HW4-P1

2024年8月22日 12:09

S24版助教作业辅导说明：https://www.youtube.com/watch?v=Nq3HUS-m574

- PPT内容跟S23版基本一样。

文件结构：

```
HW4P1
├─ dataset                                语言模型的【预测】和【生成】任务训练集
│  ├─ vocab.csv
│  ├─ vocab.npy
│  └─ wiki.train.npy
├─ fixtures                               语言模型的【预测】和【生成】任务验证集
│  ├─ generation.npy
│  ├─ generation_test.npy
│  ├─ prediction.npz
│  └─ prediction_test.npz
├─ Transformer                            transformer模型在文本翻译和图像分类中的2种应用
│  ├─ en_to_ch_data                       英中翻译数据集
│  │  ├─ test.txt
│  │  ├─ train.txt
│  │  └─ val.txt
│  ├─ en_to_de_data                       英德翻译数据集
│  │  ├─ test
│  │  │  ├─ test.de
│  │  │  └─ test.en
│  │  ├─ train
│  │  │  ├─ train.de
│  │  │  └─ train.en
│  │  └─ val
│  │     ├─ val.de
│  │     └─ val.en
│  ├─ Lab 11_Transformers_Vision.ipynb    官方VIT图像分类代码
│  ├─ Lab_11_Transformers_Language.ipynb  官方英德翻译代码
│  ├─ Transformer_en_to_de1.ipynb
│  ├─ Transformer_en_to_de2.ipynb
│  └─ Transformer_en_to_zh.ipynb          我的英中翻译代码
├─ attention.py
├─ hw4p1.ipynb
├─ hw4p1_autograder.py
├─ HW4P1_Writeup.pdf
├─ hw4p1_ym.ipynb                         我的HW4P1代码
├─ tests_hw4.py
└─ 文件说明.md
```


一 概述

作业目标：

具体任务：

二 Masked Self-Attention

注意力机制概述

1. 注意力机制的核心概念

2. 注意力机制的不同类型

2.1 自注意力 (Self-attention)

2.2 掩码自注意力 (Masked Self-attention)

2.3 交叉注意力 (Cross-attention)

2.4 多头注意力 (Multi-head attention)

Masked Self-Attention（加性因果掩码自注意力）

1. 关键概念解释

2. 加性因果掩码 (Additive Causal Mask)

前向传播

1. 注意力机制中的主要组件（Attention Components）

2. 前向传播方程（Forward Equations）

反向传播

1 关于注意力权重（原始和归一化）的梯度

2 关于键、查询和值的梯度

3 关于权重矩阵的梯度

4 关于输入的梯度

代码

三 语言模型的【预测】和【生成】任务

1 数据集形态

2 训练集和标签制作

3 模型结构

4 训练原理

5 【预测】的val、test集验证原理

负对数似然函数逐行解释

1. log_softmax(out, 1)

2. nlls = out[np.arange(out.shape[0]), targ]

3. nll = -np.mean(nlls)

4. 返回 nll

计算 NLL 的原理

总结

6 【生成】的val、test集验证原理

精度的评估方式：

1. 序列概率计算

2. 对数似然

3. 计算平均对数似然

4. 计算困惑度

5. GPT 如何计算困惑度

总结

7 我的训练结果 及 作业精度要求

8 讲义中提到的精度提升建议

1. Locked Dropout（锁定的Dropout）

2. Embedding Dropout

3. Weight Tying（权重绑定）

4. Activation Regularization（激活正则化，AR）

5. Temporal Activation Regularization（时间激活正则化，TAR）

四 Transform 英中翻译任务+VIT图像分类任务

一 概述

作业目标：

- 学习如何从头实现 **Masked Self-Attention。**
- 学习如何训练生成文本的 **循环神经网络** 模型。
- 学习各种正则化循环神经网络的技巧，如 **Locked Dropout、Embedding Dropout、权重衰减、权重共享、激活正则化。**
- 你将理解 **语言建模** 的工作原理，并且还将训练一个能够生成下一个单词以及整个序列的模型！

具体任务：

1. **实现 Masked Self-Attention 类**：你需要在 `attention.py` 文件中完成一个 `Attention` 类的前向传播和反向传播函数的实现。这部分作业是为了让你深入理解注意力机制的工作原理。
   a. 测试方法：python hw4/hw4p1_autograder.py
2. **语言模型的【预测】任务**：在Jupyter Notebook中，你需要完成 `LanguageModel` 类中的 `predict` 函数。这个函数接受一个批次的序列，利用训练好的模型进行前向传播，并返回每个序列的下一个可能的单词的预测分数。
   a. 预测任务只关注生成下一个单词的概率分布，即给定部分文本后，模型“猜测”下一个单词是什么。
   b. **举个例子：**

   假设你输入一个句子片段 `"The cat is on the"`，模型的任务是预测接下来最可能的单词。模型可能输出类似以下的概率分布：

```
{
  "mat": 0.6,
  "table": 0.2,
```

```
  "roof": 0.1,
  "tree": 0.1
}
```

在这个例子中，模型预测最可能的下一个单词是“mat”（地毯），并为每个可能的单词赋予了一个概率分数。

3. **语言模型的【生成】任务**：同样在Jupyter Notebook中，你需要完成 `generate` 函数，该函数会根据给定的输入序列生成一个完整的单词序列。
   a. 生成任务则是从部分文本开始，持续生成单词，直到形成一个完整的句子或段落。

   举个例子：

   假设你给模型输入一句话的开头 `"Once upon a time"`，模型的任务是继续生成一段完整的句子或段落。模型可能生成的序列是：

```
"Once upon a time, there was a little girl who lived in a small village."
从前，有一个小女孩住在一个小村庄里。
```

二 Masked Self-Attention

注意力机制概述

1. 注意力机制的核心概念

注意力机制的核心是系统在处理大数据集时，能够聚焦于其中的特定元素，并为输入的不同部分分配不同的重要性权重。通过这种机制，模型可以选择性地处理和加权信息，从而更加高效且具有上下文感知地处理复杂数据。这种机制在自然语言理解、图像识别以及许多其他机器学习任务中具有重要应用。

2. 注意力机制的不同类型

2.1 自注意力 (Self-attention)

- **定义**：自注意力是一种分析单个序列内部关系的机制。在处理文本时，它用于衡量序列中不同单词之间的相关性。自注意力机制使得每个单词可以根据序列中其他单词的重要性调整其表示。
- **应用场景**：通常用于语言模型中，如Transformer模型，在无须考虑顺序的情况下计算单词之间的依赖关系。

2.2 掩码自注意力 (Masked Self-attention)

- **定义**：掩码自注意力类似于自注意力，但它通过掩码来防止某些数据影响当前的位置。这种机制可以屏蔽序列中的部分信息。
- **类型**：
  - **因果掩码 (Causal mask)**：阻止序列中的当前位置看到未来位置的值。这确保了预测仅依赖于过去的数据。这在生成任务中非常重要，比如语言模型生成下一个单词时只能看到前面的单词。
  - **填充掩码 (Padding mask)**：消除填充符（padding tokens）在序列中的影响。在处理变长序列时尤其有用，确保填充部分不影响实际数据。
  - **局部掩码 (Localized mask)**：将注意力限制在每个标记周围的一个邻近范围内。这种方法可以减少计算复杂度，同时关注局部相关性。

2.3 交叉注意力 (Cross-attention)

- **定义**：交叉注意力侧重于两个不同序列之间的交互，例如在翻译任务中，从源语言序列到目标语言序列的映射。通过这种机制，模型可以增强相关性和对齐，从而更好地捕捉两个序列之间的联系。
- **应用场景**：多模态数据处理（例如视觉问答），以及机器翻译中的编码器-解码器架构。

2.4 多头注意力 (Multi-head attention)

- **定义**：多头注意力在并行运行多个注意力机制，每个头捕捉数据中的不同关系和细微差别。每个注意力头关注输入的不同部分，最终将它们的结果拼接起来，形成更加丰富的表示。
- **应用场景**：广泛应用于Transformer模型中，如BERT、GPT等，通过多头注意力机制，模型能够同时关注多个方面的信息。

Masked Self-Attention（加性因果掩码自注意力）

1. 关键概念解释

- **Keys（键）**：
  - 键是从输入数据中派生的元素的表示形式，模型需要关注这些元素。键编码了有关这些元素的特定信息。在自然语言处理的上下文中，键可以表示句子中的单词或标记；在计算机视觉中，键可能对应于图像中的空间位置。
- **Queries（查询）**：
  - 查询是从输入数据中派生的另一组表示形式。它们用于从键中查找信息。查询编码了模型在输入中具体寻找的内容。例如，在语言翻译中，查询可能代表目标语言中的一个单词，模型使用这个查询来查找源语言中的相关信息。
- **Values（值）**：
  - 值是从输入数据中派生的另一组表示形式，它们包含了模型感兴趣的实际信息。值与键相关联，可以被认为是那些特定位置的内容。当查询和键紧密匹配时，与这些键关联的值在最终输出中被赋予更大的重要性。

2. 加性因果掩码 (Additive Causal Mask)

- **定义**：
  - 这种掩码确保一个位置只受过去和当前数据的影响，而不会被未来的数据影响。"因果"意味着序列中的每个位置都不能看到未来的位置。"加性"指的是在注意力矩阵的主对角线之上添加大的负数（例如，-1e9）。这些负数在softmax步骤后会变成零，从而有效地隐藏了未来的位置。
- **作用**：
  - 这种方法在序列到序列的任务中至关重要，因为它确保信息流的正确性。我们使用加法掩码而不是乘法掩码，因为加法操作可以保持softmax步骤的稳定性，防止出现数值不稳定或溢出的情况。

前向传播

1. 注意力机制中的主要组件（Attention Components）

表格中列出了注意力机制中涉及的主要组件，具体如下：

代码名称	数学表示	类型	形状	说明

W_q	W_q	矩阵	$D \times D_k$	查询的权重矩阵
W_k	W_k	矩阵	$D \times D_k$	键的权重矩阵
W_v	W_v	矩阵	$D \times D_v$	值的权重矩阵
X	X	矩阵	$B \times T \times D$	输入到注意力机制的数据
K	K	矩阵	$B \times T \times D_k$	键
Q	Q	矩阵	$B \times T \times D_k$	查询
V	V	矩阵	$B \times T \times D_v$	值
A_w	A_w	矩阵	$B \times T \times T$	原始注意力权重
A_sig	A_σ	矩阵	$B \times T \times T$	原始注意力权重的Softmax输出
X_new	X_n	矩阵	$B \times T \times D_v$	最终的注意力上下文
dLdX_new	$\frac{\partial L}{\partial X_{new}}$	矩阵	$B \times T \times D_v$	关于注意力上下文的损失梯度
dLdA_sig	$\frac{\partial L}{\partial A_\sigma}$	矩阵	$B \times T \times T$	关于注意力权重的损失梯度
dLdV	$\frac{\partial L}{\partial V}$	矩阵	$B \times T \times D_v$	关于值的损失梯度
dLdA_w	$\frac{\partial L}{\partial A_w}$	矩阵	$B \times T \times T$	关于原始注意力权重的损失梯度
dLdK	$\frac{\partial L}{\partial K}$	矩阵	$B \times T \times D_k$	关于键的损失梯度
dLdQ	$\frac{\partial L}{\partial Q}$	矩阵	$B \times T \times D_k$	关于查询的损失梯度
dLdW_q	$\frac{\partial L}{\partial W_q}$	矩阵	$D \times D_k$	关于查询权重的损失梯度
dLdW_v	$\frac{\partial L}{\partial W_v}$	矩阵	$D \times D_v$	关于值权重的损失梯度
dLdW_k	$\frac{\partial L}{\partial W_k}$	矩阵	$D \times D_k$	关于键权重的损失梯度
dLdX	$\frac{\partial L}{\partial X}$	矩阵	$B \times T \times D$	关于输入的损失梯度

2. 前向传播方程（Forward Equations）

前向传播计算过程中涉及的主要方程如下：

1. **计算【键】**：

$$K = X \cdot W_k \in \mathbb{R}^{B \times T \times D_k}$$

输入数据通过键的权重矩阵 W_k 计算得到键矩阵 K。

2. **计算【值】**：

$$V = X \cdot W_v \in \mathbb{R}^{B \times T \times D_v}$$

输入数据通过值的权重矩阵 W_v 计算得到值矩阵 V。

3. **计算【查询】**：

$$Q = X \cdot W_q \in \mathbb{R}^{B \times T \times D_k}$$

输入数据通过查询的权重矩阵 W_q 计算得到查询矩阵 Q。

4. **计算原始注意力权重（logits）**：

$$A_w = Q \cdot K^T \in \mathbb{R}^{B \times T \times T}$$

查询矩阵 Q 与键矩阵 K 的转置相乘得到原始注意力权重矩阵 A_w。

5. **应用掩码**：

$$A_w(\text{w/mask}) = A_w + \text{mask} \in \mathbb{R}^{B\times T\times T}$$

将掩码添加到原始注意力权重矩阵 A_w 上，得到加了掩码的注意力权重矩阵。

6. **计算Softmax归一化注意力权重**：

$$A_\sigma = \sigma\left(\frac{A_w}{\sqrt{D_k}}\right) \in \mathbb{R}^{B\times T\times T}$$

将加掩码后的注意力权重除以 D_k 的平方根，并进行softmax归一化，得到最终的注意力权重矩阵 A_σ。

7. **计算最终注意力上下文**：

$$X_n = A_\sigma \cdot V \in \mathbb{R}^{B\times T\times D_v}$$

使用归一化后的注意力权重矩阵 A_σ 与值矩阵 V 相乘，得到最终的注意力上下文矩阵 X_n。

反向传播

反向传播涉及计算关于各种参数和输入的梯度，以便在训练过程中更新模型权重。以下是反向传播过程中涉及的主要方程：

1 关于注意力权重（原始和归一化）的梯度

- **关于归一化注意力权重的梯度**：

$$\frac{\partial L}{\partial A_\sigma} = \left(\frac{\partial L}{\partial X_n}\right)\cdot V^T \in \mathbb{R}^{B\times T\times T}$$

这里，$\frac{\partial L}{\partial X_n}$ 是关于最终注意力上下文 X_n 的损失梯度，与值矩阵 V 的转置相乘得到关于归一化注意力权重 A_σ 的梯度。

- **关于原始注意力权重的梯度**：

$$\frac{\partial L}{\partial A_w} = \frac{1}{\sqrt{D_k}}\cdot\sigma'\left(\frac{\partial L}{\partial A_\sigma}\right) \in \mathbb{R}^{B\times T\times T}$$

原始注意力权重的梯度通过归一化注意力权重的梯度反向传播得到，其中包含了对softmax函数的导数计算。

2 关于键、查询和值的梯度

- **关于值的梯度**：

$$\frac{\partial L}{\partial V} = A_\sigma^T \cdot \left(\frac{\partial L}{\partial X_n}\right) \in \mathbb{R}^{B\times T\times D_v}$$

值矩阵的梯度由归一化后的注意力权重矩阵 A_σ 与 X_n 的梯度相乘得到。

- **关于键的梯度**：

$$\frac{\partial L}{\partial K} = \left(\frac{\partial L}{\partial A_w}\right)\cdot Q^T \in \mathbb{R}^{B\times T\times D_k}$$

键矩阵的梯度通过原始注意力权重的梯度与查询矩阵 Q 的转置相乘得到。

- **关于查询的梯度**：

$$\frac{\partial L}{\partial Q} = \left(\frac{\partial L}{\partial A_w}\right)\cdot K^T \in \mathbb{R}^{B\times T\times D_k}$$

查询矩阵的梯度通过原始注意力权重的梯度与键矩阵 K 的转置相乘得到。

3 关于权重矩阵的梯度

注意在批次中的所有输入都会影响相应的键、查询和值，因此需要对每个输入的梯度求和。

- **关于查询权重矩阵的梯度**：

$$\frac{\partial L}{\partial W_q} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial Q}\right) \in \mathbb{R}^{D \times D_k}$$

- **关于值权重矩阵的梯度**:

$$\frac{\partial L}{\partial W_v} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial V}\right) \in \mathbb{R}^{D \times D_v}$$

- **关于键权重矩阵的梯度**:

$$\frac{\partial L}{\partial W_k} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial K}\right) \in \mathbb{R}^{D \times D_k}$$

4 关于输入的梯度

- **关于输入的梯度**:

$$\frac{\partial L}{\partial X} = \left(\frac{\partial L}{\partial V}\right) \cdot W_v^T + \left(\frac{\partial L}{\partial K}\right) \cdot W_k^T + \left(\frac{\partial L}{\partial Q}\right) \cdot W_q^T \in \mathbb{R}^{B \times T \times D}$$

  输入的梯度通过关于值、键和查询的梯度分别与相应的权重矩阵转置相乘得到，并最终加和。

代码

```
import torch

class Softmax:

    '''
    不要修改！Softmax类已经在Attention类的构造函数中初始化，可以直接使用。
    该类实现了在最后一个维度上的softmax操作。
    '''
    def forward(self, Z):

        # 存储输入Z的原始形状，以便后续reshape操作
        z_original_shape = Z.shape

        # N是批次大小乘以序列长度，C是输入的最后一个维度（词汇量的大小）
        self.N = Z.shape[0] * Z.shape[1]
        self.C = Z.shape[2]

        # 将Z reshape成二维矩阵，以便进行矩阵运算
        Z = Z.reshape(self.N, self.C)

        # 初始化单位矩阵用于进行softmax归一化
        Ones_C = torch.ones((self.C, 1))

        # 计算softmax归一化值，并将结果存储在self.A中
        self.A = torch.exp(Z) / (torch.exp(Z) @ Ones_C)

        # 返回与原始Z形状一致的softmax结果
        return self.A.reshape(z_original_shape)

    def backward(self, dLdA):

        # 存储dLdA的原始形状，以便后续reshape操作
        dLdA_original_shape = dLdA.shape
```

```
        # 将dLdA reshape成二维矩阵，以便进行矩阵运算
        dLdA = dLdA.reshape(self.N, self.C)

        # 初始化dLdZ，用于存储对Z的梯度
        dLdZ = torch.zeros((self.N, self.C))

        # 遍历每一个样本，计算每个样本的Jacobian矩阵
        for i in range(self.N):

            # 初始化Jacobian矩阵
            J = torch.zeros((self.C, self.C))

            # 计算Jacobian矩阵的每个元素
            for m in range(self.C):
                for n in range(self.C):
                    if n == m:
                        # 对角线元素的梯度公式
                        J[m, n] = self.A[i][m] * (1 - self.A[i][m])
                    else:
                        # 非对角线元素的梯度公式
                        J[m, n] = -self.A[i][m] * self.A[i][n]

            # 计算当前样本的dLdZ
            dLdZ[i, :] = dLdA[i, :] @ J

        # 返回与原始dLdA形状一致的dLdZ
        return dLdZ.reshape(dLdA_original_shape)


class Attention:

        def __init__(self, weights_keys, weights_queries, weights_values):

            """
            初始化Attention类的权重参数。
            输入维度为D，键和查询的维度为D_k，值的维度为D_v

            参数说明:
            -----------
            weights_keys (torch.tensor, dim = (D X D_k)): 键的权重矩阵
            weights_queries (torch.tensor, dim = (D X D_k)): 查询的权重矩阵
            weights_values (torch.tensor, dim = (D X D_v)): 值的权重矩阵
            """

            # 将键、查询和值的权重存储为类的参数
            self.W_k = weights_keys
            self.W_q = weights_queries
            self.W_v = weights_values

            # 存储D_k和D_v的维度信息
            self.Dk = weights_keys.shape[1]
            self.Dv = weights_values.shape[1]
```

```
        # 初始化Softmax实例，用于后续的归一化计算
        self.softmax = Softmax()

    def forward(self, X):

        """
        计算自注意力层的输出。
        该函数将存储键、查询、值，以及未归一化和归一化的注意力权重。
        输入为一个批次数据，而非单一序列，因此在操作时应注意维度转换。

        输入
        -----
        X (torch.tensor, dim = (B, T, D)): 输入的批次张量

        返回
        ------
        X_new (torch.tensor, dim = (B, T, D_v)): 输出的批次张量
        """

        # 存储输入X
        self.X = X

        # 计算查询、键和值的矩阵
        self.Q = X @ self.W_q
        self.K = X @ self.W_k
        self.V = X @ self.W_v

        # 计算未归一化的注意力分数（logits）
        self.A_w = self.Q @ self.K.transpose(1, 2)


        # 创建加性因果注意力掩码并应用
        # 使用torch.triu生成上三角矩阵作为掩码
        mask_single = torch.triu(torch.ones(self.A_w.shape[1], self.A_w.shape[2]), diagonal=1)
        # 为批次中的每个样本应用掩码
        mask_batch = mask_single.unsqueeze(0).repeat(self.A_w.shape[0], 1, 1)
        # 将掩码中为1的地方填充为负无穷大，避免其影响计算
        mask_final = mask_batch.masked_fill(mask_batch == 1, float('-inf'))

        # 将掩码应用到注意力分数中
        self.A_w_attn_mask = self.A_w + mask_final


        # 归一化注意力分数
        self.A_sig = self.softmax.forward(self.A_w_attn_mask / torch.sqrt(torch.tensor(self.Dk)))

        # 计算注意力上下文
        X_new = self.A_sig @ self.V

        # 返回新计算的上下文向量
        return X_new

    def backward(self, dLdXnew):
```

```
            """
            通过自注意力层进行反向传播。
            该函数将存储关于键、查询、值和权重矩阵的导数。
            输入为一个批次数据，而非单一序列，因此在操作时应注意维度转换。

            输入
            -----
            dLdXnew (torch.tensor, dim = (B, T, D_v)): 关于注意力层输出的梯度

            返回
            ------
            dLdX (torch.tensor, dim = (B, T, D)): 关于注意力层输入的梯度
            """

            # 计算关于归一化注意力权重的梯度
            dLdA_sig = dLdXnew @ self.V.transpose(1, 2)
            # 计算关于未归一化注意力权重的梯度
            dLdA_w = (1 / torch.sqrt(torch.tensor(self.Dk))) *
self.softmax.backward(dLdA_sig).transpose(1, 2)

            # 计算关于值、键、查询的梯度
            self.dLdV = self.A_sig.transpose(1, 2) @ dLdXnew
            self.dLdK = dLdA_w @ self.Q
            self.dLdQ = dLdA_w.transpose(1, 2) @ self.K

            # 计算关于权重矩阵的梯度（需要在批次维度上求和）
            self.dLdWq = torch.sum(self.X.transpose(1, 2) @ self.dLdQ, dim=0)
            self.dLdWv = torch.sum(self.X.transpose(1, 2) @ self.dLdV, dim=0)
            self.dLdWk = torch.sum(self.X.transpose(1, 2) @ self.dLdK, dim=0)

            # 计算关于输入X的梯度
            dLdX = self.dLdV @ self.W_v.T + self.dLdK @ self.W_k.T + self.dLdQ @ self.W_q.T

            # 返回关于输入X的梯度
            return dLdX
```

三 语言模型的【预测】和【生成】任务

看了很久S24版HW4P1讲义内容和S23和S24届人的代码，基本清楚是要干什么了。语言表达一下，具体细节看代码。

本项目语音模型的主要任务：

1. 输入一句话或一个词，模型能预测这句话下一个单词最可能是什么。
2. 输入一句话或一个词，模型能生成指定字数的新内容。

1 数据集形态

给你一个数据集，npy文件存着，里面有500多篇英语文章（总词汇个数是207万）。文章中单词、字符、数字等全部用的索引号存储着。索引号对应一个33280个元素的字典，具体可查看dataset中csv文件（里面缺少sos和eos）。

特别注意，这个字典文件，里面既有数字，又有单词，又有各种杂乱的符号，还有数字+单词+符号的拼接形式。此外，文章中大部分句子的完整性都很差，很多句话中存在各种杂乱符号，比如<unk>、@、===、等等。下图就是val集6句话的开头内容：

```
<sos> Division crossed the <unk> at a number of places and climbed the hills quietly
 <sos> = <eol> = = = French VIII . Corps ( Corps <unk> ) = = = <eol> On 6 November |
 <sos> of the World from 9th Avenue " . This is regarded as his most famous work . I
 <sos> – <unk> @-@ 10 , <unk> @-@ 12 , <unk> @-@ 16 , <unk> @-@ 17 – were all conver
 <sos> And now he has . " <eol> = = Family = = <eol> <unk> lived 37 of his years in
 <sos> Hell to which he has been condemned for <unk> . Eliot , in a letter to John <
```

2 训练集和标签制作

数据集：

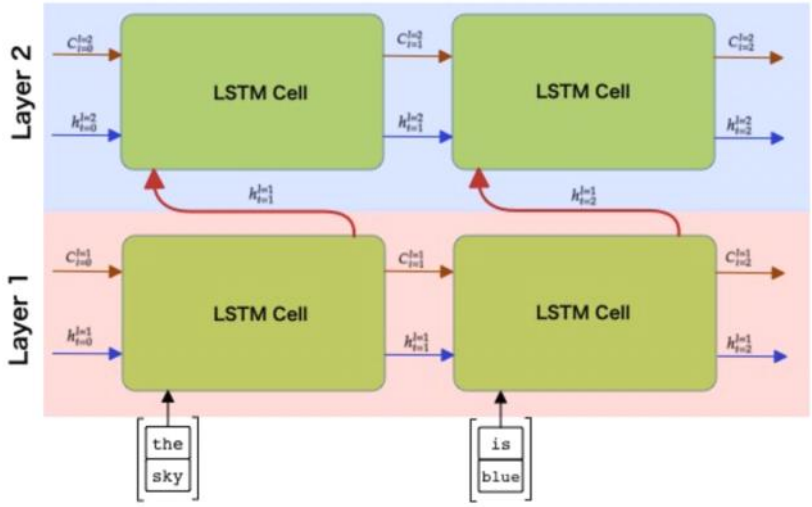
我们要把上述500多篇文章，先首尾拼接起来，然后开始从头到尾，切香肠一样，切成一个个序列，每个序列长度是个可自定义的参数，比如设定为10。500多篇文章总词汇个数是207万，就可切成20万个句子序列。

标签：

标签的作用是用来判断模型每个时间步预测出的单词跟句子中原始位置的单词是否相同。所以标签也需要切分句子，只是切分时需要跟训练集句子错位对齐。比如：模型输入是 `[The cat sat on the]`，标签得是 `[cat sat on the mat]`。即将‘the’输入模型后，会得到下一个单词的预测，我们需要把这个预测的单词跟标签中的‘cat’计算损失，当第二个时间步输入‘cat’时，我们需要把模型的预测结果跟‘Sat’进行损失计算。这种对齐方式使得模型能够学习到如何根据前面的单词预测下一个单词。

3 模型结构

本项目应该是限定用如下网络结构：（正则化方法不限定）



```
LanguageModel(
  (token_embedding): Embedding(33280, 128)
  (lstm_cells): Sequential(
    (0): LSTMCell(128, 128)
    (1): LSTMCell(128, 128)
  )
  (token_probability): Linear(in_features=128, out_features=33280, bias=True)
)
```

输入输出：

1. 输入句子（每个句子长度是10，每个单词用索引号表示，即一个[10,]向量）
2. 对句子进行编码：torch.nn.Embedding(num_embeddings=vocab_size, embedding_dim=embed_dim)
3. 输入两层LSTM中
4. 输入一个全连接中，将维度从128映射到33280，即表示每个词的预测概率

如果序列长度设定为10，则完整的网络结构如下：

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
LanguageModel                            [32, 10, 33280]           --
├─Embedding: 1-1                         [32, 10, 128]             4,259,840
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-1                     [32, 128]                 132,096
│    └─LSTMCell: 2-2                     [32, 128]                 132,096
├─Linear: 1-3                            [32, 33280]               4,293,120
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-3                     [32, 128]                 (recursive)
│    └─LSTMCell: 2-4                     [32, 128]                 (recursive)
├─Linear: 1-5                            [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-5                     [32, 128]                 (recursive)
│    └─LSTMCell: 2-6                     [32, 128]                 (recursive)
├─Linear: 1-7                            [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-7                     [32, 128]                 (recursive)
│    └─LSTMCell: 2-8                     [32, 128]                 (recursive)
├─Linear: 1-9                            [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-9                     [32, 128]                 (recursive)
│    └─LSTMCell: 2-10                    [32, 128]                 (recursive)
├─Linear: 1-11                           [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-11                    [32, 128]                 (recursive)
│    └─LSTMCell: 2-12                    [32, 128]                 (recursive)
├─Linear: 1-13                           [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-13                    [32, 128]                 (recursive)
│    └─LSTMCell: 2-14                    [32, 128]                 (recursive)
├─Linear: 1-15                           [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-15                    [32, 128]                 (recursive)
│    └─LSTMCell: 2-16                    [32, 128]                 (recursive)
├─Linear: 1-17                           [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-17                    [32, 128]                 (recursive)
│    └─LSTMCell: 2-18                    [32, 128]                 (recursive)
├─Linear: 1-19                           [32, 33280]               (recursive)
├─Sequential: 1-20                       --                        (recursive)
│    └─LSTMCell: 2-19                    [32, 128]                 (recursive)
│    └─LSTMCell: 2-20                    [32, 128]                 (recursive)
├─Linear: 1-21                           [32, 33280]               (recursive)
==========================================================================================
Total params: 8,817,152
Trainable params: 8,817,152
Non-trainable params: 0
Total mult-adds (G): 12.33
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 86.18
Params size (MB): 35.27
Estimated Total Size (MB): 121.45
```

batch,序列长度，词汇表长度

embedding编码维度

图中有10对重复的结构，原因是序列输入长度为10，即10个单词逐一进入LSTM，需要自循环10次，出来新的10个预测单词。

4 训练原理

500多篇文章，首尾先加上sos和eos标志符，然后把所有文章拼接起来，形成207万总长度的数组，每10个单词切成一个序列，总共就是20万左右序列句子。

特别注意：本项目因为数据集很少，epoch数又只定了10个，所以bathsize不能很大，或者说batch越小，最终的精度会越高。本质是因为每个epoch，模型参数更新的次数会更多，因为数据集实在是太少了。（测试发现，其他参数完全相同情况下，batch=256，NLL=5.1左右，batch=32，NLL=4.7）

模型的输入：（batchsize, 10）

模型的输出：（batchsize，10，Vocab_size），即批次大小、时间步（句子的单词个数）和词汇表大小

损失函数计算：使用交叉熵损失，判断在不同时间步预测的10单词，跟标签中不同位置的10个单词是否一致。

5 【预测】的val、test集验证原理

val集总共128句话，每个句子21个词汇长度。val集的标签是每句话第22个单词具体是什么。把val集句子输入训练好的模型，模型会输出（batchsize，21，Vocab_size）。只取最后一个时间步的输出，即模型对第22个单词的预测结果，维度是（batchsize，Vocab_size）。

预测结果好坏的评估，用的负对数似然（Negative Log-Likelihood, NLL），这是一个常用的损失函数，特别是在分类任务中。

下面是这个函数的详细解释：

负对数似然函数逐行解释

out维度：（batchsize，Vocab_size）。模型对句子第22个单词的预测结果。元素值是全连接层出来的。

targ维度：（batchsize，）。句子真实的第22个单词结果。元素值是词汇表中的索引号。

```
def get_prediction_nll(out, targ):
    out = log_softmax(out, 1)  # 对模型的输出应用 log_softmax 函数
    nlls = out[np.arange(out.shape[0]), targ]  # 提取 out 中所有样本在 targ 所指定类别索引上的对数概率。
    nll = -np.mean(nlls)  # 计算负对数似然的平均值
    return nll  # 返回负对数似然
```

1. `log_softmax(out, 1)`

- `log_softmax` 是对模型的输出 `out` 进行处理，计算它在类别维度（通常是 `axis=1`）上的对数概率。
- `softmax` 将模型的原始输出转换为概率分布，而 `log_softmax` 则进一步对这些概率取对数，输出的形状与输入 `out` 一样。
- 计算对数的原因是，直接计算概率可能会导致数值下溢，而取对数后可以将乘法操作转换为加法，从而避免数值下溢问题。

2. `nlls = out[np.arange(out.shape[0]), targ]`

- 这一步从 `log_softmax` 的输出中提取与目标标签 `targ` 对应的对数概率值。
- `np.arange(out.shape[0])` 生成了一个从 `0` 到 `batch_size - 1` 的数组，用于索引批次中的每个样本。
- `targ` 是每个样本的真实标签，用于选择每个样本在类别维度（`axis=1`）上的对数概率值。
- 结果是一个一维数组 `nlls`，其中每个元素都是该样本的目标类别对应的对数概率。

3. `nll = -np.mean(nlls)`

- 计算 `nlls` 数组的平均值并取负，得到当前批次的负对数似然。
- 负对数似然衡量的是模型在目标类别上的预测性能。值越低，表明模型的预测越好，因为对数概率越接近0（即概率越接近1）。

4. 返回 `nll`

- 返回计算得到的负对数似然（NLL），用于评估模型在当前批次上的表现。

计算 NLL 的原理

负对数似然（NLL）是分类任务中常用的损失函数之一，其目标是最大化模型在真实标签上的预测概率。计算步骤如下：

1. **计算对数概率**：首先，通过 `log_softmax` 函数将模型的输出转换为对数概率。这一步消除了对概率直接操作的数值不稳定性。
2. **提取目标标签的对数概率**：从对数概率中，提取与真实标签对应的值。这个值越大，说明模型越确信这个标签是正确的。
3. **计算平均负对数概率**：将所有样本的目标标签对应的对数概率取平均，并取负，得到整体的NLL值。负对数似然越小，表示模型的预测越好。

总结

这个函数通过计算负对数似然（NLL）来评估模型的预测质量。负对数似然越小，表示模型对目标标签的预测越准确。这个指标在分类任务中非常常用，因为它直接衡量了模型对正确标签的预测概率。

6 【生成】的val、test集验证原理

【生成】专业术语讲就是【序列补全】（**Sequence Completion**）：

- 给定一个部分的词序列，如 $< sos > fourscoreand$，模型需要生成接下来的 N 个词来扩展或完成序列。
- 模型会逐步生成序列中的下一个词，并在一定步数或生成了 $< eos >$ 标记时停止。
- 如果模型训练得很好，这个扩展（或补全）序列应该是“自然的”。然而，这个“自然”的定义较为模糊，所以在理想情况下，模型可以通过一个“oracle”（如另一个在大量数据上训练的语言模型）来评分生成的序列，评分越高，模型的生成效果越好。

本项目【生成】的val集总共32个句子，每个句子21个单词。test集128句话，每句话31个单词。

本项目需要生成句子的未来10个词。本质上【生成】任务就是上述【预测】任务的延伸，当我输入句子的21个单词，获得第22个单词的预测结果时，我把这个预测结果及其LSTM中的长短期记忆状态信息继续输入下一个时间步LSTM，进而得到第23个单词的预测结果，以此类推，直到得到未来第10个单词的预测结果。

val和test集的结果都能在本地预测出来后，将原始输入句子和未来预测的10个单词拼接起来，自己看看是否前言是否搭后语。对于提交作业的分数，必须将test集128话的预测结果，上传GPT API进行评分。

精度的评估方式：

本作业使用GPT API来评估序列补全任务的精度，具体使用的是困惑度（Perplexity, PPL） 作为评估指标。以下是这个评估方法的具

体原理：

GPT 计算困惑度的原理基于模型对序列中每个词的预测概率。困惑度（Perplexity, PPL）是衡量语言模型预测下一个词的能力的指标，反映了模型在生成语言序列时的不确定性。以下是 GPT 计算困惑度的详细原理：

1. 序列概率计算

对于给定的词序列 $w_1, w_2, \ldots, w_T$，GPT 作为一种自回归模型，会依次计算每个词的条件概率：

$$P(w_1, w_2, \ldots, w_T) = P(w_1) \times P(w_2|w_1) \times \cdots \times P(w_T|w_1, w_2, \ldots, w_{T-1})$$

其中，$P(w_t|w_1, w_2, \ldots, w_{t-1})$ 表示在给定前面词的情况下，模型预测当前词 w_t 的概率。

2. 对数似然

为了计算困惑度，首先要计算整个序列的对数似然：

$$log\ P\ (w_1, w_2, \ldots, w_T) = log\ P\ (w_1) + log\ P\ (w_2|w_1) + \cdots + log\ P\ (w_T|w_1, w_2, \ldots, w_{T-1})$$

对数似然的目的是将乘法转换为加法，简化计算，同时也防止数值下溢。

3. 计算平均对数似然

困惑度是基于序列的平均对数似然计算的。平均对数似然定义为：

$$\frac{1}{T} \sum_{t=1}^{T} log\ P\ (w_t|w_1, w_2, \ldots, w_{t-1})$$

4. 计算困惑度

困惑度（Perplexity, PPL）是对平均对数似然的指数形式，它的公式如下：

$$\mathrm{PPL} = 2^{-\frac{1}{T} \sum_{t=1}^{T} log_2\ P(w_t|w_1, w_2, \ldots, w_{t-1})}$$

困惑度的物理意义是“在当前模型下，预测下一个词的平均可能性是多少”。具体解释如下：

- **低困惑度**：表示模型对每个词的预测很确定（概率接近1），生成的文本流畅自然。
- **高困惑度**：表示模型在预测下一个词时非常不确定（概率分布很分散），生成的文本质量较差。

5. GPT 如何计算困惑度

在 GPT 模型中，困惑度的计算可以总结为以下步骤：

1. **输入序列**：将一个文本序列输入到 GPT 模型中。
2. **计算条件概率**：GPT 模型通过前向传播计算每个词的条件概率 $P(w_t|w_1, \ldots, w_{t-1})$。
3. **求对数似然**：对这些条件概率取对数并求和，得到整个序列的对数似然。
4. **计算困惑度**：将对数似然转化为困惑度，这个值反映了模型在生成这段文本时的预测质量。

总结

GPT 计算困惑度的原理基于模型预测词序列的条件概率，通过对数似然来评估模型生成的文本与真实数据的匹配程度。困惑度是一个综合性指标，低困惑度意味着模型生成的文本与真实语言的匹配程度更高，文本质量更好。

为什么要用GPT API计算困惑度?

虽然理论上可以通过函数计算困惑度，但使用 GPT API 能够提供更准确、更客观的评估结果。它不仅保证了评估基准的一致性，而且利用了 GPT 等大规模预训练模型的强大能力，从而更有效地评估生成文本的质量。

7 我的训练结果 及 作业精度要求

作业精度要求：

- 【预测】：val集nll低于4.6。test集低于5.3（上传课程服务器评估）
- 【生成】：困惑度低于1400。（上传GPT API评估，大概需要充值5美金）

我的结果：

batch=32（由于数据很少，batch越大，精度越低，本质可能是每个epoch反向传播次数少了很多）

epoch=20

优化器：adam

学习器：余弦退火（初始学习率1e-3）

损失函数：交叉熵

序列切片长度：10个单词一句话

LSTM中隐藏单元：128

embedding维度：128

【预测】的val集结果：nll=4.76~4.85

【生成】的val集结果：没有用GPT API测试，具体生成效果见下图：

```
[TRAIN]         Epoch [10/10]   Loss: 4.8391    Lr: 0.000025
[VAL]   Epoch [10/10]   Loss: 4.8564
4.8563704
Input ‖ Output #0: <sos> while the group was en route , but only three were ultimately able to attack . None of them were ‖ the first to be a <unk> of the <unk> .
Input ‖ Output #1: <sos> <unk> , where he remained on loan until 30 June 2010 . <eol> = = = Return to Manchester United ‖ States = = = <eol> The first @-@ class ships
Input ‖ Output #2: <sos> 25 April 2013 , denoting shipments of 500 @,@ 000 copies . <eol> The song became One Direction 's fourth ‖ season , he was a <unk> of the <unk> <unk>
Input ‖ Output #3: <sos> , and Bruce R. ) one daughter ( Wendy J. <unk> ) and two grandchildren , died in <unk> , ‖ and the <unk> of the <unk> , and the <unk>
Input ‖ Output #4: <sos> Warrior were examples of this type . Because their armor was so heavy , they could only carry a single ‖ from the <unk> of the <unk> . <eol> = =
Input ‖ Output #5: <sos> the embassy at 1 : 49 and landed on Guam at 2 : 23 ; twenty minutes later , Ambassador ‖ in the United States . <eol> = = = <unk>
Input ‖ Output #6: <sos> <unk> , $ 96 million USD ) . Damage was heaviest in South Korea , notably where it moved ashore ‖ to the north and south @-@ west of the city
Input ‖ Output #7: <sos> The <unk> were condemned as <unk> by <unk> , who saw the riots as hampering attempts to resolve the situation ‖ . The <unk> of the <unk> <unk> <unk> <unk> ,
Input ‖ Output #8: <sos> by a decision made by the War Office in mid @-@ 1941 , as it was considering the equipment to ‖ the <unk> of the <unk> <unk> <unk> , and <unk>
Input ‖ Output #9: <sos> Division crossed the <unk> at a number of places and climbed the hills quietly toward the 9th Infantry river line ‖ . The <unk> of the <unk> <unk> <unk> <unk> ,
Input ‖ Output #10: <sos> = <eol> = = = French VIII . Corps ( Corps <unk> ) = = = <eol> On 6 November ‖ , the United States entered World War I , and
Input ‖ Output #11: <sos> of the World from 9th Avenue " . This is regarded as his most famous work . It is considered ‖ to be a <unk> of the <unk> <unk> <unk> ,
Input ‖ Output #12: <sos> – <unk> @-@ 10 , <unk> @-@ 12 , <unk> @-@ 16 , <unk> @-@ 17 – were all converted ‖ to the north and south @-@ west of the city
Input ‖ Output #13: <sos> And now he has . " <eol> = = Family = = <eol> <unk> lived 37 of his years in ‖ the United States , the <unk> of the <unk> <unk>
Input ‖ Output #14: <sos> Hell to which he has been condemned for <unk> . Eliot , in a letter to John <unk> dated 27 ‖ May , the <unk> of the <unk> <unk> <unk> <unk>
Input ‖ Output #15: <sos> Luoyang area , fulfilling his duties in domestic affairs . <eol> In the autumn of <unk> , he met Li ‖ Jie 's first appearance in the United States , the
Input ‖ Output #16: <sos> Power said they enjoyed Block Ball and its number of stages , but wondered how its eight <unk> of memory ‖ , and the <unk> of the <unk> , and the
Input ‖ Output #17: <sos> by Lloyd F. Lonergan . The cameraman was Jacques <unk> . <eol> = = Release and reception = = <eol> ‖ The first @-@ class ships were not included in the
Input ‖ Output #18: <sos> alone , the Austrians lost more than half their reserve artillery park , 6 @,@ 000 ( out of 8 ‖ @.@ 5 m ) . The first @-@ game was
Input ‖ Output #19: <sos> while attacking a ship at <unk> in the Dutch East Indies ; the loss was compounded by the fact that ‖ the <unk> of the <unk> , and the <unk> of
Input ‖ Output #20: <sos> first raised in 2007 by the member of parliament ( MP ) for <unk> . The gangsters may have run ‖ the <unk> of the <unk> , and the <unk> of
Input ‖ Output #21: <sos> Species are also non @-@ spiny <unk> and includes both large trees with stout stems up to 30 metres ( ‖ 1 @.@ 5 m ) . <eol> = = =
...
Input ‖ Output #29: <sos> , but began patrolling the English Channel after <unk> @-@ 6 pioneered a route past British <unk> nets and mines ‖ . <eol> = = = <unk> = = = <eol>
Input ‖ Output #30: <sos> production executives to let him direct . He had already discussed the film with <unk> and Cohen , and felt ‖ that the " <unk> " , " <unk> " ,
Input ‖ Output #31: <sos> and Nick <unk> at Studio <unk> in Los Angeles , California , and was released on August 1 , 2006 ‖ , the first time in the United States , the
```

8 讲义中提到的精度提升建议

P1类型作业不是kaggle，重要的是理解整个项目的原理和代码实现方法，所以对模型结构什么的不需要去复杂化。只是讲义中提到可使用一些正则化方法提升精度。如下（我没去实践代码测试，S23和S24届人的代码也没看到在本项目中有添加这些方法）：

1. Locked Dropout（锁定的Dropout）

- **原理**：在标准的dropout中，每次调用dropout函数时会生成一个新的二进制dropout掩码。而在Locked Dropout（也称为Variational Dropout）中，仅在第一次调用时生成一次dropout掩码，并在前向和后向传播过程中重复使用该掩码。
- **效果**：这种方式确保了LSTM的输入和输出在整个训练过程中使用一致的dropout掩码，从而提高模型的稳定性和正则化效果。

2. Embedding Dropout

- **原理**：Embedding Dropout是在词嵌入矩阵上进行的dropout，作用于词的整个embedding向量。其做法是对每个词的embedding向量应用dropout，并将剩余的非drop出的词embedding向量按比例缩放。
- **效果**：这种方法在整个前向和后向传播过程中都对每个特定的词进行dropout，有效减少了过拟合的风险。

3. Weight Tying（权重绑定）

- **原理**：Weight Tying是一种将词嵌入层和softmax层的权重共享的技术。这种方法可以大幅减少模型的参数数量，并防止模型在输入和输出之间学习一对一的对应关系。
- **效果**：权重绑定减少了模型参数，降低了过拟合的风险，并提高了LSTM语言模型的性能。

4. Activation Regularization（激活正则化，AR）

- **原理**：AR是一种对激活函数进行的正则化，不同于对权重进行的L2正则化，AR惩罚那些激活值显著大于0的激活函数。
- **效果**：通过对激活函数进行正则化，可以防止模型过度激活特定特征，从而减少模型的过拟合风险。

5. Temporal Activation Regularization（时间激活正则化，TAR）

- **原理**：TAR是一种对RNNs进行"慢速性"正则化的方法，它通过惩罚模型在隐藏状态上的大幅度变化来实现。具体公式为：

  $$TAR = \beta \cdot L2(h_t - h_{t+1})$$

  其中，$L2$ 是L2范数，h_t 是RNN在时间步t的输出，β 是缩放系数。
- **效果**：TAR减少了时间步之间的激活变化，从而提高了模型的平稳性和泛化能力。

四 Transformer 英中翻译任务+VIT图像分类任务

11785课程中有额外两个lab文件，学习如何基于transformer架构实现文本翻译和图像分类。代码文件在Transformer文件夹中，原本的翻译是英德翻译，我改造成英中翻译了，绘制了详细的transformer架构图和英中翻译消融实验记录等。

翻译任务跟文本预测任务差不多，只是翻译任务每句话需要有个结束符，表示一句话翻译结束了。

翻译和图像分类的原理不阐述了，具体见代码和draw.io架构图。