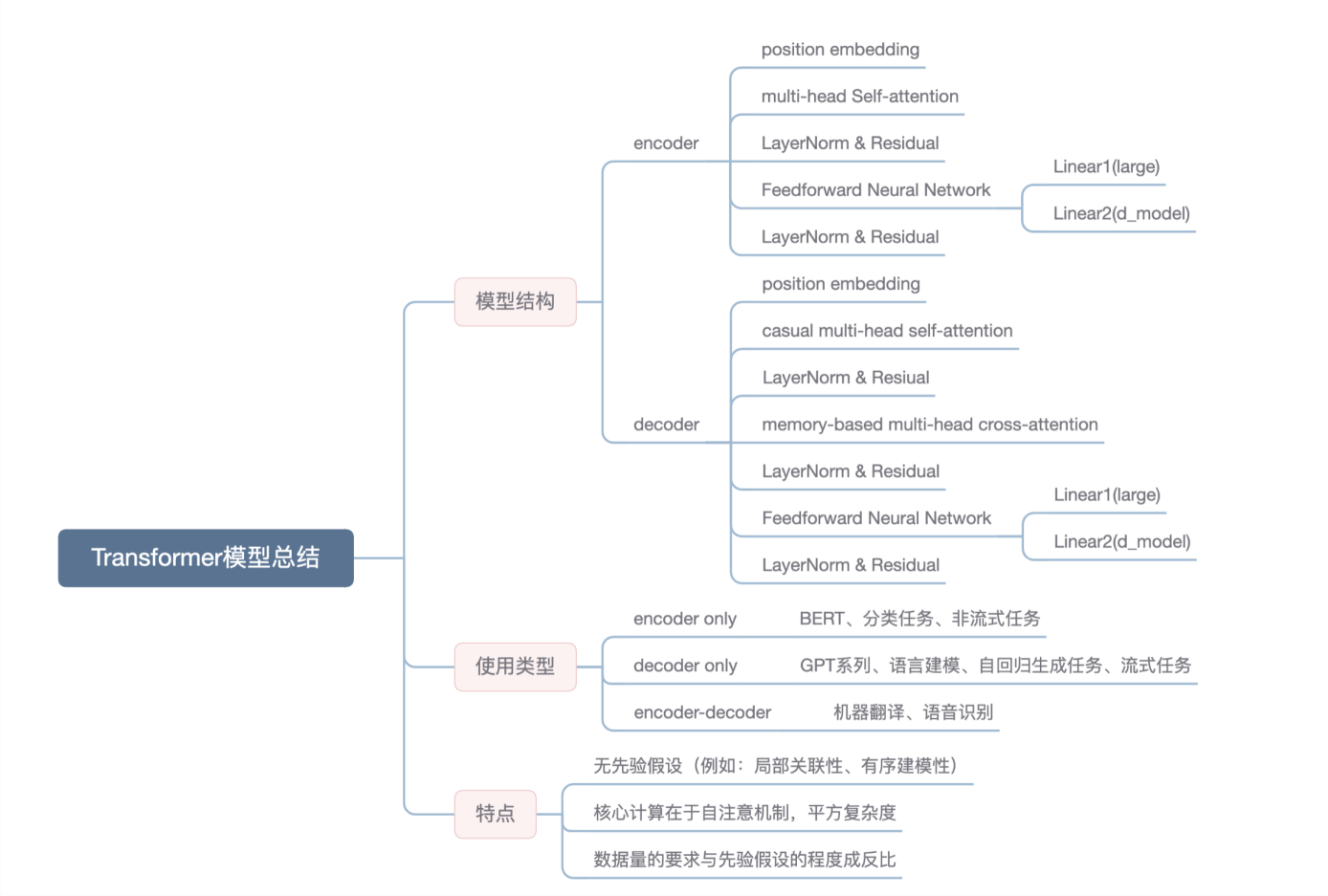
时间线：

- 241026 开始
- 241027 下午14.19 done！

topic：Transformer模型总结 & maksed loss的实现



首先总结一下Transformer的模型，一个是模型结构，一个是使用类型，一个是特点；

无先验假设（例如：局部关联性、有序建模性）

首先来看特点，首先Transformer和CNN、RNN的最大区别就是先验假设、归纳偏置都比较少，没有假设局部关联性，也没有假设要进行有序建模，它的假设是任意一个位置都可以与其他位置有关联性，任意一个位置的表征 都可以与其余位置是有关的，是一个很宽泛的或者说基本上就是说没有先验假设，这是一个好处也是一个特点 好处就是说

相比于CNN和RNN而言，它可以更快速的去学到无论是长时的建模性 还是短时的关联性 都能够很好的 学到，这是先验假设比较少的优点，那么缺点是什么呢?

缺点就是第三点，就是数据量的要求它是跟先验假设的要求成反比的，其实也就是说 我们看到每一个模型的时候 我们都需要首先想一下 这个模型之所以提出来 是否有一个 归纳偏置 或者说 是否有一个先验假设

比如说 卷积网络 它的先验假设是什么呢？它就是认为 我们可以通过局部的像素点 能够学好 相当于有一个局部关联性的假设

而循环神经网络呢？循环神经网络的假设是我们的数据 是要进行有序地建模 就是先后顺序的 当前位置的输出 必须是通过 过去的上一时刻的输出 来进行建模，所以RNN是一个条件自回归的一个建模 这是它的假设

其实 先验假设越多的话 就是说我们人为地注入更多我们的经验 这样的话 模型就更容易去学 换句话说 对数据量的要求 就越低 那如果是一个很小很小的数据 上来就用Transformer模型去搞的话 那这样的话 就很难去学好 为什么呢？

因为Transformer模型 它里面并没有注入什么 人类对任务的先验假设 所以这就是第一点无先验假设 其实和第三点 数据量的要求 跟先验假设的程度 成反比 这两个其实是一个意思

所以说Transformer模型 虽然Transformer模型 现在很火 但是Transformer模型也不能无脑用 因为先验假设很少 如果要用好的话 还得根据特定的任务 注入任务相关的 先验假设；比如说 注意力机制上、loss上 或者说结构上 注入先验假设，根据先验假设 来去做一些改变 或者说 优化，那这是一个特点；

另外一个特点就是说 Transformer模型 它的核心就在于 自注意力机制，根据自注意力机制 或者说 scaled Multihead Attention这种机制 来对序列的每一个位置 进行一个建模

核心计算在于自注意机制，平方复杂度

原本的自注意力机制 它有一个特点就是它随着序列的长度 序列越长 计算的复杂度 是呈平方的一个关系 在增长的 我们在循环神经网络中 它虽然是有序建模 每一次 都是递归的去算 但是它每一次去算 它的计算量是 固定的 这是循环神经网络的一个特点 每一个单步的运算量 是固定的 对于Transformer而言 你去算一个 长度为10的句子 与 去算一个长度为20的句子 这两个计算的复杂度 是不一样的 是跟序列长度的平方成比例的

这个是 Transformer模型 如果是 对于很长很长的序列的话 它的计算的瓶颈是在这里的 它跟序列长度的平方成正比的 而不是 线性比例的

所以 后来也有很多工作 是降低 注意力机制的一个复杂度

之所以能够降低复杂度 如果从算法上去降低 或者从模型上去降低的话 那么必须要注入一些先验假设 这是必然的 你要想 我们去算 注意力机制 不是对所有位置都算的话 那需要注入先验假设

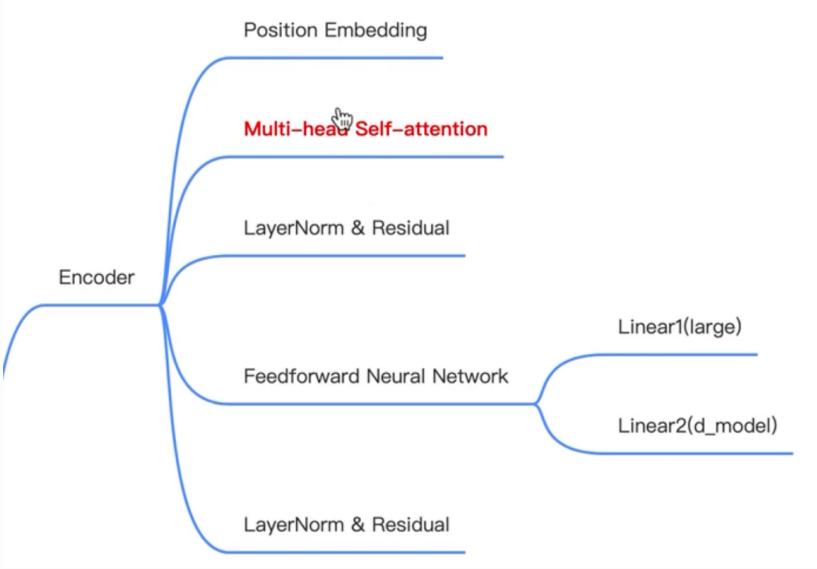
比5如说，你认为 每个位置 注意力机制的计算 不需要 针对 整个序列 而是针对这个序列的周围的几个token，那这个假设是什么呢？这个假设其实就是 局部关联性或者说 单调性，这些假设 你注入进来，那么自注意力的计算复杂度，都可以降低。以上是 Transformer的特点。

接下来，开始Transformer模型的总结

首先 Transformer模型是一个 seq2seq的 序列建模模型，大概在 2013和2014年的时候，seq2seq是通过 循环神经网络 去构建的，也就是说 不论是 encoder 还是 decoder 还是 Attention 都是由RNN 或者LSTM 或者是GRU 构建的 。那RNN的缺点 它是由 递归 去计算每一步的，所以长时的建模性 就不那么强，这是RNN的一个缺点 并且 RNN是串行计算的 它不能去 并行运算。

也就是说 RNN虽然 单步计算量是恒定的 ，但是整个序列 是基于 顺序的去计算，它无法并行去算，那Transformer模型 就有效的解决了 这两个缺点

一个是长时建模性 不强的一个缺点。另外一个是 必须串行运算的一个缺点，那Transformer模型 一方面 通过无先验假设的 Self Attention能够使得 我们 既可以 进行短程的建模 也可以做序列的长程 建模。另外 Transformer完全是一个DNN结构 所构成的，可以很好的去进行并行 (?)



Transformer主要包含 encoder 和 decoder两部分，那在Transformer中 encoder，主要由以上几个部分 构成；第一个是 position embedding，为什么要有position embedding，因为Transformer模型 本质上是一个dnn的结构，整个运算是没有考虑到位置信息的，换句话说就是 如果把 直播 换成 播直，得到的上下文表征是一样的，那这样就很不合理 因为我们要进行 序列建模，那进行序列建模的话 序列之间 token的相对顺序和绝对顺序 还是很重要的 所以我们会注入一个position embedding

当然也有人把 Transformer应用到无序建模

有些任务跟顺序 没那么重要 可以认为是一个集合，对一个集合的token 进行建模。也有这样论文发表的。

这是第一块：position embedding的讲解

接下来第二块：

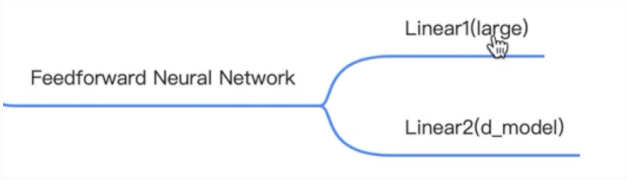
这个是 整个 Transformer的一个核心 也是它的计算量 最大的一块；涉及到的论文 也是很多很多的

mutihead self attention之前已经讲了很多了

总之计算任意两个token之间的相关性；

跟着multihead Self attention之后 是层归一化和残差连接

也就是 首先input embedding输入进来，首先 对这个embedding加上一个position embedding，接着对含有位置信息的词嵌入 计算 multihead self Attention，得到一个新的序列，再把这个序列进行层归一化 和，然后再把输入 也就是 总的embedding 与 层归一化的输出 相加，以上就是残差连接怎么做的，论文中 多头设置的是8，紧接着是FNN，FNN结构 跟 multihead self Attention最大的区别是什么？就是为什么后面要再加一个FNN，类比卷积，首先 有一种deep-wise的卷积 也就是 通道分离的卷积，然后紧接着 就是1×1的卷积，那通道分离的卷积主要做的是什么？主要做的是 像素 也就是 空间上的 混合；然后1×1的卷积，主要做的是 通道上的一个混合，那在Transformer中，multihead self Attention跟feed forward Neural network的关系是类似的，self Attention 做的是位置上的一个混合，就是会算一个权重，然后把value的每个位置上 进行一个加权求和 就是对位置上的一个混合，FNN是对特征层进行一个混合，对每个位置上的一个特征维度 进行一个mix 操作，所以



MHA和FFN作用是不一样的，MHA是对序列的每个位置进行一个混合，FFN是对每个位置上的 特征的维度 进行混合，也可以说 每一个小特征进行混合，两个作用是不一样的，所以必须要有一个FFN「以上回答了 为什么设置FFN模块」

FFN在原始论文中是分为两层的，第一层是比较大一点，large=2048，第二层会小一点 设置成512 ，这些没有那么重要 是超参数，超参数怎么设置 需要自己去调；

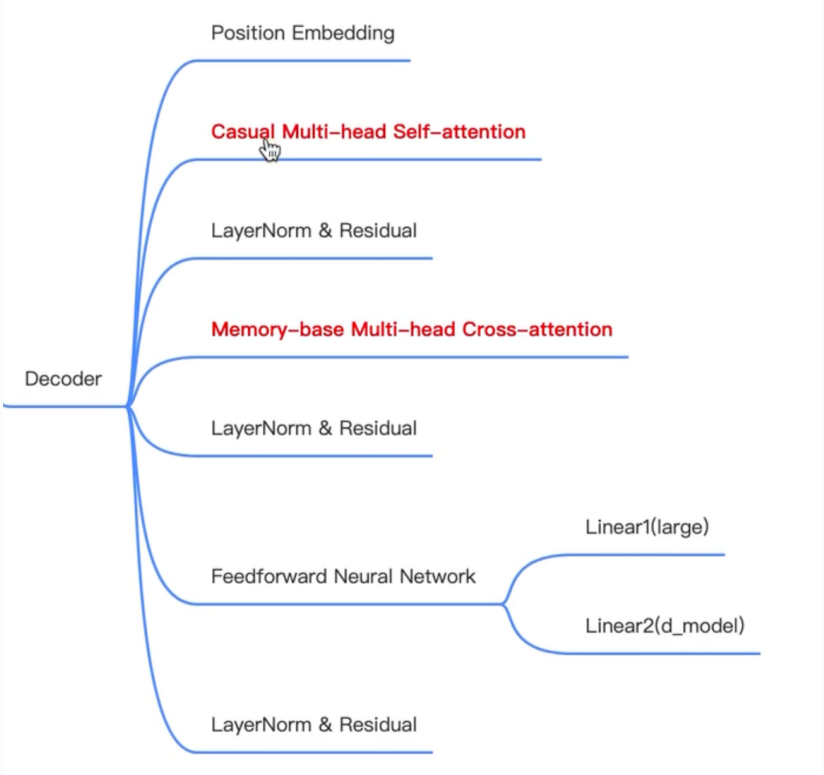
跟随着FNN，同样也是层归一化和残差连接；这个残差连接 就是跟FNN的输入加起来

以上 得到了 第一个 encoder block的输出，然后 我们把 很多个block 堆叠起来

以上是 encoder

接下来 讲解 encoder和Transformer会怎么用？其实Transformer的很多论文 并不是encoder和decoder一起去用，仅仅针对encoder 也可以用，比如说 那些模型 是encoder only的模型呢？所谓的encoder only 就是说 这个模型仅仅是由Transformer encoder构成的，没有cross Attention，也没有decoder部分。像BERT，就是仅仅用到了Transformer encoder；比如说 分类任务，一个句子 对情感进行判断，也是encoder only的模型，还有比如 非流式任务，也就是说直进直出，而不是 每次返回一小部分，非流式任务也可以用encoder only的模型；

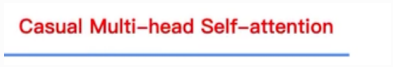
以上是encoder only的应用。接下来decoder的结构



decoder 的每个block是多了一个casual multi head self Attention以及memory based multihead cross Attention；

首先decoder 的部分 也会有 token embedding的部分，比如在机器翻译中，encoder的输入可能是中文，decoder的部分 可能的输入是 英文，得到英文的embedding，同样英文的embedding 也需要 position embedding，因为 我们的目标序列 也是需要 考虑位置的，我们首先把 word embedding和position embedding加起来构成一个embedding，再把这个embedding，输入到casual multihead self Attention，所谓casual 就是一个因

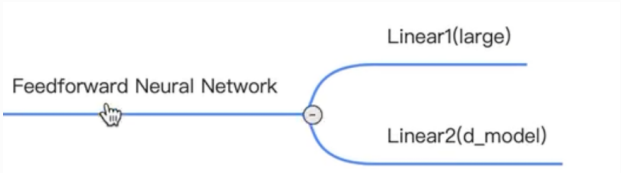
果的 多头自回归，这个因果体现在哪里呢?



体现在 我们在预测第t步的输出的时候，我们是decoder的输入 只能看到之前的字符 这就是因果性，因果性 数学上 我们用mask表示，用一个三角的mask，来对score进行一个处理，并且在训练阶段，我们喂入的输入是真的target embedding，但是在推理阶段，我们每次输入到decoder的是上一步预测的embedding。对于自回归模型，训练和推理是有一定差别的，

因果的Attention之后，同样跟随着一个 层归一化和残差连接；

接下来，memory based multihead cross Attention ，这个就是说 通过decoder上一层的输出，作为query，encoder的输出 memory作为value和key，计算Attention的表征，算法和原理就是multihead self Attention，没有什么新的东西 只不过这里的QKV 不是由同一个量 所生成的，那在cross Attention之后，也是层归一化和残差连接，也就是说需要把注意力机制的输入跟层归一化的输出加起来，这是第二层；



那第三层呢，仍然是一个FNN的一个网络，刚刚也说了FNN和MHA的区别是，MHA是在位置上的一个混合，而FNN是在特征上的一个混合，它们俩的关系就像之前说的可分离卷积和1×1的卷积，它们俩的组合一个是做像素点上的混合，一个做通道上的一个混合，它们两个组合起来 一个是使得运算量更小 又能实现 普通卷积的一个效果，FNN之后，跟随的是 层归一化和残差连接，以上是decoder的结构

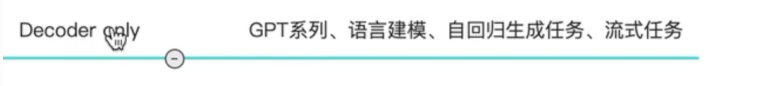
decoder的特点，第一个 对于Transformer而言，有一个因果的multihead self Attention，第二个就是 有一个Cross Attention，是encoder的输出 作为key和memory（value）的 这样的 一个 multihead self Attention

训练和测试的区别

比如说 中英文的翻译，训练的时候 decoder喂入的是 真实的 英文 target；但是 如果 给一段 中文，我们对模型进行一个测试的话，那decoder 输入的embedding 就是上一步预测出来的 概率最大的英文字符；

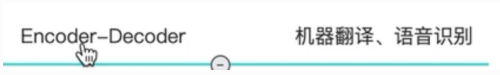
重复：就是说训练的时候 喂入的是真实的 target；测试的时候 喂入的是 上一时刻 预测的 embedding，这就是区别

这种训练自回归的方法 叫做 teacher force；teacher force training；



也有一些 模型 只用到了 Transformer decoder

比方说GPT系列，所谓decoder only的结果 就是在计算 multihead self Attention的时候需要一个因果的掩码，还有语言建模、自回归的生成任务、流式任务 都需要decoder only的建模



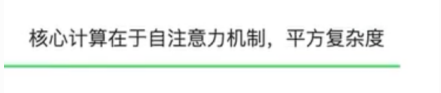
也有任务 用到 encoder-decoder 整个Transformer 结构去用，比如机器翻译和语音识别，encoder部分我们输入的是语音，decoder部分是我们预测出来的汉字，而且语音识别的一个特点是什么呢？就是encoder部分 的输入是流式的，这是语音识别的一个不同，因为我们人说话，如果我们人说完了 再返回结果。体验感很差，就是说 我们边说的时候 decoder 边解码，就是说 我们encoder一边接收输入一边inference，这也是一个非常复杂的问题；

第一部分：以上是所有关于Transformer 模型的总结

关于Transformer的变体

一方面是只用encoder 或者 只用decoder 就是我们上面说的

还有就是，加入先验假设 比如在注意力机制中 认为注意力权重不是随机的 可能是对角的，或者做哈希，总之就是为了减低自注意力机制的平方复杂度



在后面 我们在讲hugging face Transformer库的时候，可以看到很多Transformer模型的变体，后面会阅读hugging face的源码 以及 论文

第二部分：masked loss代码演示

以机器翻译为例，计算loss，机器翻译本质上是一个 分类任务

decoder 计算 概率，再跟目标的标签 计算一个交叉熵loss，所以我们可以假设，通过模型预测出来的定义成logits,假设 batch size=2，sequence length=3，单词表的数目=4，我们可以通过Transformer输出这样的shape的logits

使用torch必备的三个库 无脑导入

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

假设Transformer预测出来的概率 张量 通过pytorch 生成 高斯分布 随机数

```
logits = torch.randn(2,3,4)
# batch size=2, sequence len=3, vocab_size=4 单词表大小
```

接下来 我们还需要生成一个Label 用来 计算 loss，Label 应该是一个 整型的标签，最小值应该是0 ， 最大值 是vocab size是4 ； size应该是batch size×sequence length × vocab size，因为一般情况下 我们的Label是 某一个单词，但也有时候 我们的Label 是一个概率，pytorch中也是支持的

我们演示用的第一种情况 标签是 单词

```
label = torch.randint(0,4,(2,3))
```

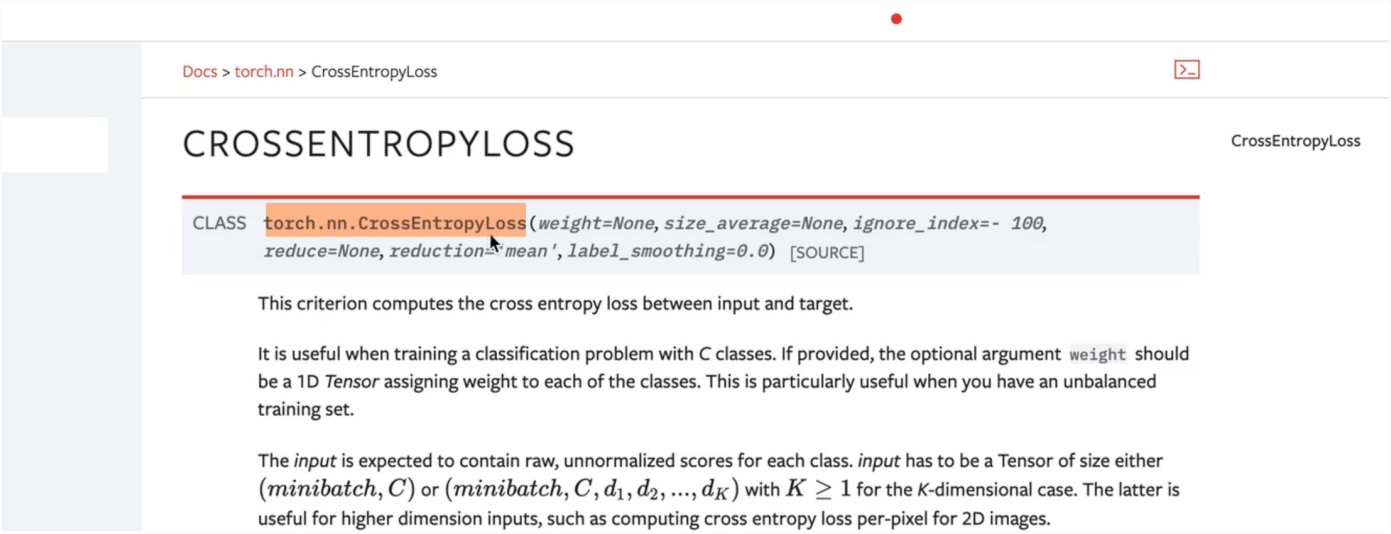
这个意思就是说 我们对每一个样本的 每一个位置上 都有一个 word的Label，这个Label 就是 这个单词 在 单词表中的索引

有了Logits有了Label之后，就可以算 模型预测的 概率 跟真实标签之间的 交叉熵的 一个loss

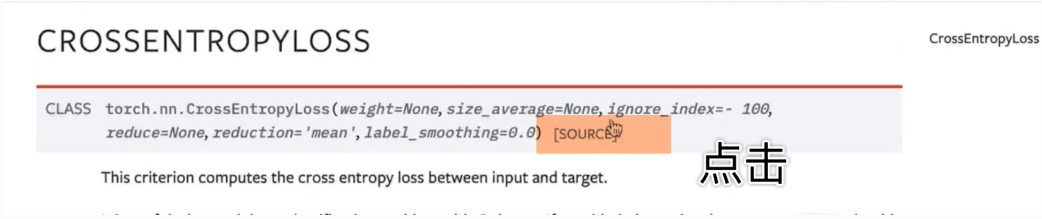
接下来 去看 在pytorch 中 怎么算 交叉熵 loss

搜索：

在pytorch1.10中 有一个类 叫做 `torch.nn.CrossEntropyLoss`



这是一个class，我们可以看一下 这个 class的源码：



可以看到：

crossEntropyLoss 其实 就是 一个包装

首先 init 中传入一些参数，然后赋值一下；

然后在forward中 调用的还是 `F.cross_entropy`

```
    def __init__(self, weight: Optional[Tensor] = None, size_average=None, ignore_index: int = -100,
                 reduce=None, reduction: str = 'mean', label_smoothing: float = 0.0) -> None:
        super(CrossEntropyLoss, self).__init__(weight, size_average, reduce, reduction)
        self.ignore_index = ignore_index
        self.label_smoothing = label_smoothing

    def forward(self, input: Tensor, target: Tensor) -> Tensor:
        return F.cross_entropy(input, target, weight=self.weight,
                               ignore_index=self.ignore_index, reduction=self.reduction,
                               label_smoothing=self.label_smoothing)
```

这个class 还是一个包装而已，其实 最终调用的还是一个函数，这个函数 就是 torch.nn.functional 就是F.cross_entropy

接下来 返回api,来看 里面的参数 是什么意思

CROSSENTROPYLOSS

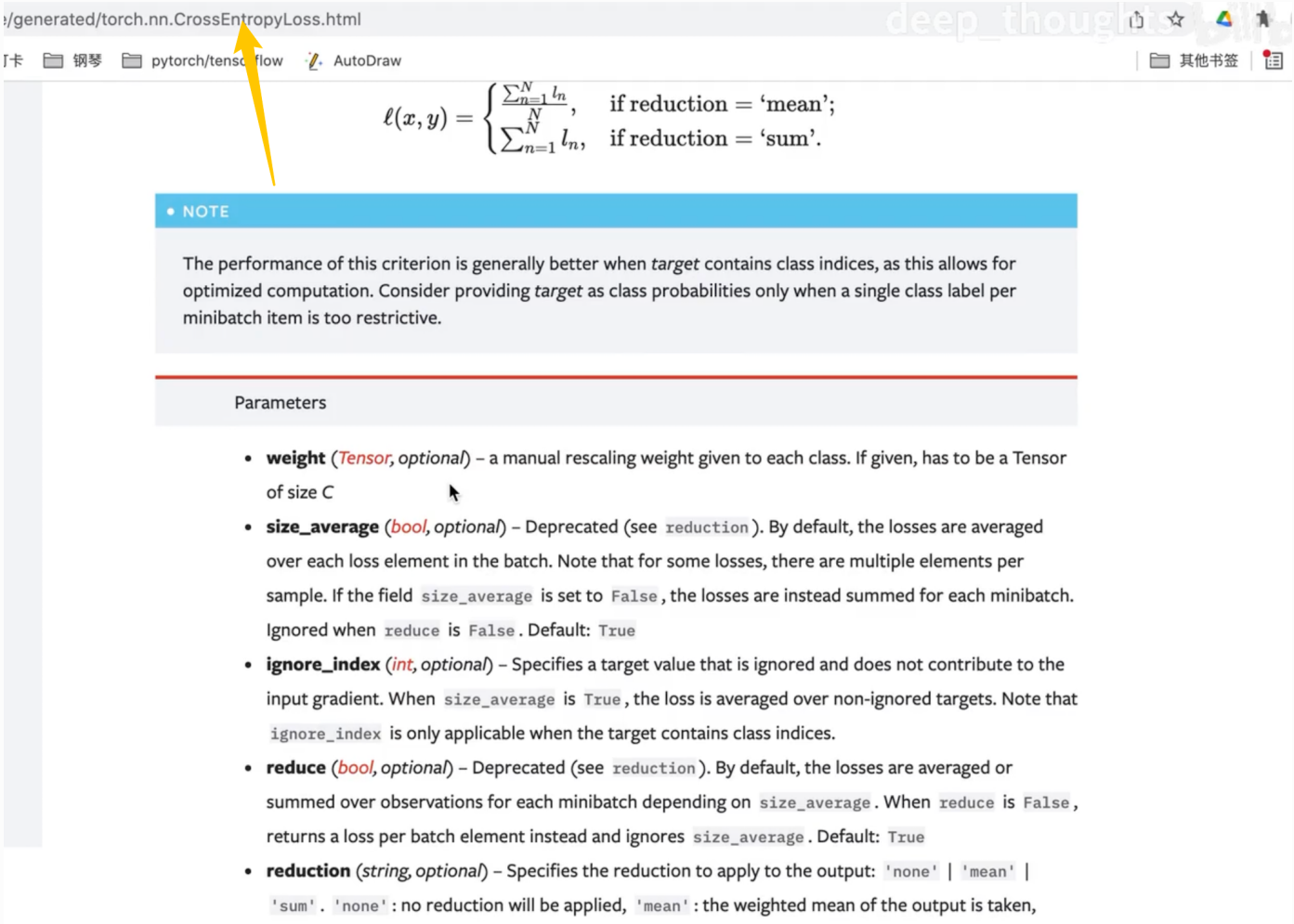
CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=- 100, reduce=None, reduction='mean', label_smoothing=0.0)` [SOURCE]

This criterion computes the cross entropy loss between input and target.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

首先：



weight就是 如果 数据不均匀的话 可以对 不同的标签 做一个 不同的 权重，这样来弥补 数据不平衡的缺陷，然后 还有两个 比较 重要的参数 一个是 `ignore_index`，这个和之前 讲的mask很相关，还有就是这个 `reduction` 这个参数，reduction 参数 有三种： none、meaning、sum,等下 会演示 区别

首先来用下 原版的softmax 函数，还得看一下 forward 函数 ，来看是怎么调用；

The *input* is expected to contain raw, unnormalized scores for each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C-1]$ where C is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}$$

首先 传入crossEntropyloss forward函数中 两个参数，一个是input 一个是 target

input 应该是一个 未归一化的 score，也就是说我们最后一层 全连接出来的 输出就可以，不需要做softmax；

就是未归一化的score，然后 input的形状，在pytorch中，我们一定要注意形状，因为这个形状 可能有时候 并不常见，可以看到 要求这个维度 要没事 batch size×C，要么是 batch size × C × d1 ....× dk

training set.

The *input* is expected to contain raw, unnormalized scores for each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

换句话说 就是我们刚刚声明 的 logits 的 tensor ，我们认为是 batch size× sequence length×vocab size，但是 在pytorch中 它希望我们把class 这个维度 放在 第二维，所以我们需要 对 logits 进行转置

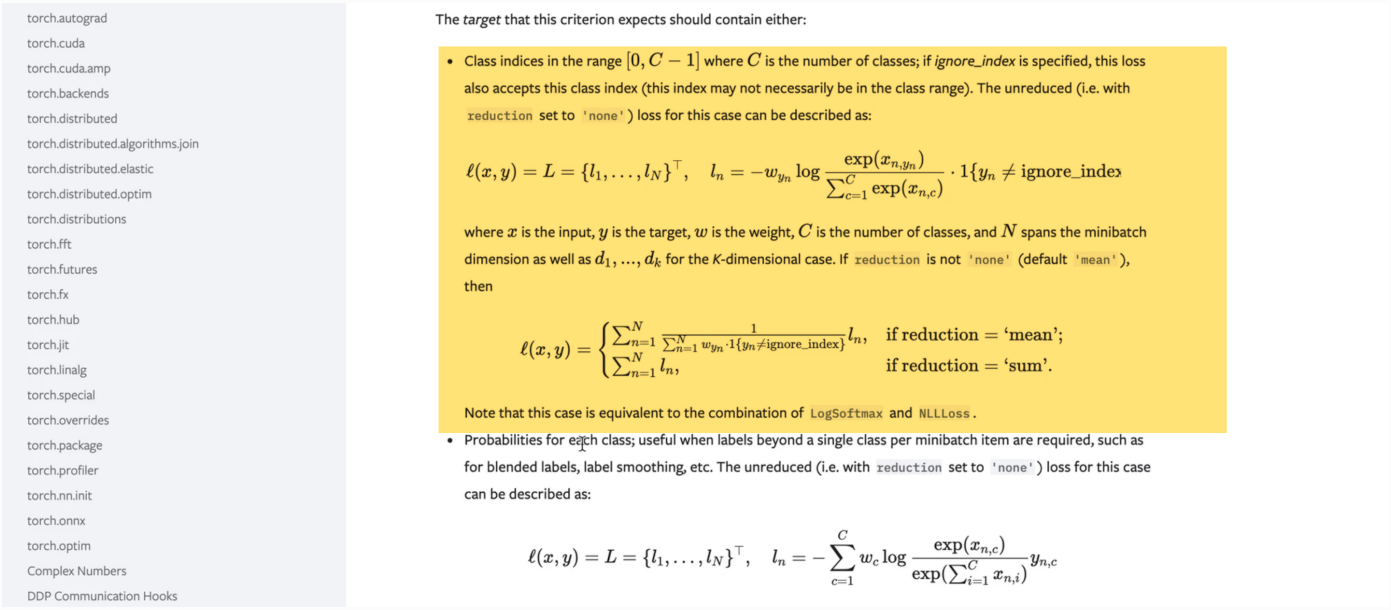
```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> logits=torch.randn(2,3,4)
>>> # batchsize=2,seqlen=3,vocab_size=4
>>> label=torch.randint(0,4,(2,3))
>>> logits=logits.transpose(1,2)
```

就是说 我们本来假设的是 中间是sequence length，但是在pytorch 中 希望 中间 这个维度是 vocab size，所以我们需要把 第1维和第2维转置一下，这边写12或者2 1都可以，transpose这里的 两个顺序 是 无关的，这样 我们把logits 写成了pytorch 官网 所要求的形式

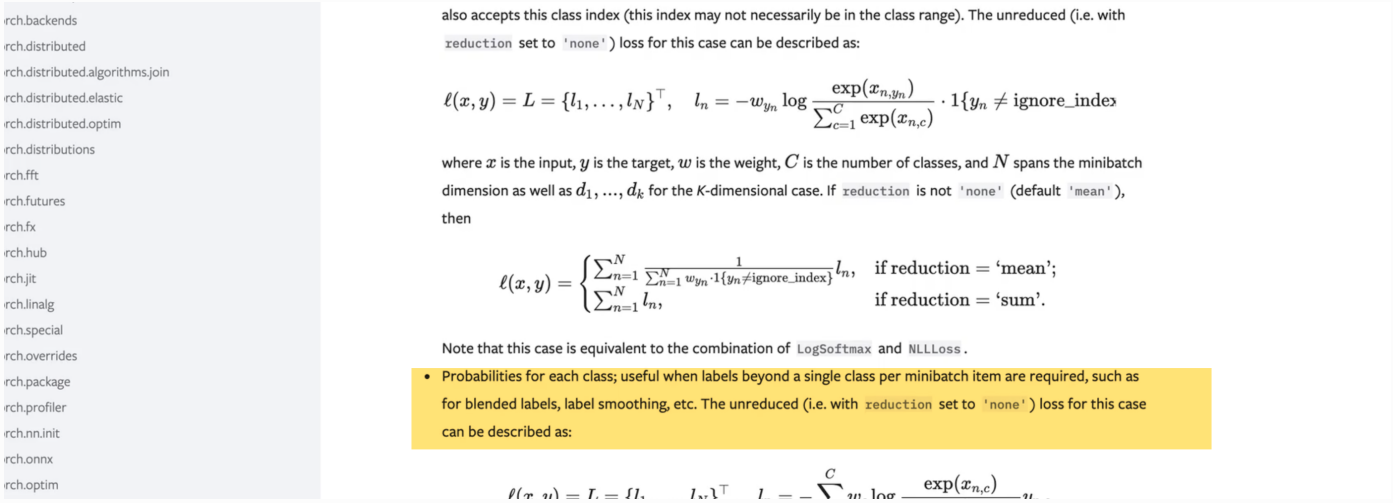
```
import torch
import torch.nn as nn
import torch.nn.functional as F

logits = torch.randn(2,3,4)
label = torch.randint(0,4,(2,3))
logits = logits.transpose(1,2)
```

接下来 我们来看 target 有什么要求，



target要么是 class indices 是Label 的一个 整型的索引，比如说 猫 是第一类，狗是 第二类



另外也可以传入是概率，如果传入是概率的话，那么target 就应该 跟input的形状是 一样的

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> logits=torch.randn(2,3,4)
>>> # batchsize=2,seqlen=3,vocab_size=4
>>> label=torch.randint(0,4,(2,3))
>>> logits=logits.transpose(1,2)
>>>
```

但我们 这里 传入的是 int的 也就是说 我们 传入的是 indices,然后 我们就可以算了

因为我们刚刚看了源码，也是调用的F.cross_entropy，所以我们不实例化了

```
        super(CrossEntropyLoss, self).__init__(weight, size_average, reduce, reduction)
        self.ignore_index = ignore_index
        self.label_smoothing = label_smoothing

    def forward(self, input: Tensor, target: Tensor) -> Tensor:
        return F.cross_entropy(input, target, weight=self.weight,
                               ignore_index=self.ignore_index, reduction=self.reduction,
                               label_smoothing=self.label_smoothing)
```

我们看到 第一个位置上是input，所以我们把logits放到 第一个位置，第二个位置上是Label

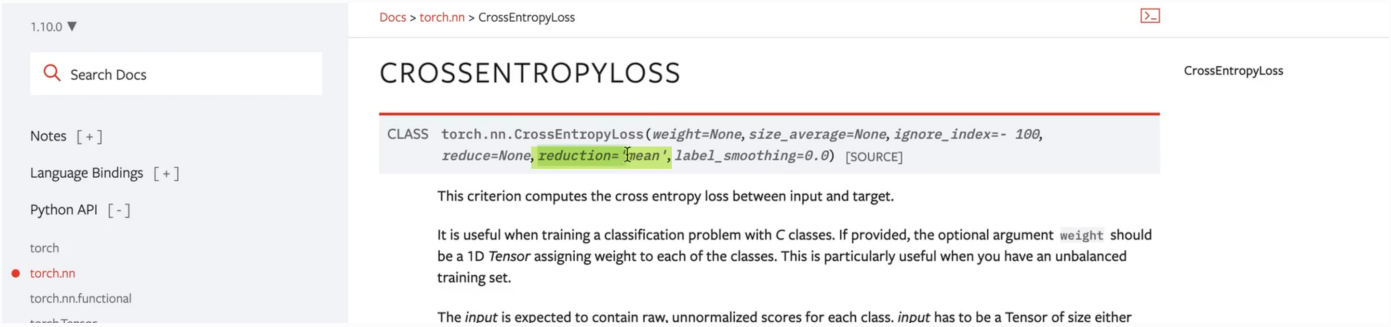
```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> logits=torch.randn(2,3,4)
>>> # batchsize=2,seqlen=3,vocab_size=4
>>> label=torch.randint(0,4,(2,3))
>>> logits=logits.transpose(1,2)
>>> F.cross_entropy(logits, label)
tensor(2.1753)
>>>
```

这样 我们算出了 logits和Label之间的 loss，看到loss 返回的是 2.1753

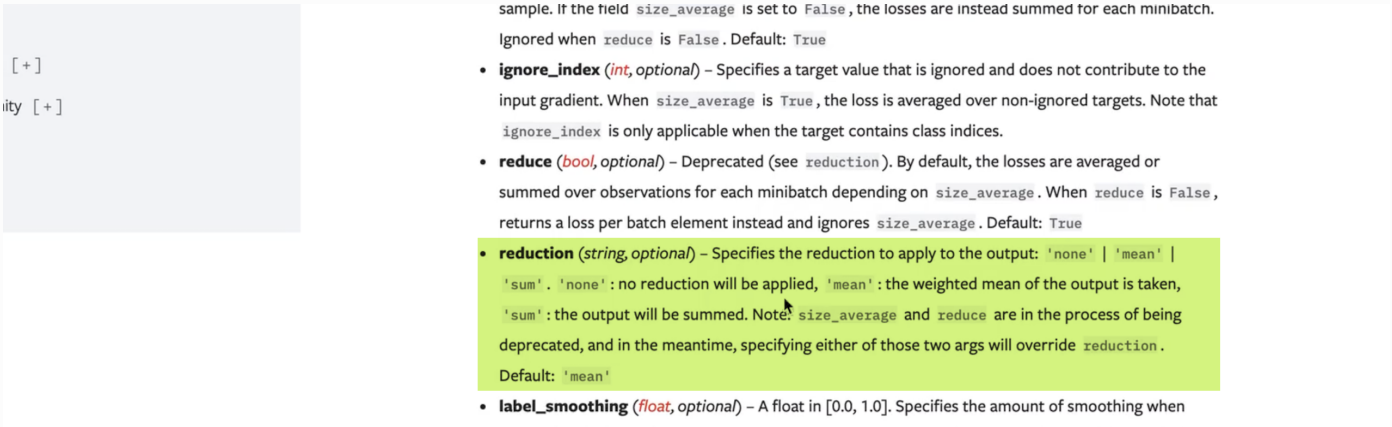
首先 2.1753是一个标量 之前在 自动微分中 讲过 ， 在做后向 梯度 传播的时候，最顶端 是有一个标量的，通过标量 找到 附近链 的梯度。我们这里的 2.1753是怎么得到的呢?

首先 我们这里输出logits 是2×3×4的维度，相当于 就是说 如果以 单词为一个单位的话，那就是有6个单词，因为有2个序列，每个序列有3个单词,所以一共6个单词

而每个单词 都是一个 分类任务，首先计算每个单词的交叉熵，然后我们再把 6个单词 求平均 或者求和，所以我们来看一下 交叉熵 函数 默认是什么，也是这个reduction



reduction默认是mean，也就是说，这个2.1753是6个单词 平均的 交叉熵，那如果我们不想得到平均，我们想得到 原本的6个交叉熵，那我们在reduction这里 设置一下，下拉 查看设置：



reduction不用mean，用none，查看 输出

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> logits=torch.randn(2,3,4)
>>> # batchsize=2,seqlen=3,vocab_size=4
>>> label=torch.randint(0,4,(2,3))
>>> logits=logits.transpose(1,2)
>>> F.cross_entropy(logits, label)
tensor(2.1753)
>>> F.cross_entropy(logits, label, reduction='none')
tensor([[1.8851, 2.4212, 1.9554],
        [0.9814, 2.0455, 3.7629]])
>>>
```

可以看到 reduction写成none的话，就会把所有单词的交叉熵 都返回出来，这样我们得到2×3的张量

每个位置上 都是相应单词的 预测的概率 跟它标签之间的一个 交叉熵的损失值

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

logits = torch.randn(2,3,4)
label = torch.randint(0,4,(2,3))
logits = logits.transpose(1,2)
print(F.cross_entropy(logits,label))
print(F.cross_entropy(logits,label,reduction='none'))
```

有了这样一个 原本的结果之后，我们对它进行mask，也就说 加入 我们的target len不都是33的长度，如果都是33的长度 我们不需要mask，因为每个位置上都是有效的单词，现在我们假设 target len分别是23，

```
tgt_len = torch.Tensor([2,3]).to(torch.int32)
```

也就是说 batch size=2，第一个句子的长度是2，第二个句子的长度是3,所以第一个句子的最后一个位置 预测的值 应该 mask掉，所以要生成一个 mask矩阵，这个mask矩阵，在tensorflow中有api，但是在pytorch中好像没有，不过自己手写也不难，具体地写法：

首先对 target len进行一个遍历，这样得到每个样本的序列长度 `for L in tgt_len:`

然后 我们不要mask的位置 是一个 全1的张量: `[torch.ones(L) for L in tgt_len]`

查看一下输出：

```
>>> tgt_len = torch.Tensor([2,3]).to(torch.int32)
>>> [torch.ones(L) for L in tgt_len]
[tensor([1., 1.]), tensor([1., 1., 1.])]
```

得到一个 一维的张量，第一个张量是11，第二个张量是111，然后 我们要做成 一个mask矩阵的话 需要保证 长度是一样的，所以需要调用一个pad函数，同时将其扩展成 一个 二维的张量，这样才能把它们 concat起来，所以我们的mask矩阵 应该怎么做呢？

首先我们要对它做一个pad,F.其中 左边不用pad，右边pad，pad几个单位呢？就是max(tgt_len)-L，就是最大长度 减去 自身长度个单位，这个就是pad的数量

```
mask = [F.pad(torch.ones(L),(0,max(tgt_len)-L)) for L in
tgt_len]
```

最后呢 对它 进行一个扩维 调用 unsqueeze函数，对这个张量 进行一个 扩维，扩成一个二维张量

```
>>> mask = [torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len]
>>> mask
[tensor([[1., 1., 0.]]), tensor([[1., 1., 1.]])]
>>>
```

torch.cat进行拼接，得到一个mask矩阵：

```
>>> mask = torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len])
>>> mask
tensor([[1., 1., 0.],
        [1., 1., 1.]])
>>>
```

可以看到 返回的mask矩阵，且这个返回的矩阵，跟上面二维交叉熵 返回的维度是一样的：

```
>>> F.cross_entropy(logits, label, reduction='none')
tensor([[1.8851, 2.4212, 1.9554],
        [0.9814, 2.0455, 3.7629]])
>>> tgt_len = torch.Tensor([2,3]).to(torch.int32)
>>> [torch.ones(L) for L in tgt_len]
[tensor([1., 1.]), tensor([1., 1., 1.])]
>>> mask = [torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len]
>>> mask
[tensor([[1., 1., 0.]]), tensor([[1., 1., 1.]])]
>>> mask = torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len
>>> mask
tensor([[1., 1., 0.],
        [1., 1., 1.]])
>>>
```

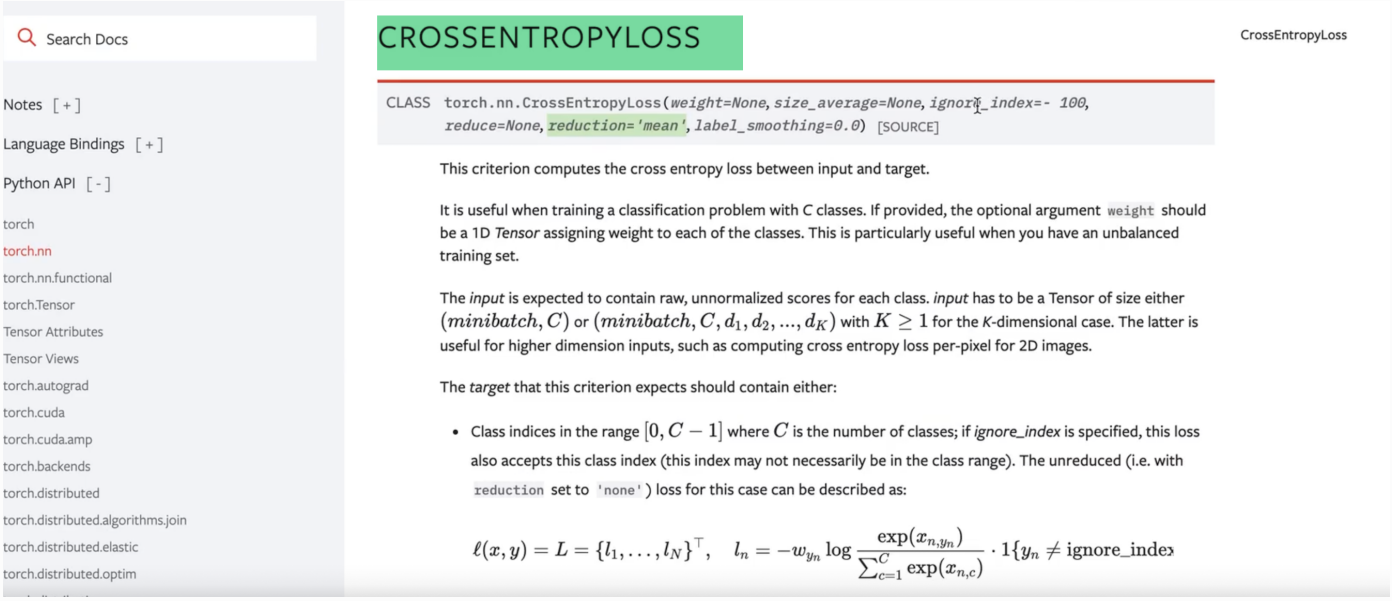
所以 我们可以直接把 交叉熵的结果，跟mask的结果 进行一个element-wise的一个相乘 就好了。（是不是 也叫 哈德玛积），这样就能得到被pad的位置上的单词 变成0

```
>>> F.cross_entropy(logits, label, reduction='none')
tensor([[1.8851, 2.4212, 1.9554],
        [0.9814, 2.0455, 3.7629]])
>>> tgt_len = torch.Tensor([2,3]).to(torch.int32)
>>> [torch.ones(L) for L in tgt_len]
[tensor([1., 1.]), tensor([1., 1., 1.])]
>>> mask = [torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len]
>>> mask
[tensor([[1., 1., 0.]]), tensor([[1., 1., 1.]])]
>>> mask = torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len])
>>> mask
tensor([[1., 1., 0.],
        [1., 1., 1.]])
>>> F.cross_entropy(logits, label, reduction='none') * mask
tensor([[1.8851, 2.4212, 0.0000],
        [0.9814, 2.0455, 3.7629]])
>>>
```

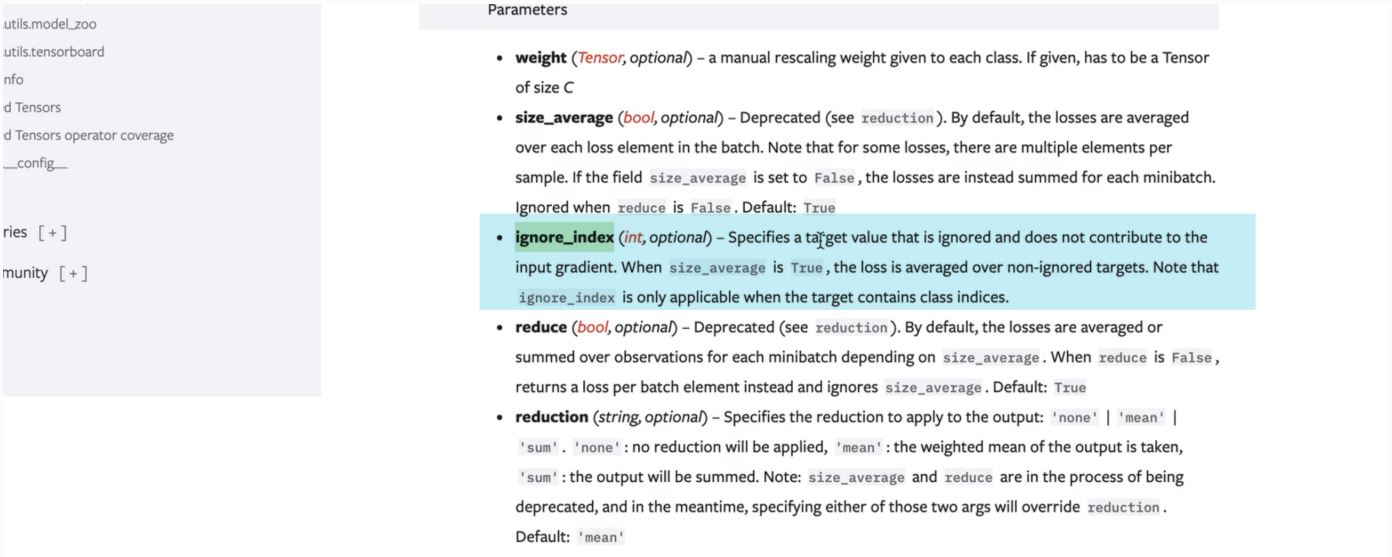
这也就是 我们的 loss mask

所以我们在进行序列建模的时候，在计算loss的时候，最好有一个loss mask

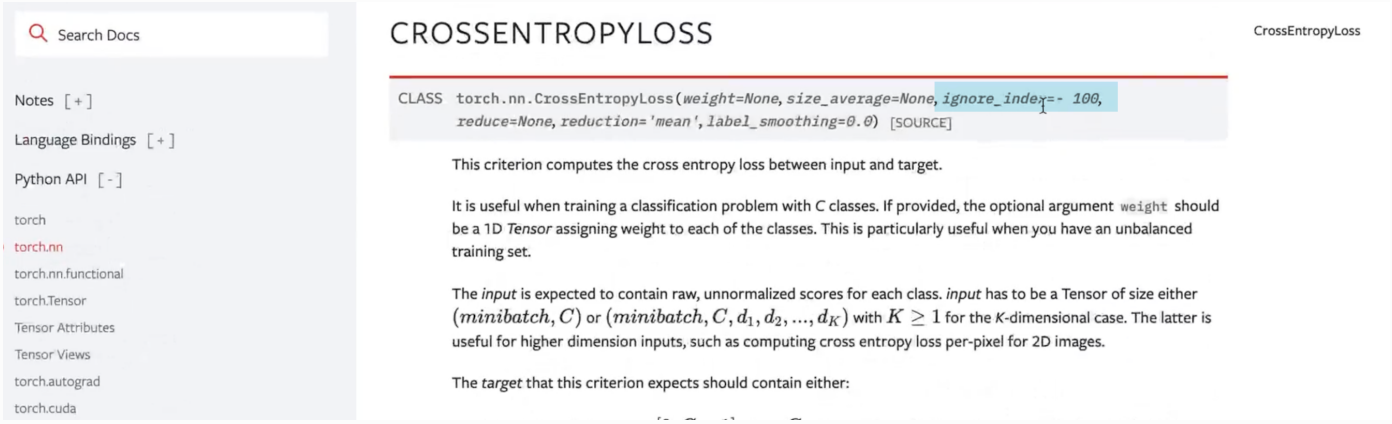
这是我们从原理的角度 完全手写的一个mask，其实在pytorch中，我们也可以 不手动写mask，我们来看一下这个api



这个 crossentropyLoss中 有一个参数，叫做ignore_index,我们来看一下 什么意思：



ignore_index就是说我们 需要，指定一个目标值，这个目标值 就是我们 标签索引，这个索引在pytorch中 是被忽略的，然后呢 也不会贡献梯度，其实就是我们做的 mask的意思，然后，看定义 这个 ignore_index的默认值 是-100



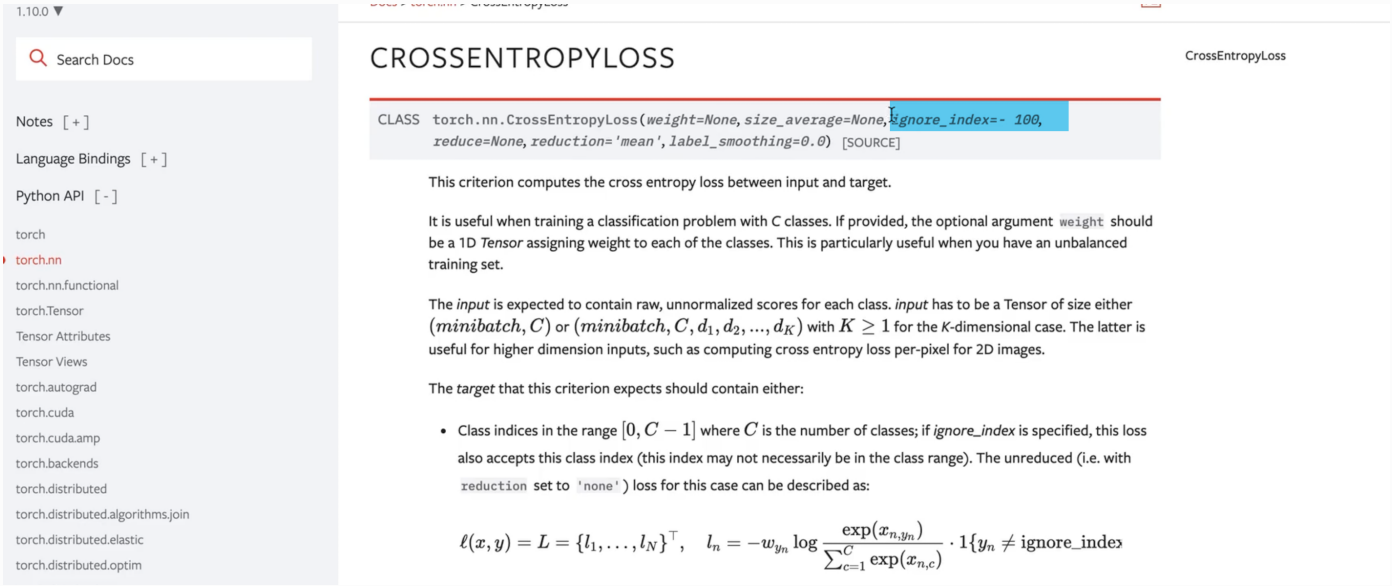
，换句话说，我们上面 定义了 一个Label，如果我们把这个 Label 的第0行的第二列 把 它置成 -100，

```
>>> label
tensor([[0, 1, 1],
        [1, 3, 1]])
>>> label[0, 2] = -100
>>> label
tensor([[   0,    1, -100],
        [   1,    3,    1]])
>>>
```

那这样的话，我们再调用 这个 cross_entropy 这个函数的话，也能实现一样的效果

```
>>> mask = [torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len]
>>> mask
[tensor([[1., 1., 0.]]), tensor([[1., 1., 1.]])]
>>> mask = torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) for L in tgt_len])
>>> mask
tensor([[1., 1., 0.],
        [1., 1., 1.]])
>>> F.cross_entropy(logits, label, reduction='none') * mask
tensor([[1.8851, 2.4212, 0.0000],
        [0.9814, 2.0455, 3.7629]])
>>> label
tensor([[0, 1, 1],
        [1, 3, 1]])
>>> label[0, 2] = -100
>>> label
tensor([[   0,    1, -100],
        [   1,    3,    1]])
>>> F.cross_entropy(logits, label, reduction='none')
tensor([[1.8851, 2.4212, 0.0000],
        [0.9814, 2.0455, 3.7629]])
>>>
```

，可以看到 (0,2)这个位置上 变成0了，所以我们用pytorch做序列建模的话，需要记住一点，pad的索引，传入到cross entropy Loss这里的 ignore_index就可以的，或者说



把pad的位置 pad成-100

只要理解 mask loss的原理，无论怎么变，都可以实现相同的效果

ok，结束，总结：

第一部分：Transformer模型的总结

第二部分：mask loss怎么构建，包括手动实现，包括如何调用pytorch的api实现类似的效果

全部代码：

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

logits = torch.randn(2,3,4)
label = torch.randint(0,4,(2,3))
logits = logits.transpose(1,2)

# reduction='mean'
print(F.cross_entropy(logits,label))
# out: tensor(1.5163)

# reduction='none' 会输出原本的 交叉熵
print(F.cross_entropy(logits,label,reduction='none'))
```

```
'''
tensor([[2.9297, 1.3549, 2.7817],
        [1.2506, 1.2661, 2.0864]])
'''

tgt_len = torch.Tensor([2,3]).to(torch.int32)

[torch.ones(L) for L in tgt_len]
# out: [tensor([1., 1.]), tensor([1., 1., 1.])]

mask = [torch.unsqueeze(F.pad(torch.ones(L),(0,max(tgt_len)-
L)),0) for L in tgt_len]
print(mask)
# [tensor([[1., 1., 0.]]), tensor([[1., 1., 1.]])]

mask = torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(tgt_len)-L)),0) for L in tgt_len])
print(mask)
'''
tensor([[1., 1., 0.],
        [1., 1., 1.]])
'''
print(F.cross_entropy(logits,label,reduction='none')* mask)
'''
tensor([[2.4503, 2.0125, 0.0000],
        [2.0472, 1.0315, 2.9729]])
'''
label
'''
tensor([[3, 2, 1],
        [3, 2, 3]])
'''
label[0,2]=-100
label
'''
tensor([[   0,    0, -100],
        [   3,    1,    1]])
'''
F.cross_entropy(logits,label,reduction='none')
'''
tensor([[1.7314, 1.9514, 0.0000],
        [1.7936, 2.3831, 2.5817]])
```

```
'''
```