorch 的内置方法高效计算点积。

自注意力层的偏置项在深度学习的自注意力机制（尤其是在Transformer模型中）中起到重要作用。在自注意力层中，查询（query）、键（key）和值（value）矩阵通过线性变换从输入生成，这些线性变换通常包括一个权重矩阵和一个偏置项。偏置项在计算查询、键和值时加到权重矩阵的输出上，以便引入一些基础偏移，使得模型的输出不仅依赖于输入，还包括了固定的偏移量。

在某些模型（例如GPT等大型语言模型）中，会选择去除偏置项，以简化模型参数和减少计算成本。移除偏置项不会显著影响性能，因为在大规模训练和数据量较大的情况下，模型往往可以通过其他方式捕获必要的模式。但是，在一些特定情况下，比如小模型或者特定任务中，偏置项可以帮助模型更好地收敛并捕获细节特征。

总结来说，偏置项的作用是引入基础偏移量，使模型能够更灵活地调整输出。在代码中使用偏置项与否，往往是根据模型的规模、计算效率和特定任务需求来决定的。

1. Batch

batch（批次）是指在一次前向传递和反向传播过程中传递到神经网络的样本集合。在深度学习训练过程中，将整个数据集分成较小的批次处理，而不是一次性将所有数据输入网络，这样不仅能更好地管理内存，还能更稳定地更新权重。

- Batch Size：批次的大小，即每次训练中输入到模型的样本数量。比如，batch_size=32 表示每次将32个样本同时输入模型。
- Mini-batch：如果批次大小较小，可以称为 mini-batch。小的批次能更快地迭代，但计算不稳定，大的批次稳定但计算开销较大。

2. Benchmark

benchmark（基准测试）是指用于测量和评估模型性能的一种标准测试。通常，通过对不同的模型在相同的数据集上进行评估，比较它们的速度、准确度、效率等指标来进行基准测试。这可以帮助开发人员和研究人员了解模型在各种任务中的表现。

- 在深度学习中，benchmark 数据集（例如 ImageNet、COCO、GLUE 等）是用于测试模型准确度的标准数据集。
- Benchmarking **训练时间**：通过测试模型在相同硬件条件下的训练时间，帮助选择适合的模型或硬件。

3. Epoch

epoch（轮次）指对整个数据集进行一次完整的训练过程。也就是说，模型在一次 epoch 中遍历了数据集中的所有样本。在深度学习训练中，模型通常要经过多次 epoch，不断更新模型权重，以最小化损失函数。

- **训练多个 Epoch**：通常训练多个 epoch 是为了使模型更好地拟合数据；每次 epoch 都会对数据集进行一次完整遍历，从而提升模型的学习效果。
- **早停策略**：如果验证集的准确率不再提升，可以通过早停（early stopping）来避免过拟合和浪费计算资源。

监督学习（Supervised Learning）是一种机器学习方法，在这种方法中，算法使用带标签的数据进行训练。带标签的数据包含输入和对应的正确输出，算法通过学习输入和输出之间的关系来进行预测。常见的监督学习任务包括**分类**和**回归**。

监督学习的基本流程

1. **数据准备**：收集带标签的数据集，即包含特征（输入）和目标值（输出）配对的数据。比如，在垃圾邮件分类中，输入可以是邮件内容的特征，输出是“垃圾”或“非垃圾”的标签。
2. **训练模型**：将带标签的数据集输入算法，让模型学习输入和输出之间的关系。模型在训练过程中逐渐调整自身参数以最小化预测误差。
3. **模型评估**：使用一部分数据（通常称为测试集）来验证模型的性能，评估它在未见数据上的预测效果。

4. **预测**：使用训练好的模型对新数据进行预测，根据输入生成对应的输出。

监督学习的两种主要任务

- **分类任务**：用于处理离散的类别标签，例如邮件是否为垃圾邮件、图片是否包含某种物体等。
- **回归任务**：用于处理连续的数值输出，例如预测房价、温度等。

监督学习的应用

监督学习被广泛应用于自然语言处理、图像识别、医疗诊断等领域。例如：

- **图像分类**：将图片分类为不同的标签（如猫或狗）。
- **语音识别**：将语音输入转换为文本。
- **金融预测**：预测股票价格或市场趋势。

总之，监督学习适用于已知的输入-输出映射情况，通过大量的标注数据训练出高精度的模型，并能在未来数据中进行有效预测。

在Scikit-learn中，Logistic Regression（逻辑回归）是一种常用的机器学习算法，主要用于分类任务。尽管其名字中带有“回归”一词，但逻辑回归实际上是一种分类模型。Scikit-learn中的LogisticRegression类使用逻辑回归实现二分类和多分类问题。

工作原理

逻辑回归的核心思想是通过一个逻辑函数（Sigmoid函数或Softmax函数）将输入的线性组合结果映射到一个概率值上，用于表示某个样本属于特定类别的概率。例如，在二分类中，逻辑回归将一个输入特征的线性组合传递给Sigmoid函数，将输出值限制在0到1之间。然后，通过一个阈值（通常为0.5）进行分类判断。

Scikit-learn中的实现

在Scikit-learn中，逻辑回归可以通过以下步骤实现：

1. **导入模型**：使用`from sklearn.linear_model import LogisticRegression`。
2. **创建模型**：通过`model = LogisticRegression()`来创建逻辑回归模型实例，可以指定参数如C（正则化强度）和solver（优化算法）。
3. **训练模型**：使用`model.fit(X_train, y_train)`来训练模型，`X_train`为特征数据，`y_train`为标签数据。
4. **预测**：使用`model.predict(X_test)`对测试数据进行预测，或用`model.predict_proba(X_test)`获取预测的概率。
5. **评估模型**：使用准确率、精度、召回率等指标来评估模型效果。

使用场景

Scikit-learn的逻辑回归广泛应用于二分类问题，例如垃圾邮件检测（垃圾邮件或非垃圾邮件）和信用风险评估（违约或不违约）。它还可以扩展到多分类任务（使用“多项逻辑回归”），例如手写数字识别（0到9的分类）。

优势和局限性

- **优势**：逻辑回归模型简单、易于实现，适合处理线性可分数据，并且对于小数据集和低计算成本的任务非常高效。
- **局限性**：逻辑回归不适合复杂的非线性数据；在数据量大或特征高度相关时，它可能表现较差，需引入正则化以改善效果。

Scikit-learn中的LogisticRegression模型是一个灵活且强大的工具，非常适合快速构建和测试分类模型。

pandas是一个用于数据分析和数据处理的Python库。它提供了高效的数据结构和数据操作工具，适合处理各种形式的结构化数据，例如表格数据、时间序列等。pandas的核心数据结构有两个：

1. `Series`：一维数组，类似于Python中的列表，但可以包含索引。适合存储一列数据，比如Excel中的单列。
2. `DataFrame`：二维数据结构，类似于电子表格或SQL表。每一列可以是不同的数据类型，适合用于存储和操作整个表格数据。

pandas的常用功能

- **数据读取与存储**：支持从CSV、Excel、SQL数据库、JSON等多种数据源读取数据，使用`pd.read_csv()`、`pd.read_excel()`等

函数。同时，可以将数据保存到这些格式中。

- **数据清洗**：可以对缺失数据进行处理（填充、删除等），重命名列名，去重，以及数据类型转换等操作。
- **数据筛选与过滤**：支持按条件筛选数据，基于标签或位置选择行列，比如使用loc和iloc方法。
- **数据聚合与分组**：pandas提供了强大的数据分组与聚合功能，例如groupby()可以根据某一列或多列的值进行分组，并应用聚合函数（如求和、平均等）。
- **数据变换**：支持对数据进行各种变换，例如数据透视表、合并与连接等操作，使用pivot_table()、merge()等方法。
- **时间序列处理**：内置了对时间序列数据的支持，包括日期范围生成、时间戳转换、数据重采样等。

baseline（基线）**指的是一种简单但有效的模型或方法，用来衡量和比较其他模型或方法的性能。基线模型可以是一个基本的预测算法，也可以是一个简单的规则，其主要目的是作为性能的参考点，帮助判断更复杂的模型是否真的带来改进。

Baseline**的类型**

1. **简单算法基线**：在分类问题中，常见的基线算法包括使用逻辑回归、决策树或K最近邻。在文本分类中，可能会使用朴素贝叶斯或基于词频的模型。
2. **常数基线**：在回归问题中，基线可以是简单的平均值预测。例如，对所有输入预测相同的均值，这种方法可以作为回归问题的一个基本参考。
3. **随机基线**：在分类问题中，随机基线会随机分配一个类别。尽管这种基线通常不会表现很好，但在某些不平衡数据集中，仍可以为模型提供最基础的性能参考。
4. **上一模型基线**：在模型改进和迭代过程中，之前版本的模型也可以作为基线，帮助评估新模型是否有实际的性能提升。

Baseline**的作用**

- **评估模型的改进**：通过与基线的性能对比，可以判断新的复杂模型是否带来了真正的性能提升。
- **调试和验证模型**：如果复杂模型表现不如基线，可能意味着模型的设计存在问题，或者数据集需要进一步清洗或处理。
- **设定性能预期**：在新项目中，baseline可以帮助确定目标，避免花费大量资源构建复杂模型却得不到显著提升。