Update:2021.11.14

今天要交作业了，回来看了一遍自己的实验报告（更像是一篇随笔吧）。这篇报告由于没有模板，我主要记录了我做实验遇到的问题以及在那个时间段找到的解决方案（有一些解决方案其实并不准确），为了记录自己的思维成长，我把当时的所思所想都写下来了。

由于本次实验我不是一口气做完的，中间有几个点卡住了导致有一段时间在查资料，所以我在最前面先总结一下我的实验过程：

本次实验我遇到的最大问题就是训练速度提不上来

我开始认为是GPU和CPU性能的差异，但在colab上用gpu实验后，我发现还是很慢

然后我查找lstm的底层实现，认为他一定使用了我们不知道的什么优化方法，发现他的运算是交给C++来实现的

最后还是感觉有问题，就在github搜了一下其他人的实现方式（借鉴借鉴），发现我的问题出在维度转换上。具体地说是 使用自定义的矩阵和输入相乘 与 使用layer层 的区别

至今还有两点疑问没有解决：

1. 自定义矩阵和输入相乘 与 layer层到底有什么区别？前者开始训练的时候损失比较大我能够理解（初始化的问题），但是为什么两者的训练时间也有差异呢？（前者3-4min/epoch，后者5-6s/epoch）
2. 在实现多层的时候，如果层数超过两层，那么这些添加的层的隐藏层（H与C）的权重共享么？我做了实验，发现没什么区别。。。、

若老师或学长对我的疑问能够提出宝贵的建议，请您联系我：QQ：2804272906 赵文昊

Update:2021.10.28

怎么说呢？在做实验的时候我的心路历程是：有点难 ==> 就这？ ==> 对不起 我错了

有点难

LSTM：一个听起来就很牛逼的名字，但是我们还是先捋清楚模型的计算过程：

每个时间点来一个输入Xt，每个单元保存一个隐藏单元Ht以及一个记忆单元Ct

1. 开始根据这些信息就可以计算输入参数、遗忘参数、输出参数、候选记忆单元

$$
\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned}
$$

$$
\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),
$$

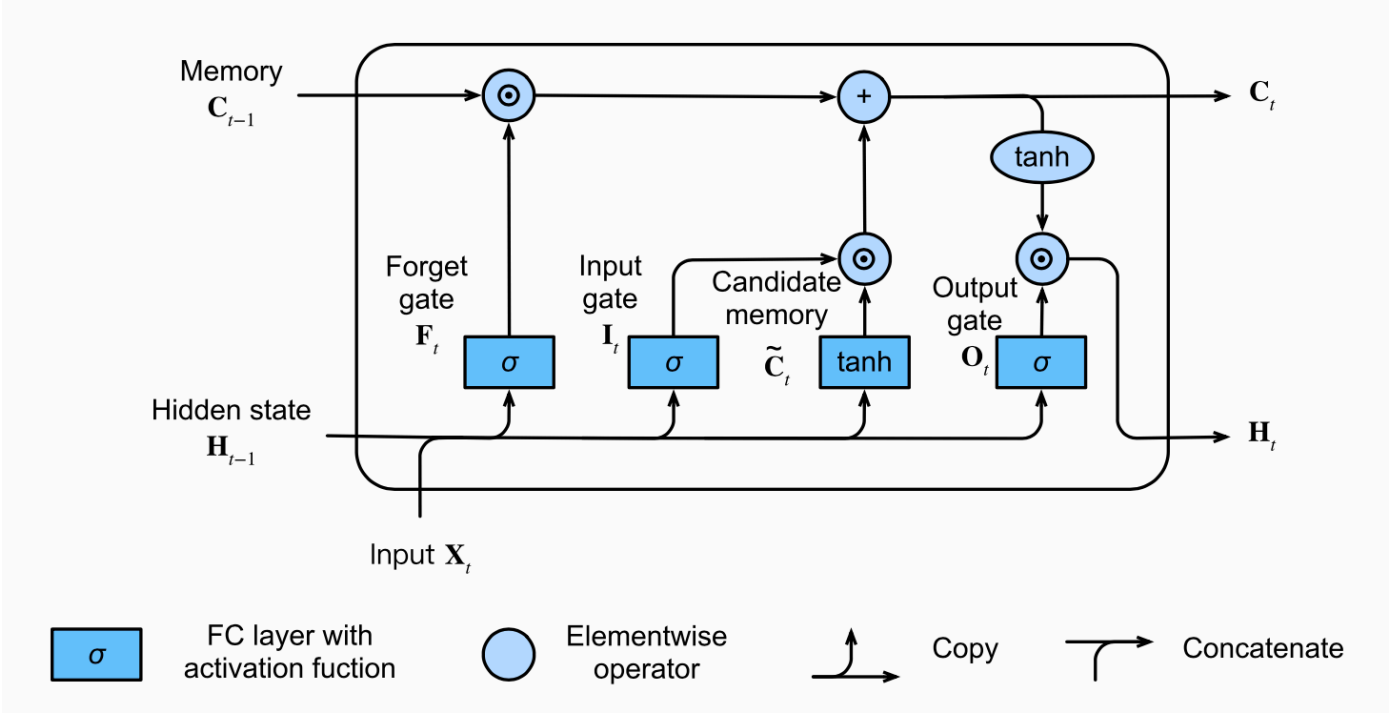
2. 更新记忆单元

$$
\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.
$$

3. 更新隐藏状态

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

这里给一张我觉得很好的LSTM流程图：



就这?

捋清楚模型思路后我们发现：其实所有参数都是一步一步算出来的。因此代码逻辑就是一条直线（甚至还有很多都是重复的，比如：计算三个参数）

为了不让这部分太空，就记录一个实现中遇到的问题吧：

RuntimeError: Trying to backward through the graph a second time, but the buffers have already been

原因：程序在试图执行backward的时候，发现计算图的**缓冲区已经被释放**。

解决办法：在`backward()`函数中添加参数`retain_graph=True`

对不起 我错了

在实现一层LSTM的时候很快就写完了代码，好！开始测试！

先说结论：以后能直接调用api就直接用，千万不要犹豫！

对于助教学长提供的数据集，我的电脑用40分钟跑了3个batch，然后结果：

```
Epoch: 0001 Batch: 100 /603 loss = 8.889562 ppl = 7255.84
Epoch: 0001 Batch: 200 /603 loss = 8.842447 ppl = 6921.91
Epoch: 0001 Batch: 300 /603 loss = 8.804670 ppl = 6665.3
```

也许训练结果会变好，但显然我等不到那天了。。。

我想了一下原因：这里只有一层，而且代码能跑通。照理来说代码逻辑应该不会错（如果代码逻辑真的错了，请您在方便的时候告诉我咋错的，让我死能瞑目。QQ:2804272906 姓名：赵文昊）

在知乎上的提问：为什么pytorch lstm比我们自己写的快?

Hu Xu

1 人赞同了该回答

Pytorch 的lstm基于Nvidia 的cudnn，是更接近cuda硬件指令级的c实现。tensorflow默认的不是但contrib也有nvidia版的。

自己写的八成是用python + pytorch 基本运算，时间浪费在不能连续在gpu端执行cuda运算。

npuichigo

你是指用python写的吗？就cpu来说，用pytorch的c++接口(libtorch)➕for循环并不慢，python太慢了

看了一下，回答主要还是硬件问题。但我想官方的LSTM有没有用到什么技术可以提高效率?

```
LSTM源码：vscode ctr+v查看
```

这一部分都是检查所传参数是否合理

```
def __init__(self, *args, **kwargs):
        super(LSTM, self).__init__('LSTM', *args, **kwargs)

def check_forward_args(self, input: Tensor, hidden: Tuple[Tensor, Tensor],
batch_sizes: Optional[Tensor]):
        self.check_input(input, batch_sizes)
        expected_hidden_size = self.get_expected_hidden_size(input, batch_sizes)

        self.check_hidden_size(hidden[0], expected_hidden_size,
                               'Expected hidden[0] size {}, got {}')
        self.check_hidden_size(hidden[1], expected_hidden_size,
                               'Expected hidden[1] size {}, got {}')
```

forward中（初始化参数以及check）：

```
orig_input = input
# xxx: isinstance check needs to be in conditional for TorchScript to compile
if isinstance(orig_input, PackedSequence):
    input, batch_sizes, sorted_indices, unsorted_indices = input
    max_batch_size = batch_sizes[0]
    max_batch_size = int(max_batch_size)
    else:
        batch_sizes = None
```

```
        max_batch_size = input.size(0) if self.batch_first else input.size(1)
        sorted_indices = None
        unsorted_indices = None

        if hx is None:
            num_directions = 2 if self.bidirectional else 1
            zeros = torch.zeros(self.num_layers * num_directions,
                                max_batch_size, self.hidden_size,
                                dtype=input.dtype, device=input.device)
            hx = (zeros, zeros)
            else:
                hx = self.permute_hidden(hx, sorted_indices)
                self.check_forward_args(input, hx, batch_sizes)
```

```
PS：hx即为我们代码中的Ht以及Ct
```

```
# 开始套娃
if batch_sizes is None:
    result = _VF.lstm(input, hx, self._flat_weights, self.bias, self.num_layers,
                      self.dropout, self.training, self.bidirectional,
self.batch_first)
    else:
        result = _VF.lstm(input, batch_sizes, hx, self._flat_weights, self.bias,
                          self.num_layers, self.dropout, self.training,
self.bidirectional)
```

最后输出

```
output = result[0]
hidden = result[1:]
# xxx: isinstance check needs to be in conditional for TorchScript to compile
if isinstance(orig_input, PackedSequence):
    output_packed = PackedSequence(output, batch_sizes, sorted_indices,
unsorted_indices)
    return output_packed, self.permute_hidden(hidden, unsorted_indices)
else:
    return output, self.permute_hidden(hidden, unsorted_indices)
```

结果我刚想要看vf.lstm怎么实现的，ctr+v点不进去。。。

后来几经周转在pytorch上找到了源码，VF_lstm是用C++实现的！！！

现在在看这个博主的回答，我终于明白他什么意思了。。。

npuichigo

你是指用python写的吗？就cpu来说，用pytorch的c++接口(libtorch) ➕for循环并不慢，python太慢了

在这里，我找到了一个网址：专门讲vf.lstm实现的，我就简单总结一下，详细请看：

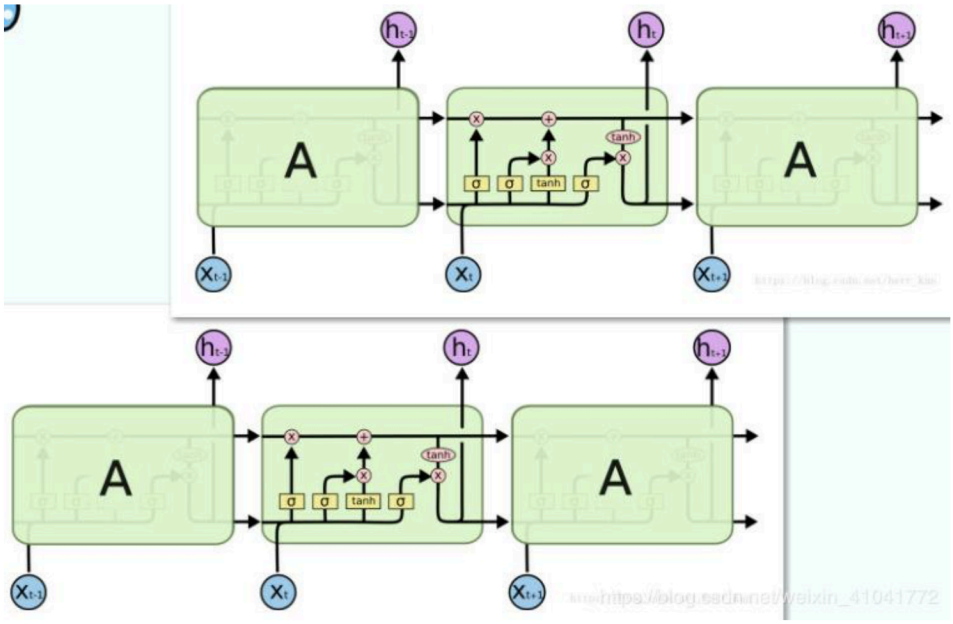
```
ref: https://blog.csdn.net/qq_23981335/article/details/105429676
```

```
template <typename cell_params>
struct LSTMCell : Cell<std::tuple<Tensor, Tensor>, cell_params> {
  using hidden_type = std::tuple<Tensor, Tensor>;
  hidden_type operator()(const Tensor& input, const hidden_type& hidden, const cell_params& params) c
    auto hx = std::get<0>(hidden);
    auto cx = std::get<1>(hidden);

    if (input.is_cuda()) {
      auto igates = params.matmul_ih(input);
      auto hgates = params.matmul_hh(hx);
      auto result = at::_thnn_fused_lstm_cell(igates, hgates, cx, params.b_ih, params.b_hh);
      // Slice off the workspace argument (it's needed only for AD).
      return std::make_tuple(std::get<0>(result), std::get<1>(result));
    }

    auto gates = params.linear_ih(input) + params.linear_hh(hx);
    auto chunked_gates = gates.chunk(4, 1);

    auto ingate = chunked_gates[0].sigmoid();
    auto forgetgate = chunked_gates[1].sigmoid();
    auto cellgate = chunked_gates[2].tanh();
    auto outgate = chunked_gates[3].sigmoid();

    auto cy = (forgetgate * cx) + (ingate * cellgate);
    auto hy = outgate * cy.tanh();

    return std::make_tuple(hy, cy);
  }
};
```


首先我们就看见了，它将集合起来的chunked_gate分成了4个gate，这和python中的操作不谋而合。然后就是根据门来更新细胞的状态和隐向量，随后组成一个tuple返回,再细查这个cell_params，它包括了所有的RNN带有cell的结构，LSTM和GRU这两个带有cell的RNN。

故事到这里就结束了，最后找到的实现不同之处就是用了C++以及对所有的门进行集合处理。不想再探究下去了，毁灭吧。

多层LSTM



简单地说就是上一层的隐层状态作为下一层的输入

但是我在实现的时候想到一个问题：不同层之间的隐藏层（Ht,Ct）参数共享么？理论上说不共享，但你永远不知道机器到底学习了什么。因此我打算做一下实验，还是因为训练时间过长的问题，我只处理了部分训练数据，但是从实验结果看差别不大。如果真的共不共享对实验结果影响不大的话，那显然共享可以节省空间。

最后插一句

在实验前，我一直以为搭建模型是最难的，但是其实他大部分都是维数错误，比较好debug。但是在自己完成所有attention模块后，我发现model不是boss，而是其他的处理模块（比如数据预处理，make_batch什么的），也许是不熟练，但是感觉那里用了我更多的时间。不过感谢老师只让我们实现模型就可以！

Update 2021.11.7

和同学讨论了一下为什么我的训练这么慢，发现问题主要出在矩阵乘法上

矩阵乘法我使用的是 A@B 的写法（一个epoch大概15分钟）

对于矩阵乘法优化我有一个误区：就是认为只有GPU才会进行优化，对于CPU而言，@、dot、matmul都是一样的处理效果，但是当我仅仅把代码中的@换为matmul后，发现训练一个epoch仅需要大概1分钟的时间（我设计的batch_size=128）

最后我在colab上用gpu做了一下实验：训练一个epoch需要10秒钟

```
Epoch: 0001 Batch: 100 /603 loss = 8.854028 ppl = 7002.54
Epoch: 0001 Batch: 200 /603 loss = 8.747683 ppl = 6296.08
Epoch: 0001 Batch: 300 /603 loss = 8.633384 ppl = 5616.05
Epoch: 0001 Batch: 400 /603 loss = 8.561346 ppl = 5225.71
Epoch: 0001 Batch: 500 /603 loss = 8.383341 ppl = 4373.6
Epoch: 0001 Batch: 600 /603 loss = 8.209922 ppl = 3677.26
```

在使用gpu实验的时候我遇到了一个问题：

```
Expected all tensors to be on the same device, but found at least two devices,
cuda:0 and cpu!
```

解决方法比较简单：就是在自己设置的矩阵后面加上 .to(device) 即可

```
例如：self.Wxi0 = (torch.randn(emb_size, n_hidden)*0.01).to(device)
```

日后有时间的话，我也想了解一下矩阵乘法是如何优化的（效果这么明显我开始是没想到的）

Update 2021.11.12

还是感觉自己的实现出了问题，自己查找了一些其他人实现的lstm（自己只是在我已经实现的基础上进行修改，并没有抄袭），发现问题主要出在：如何进行维度转换?

我最早的实现：使用矩阵进行变换

```
self.Wxi = nn.randn(emb_size, n_hidden)
torch.matmul(self.Wxi, Xt)
```

修改后：使用PyTorch线性层进行转换

```
self.Wxi = nn.Linear(emb_size, n_hidden, bias=False)
```

将所有的矩阵乘法替换为pytorch的layer实现后，发现速度变得极快（在cpu下一个epoch也就是六七秒钟）

为什么会这样？或者说用矩阵乘法进行维度变换和layer层有什么区别?

我猜测：可能是前者矩阵初始化的不太合理，导致训练的ppl很大（7000多，正常情况下为600多），但是时间上为什么会那么长呢？可能与layer的底层实现有关？我不是很明白，若老师或者学长能够解答这个疑问，请联系我（QQ：2804272906 赵文昊）

最后展示一下实验结果：

自己实现 层数为1

```
Epoch: 0001 Batch: 100 /603 loss = 6.963454 ppl = 1057.28
Epoch: 0001 Batch: 200 /603 loss = 6.484155 ppl = 654.686
Epoch: 0001 Batch: 300 /603 loss = 6.600259 ppl = 735.286
Epoch: 0001 Batch: 400 /603 loss = 6.834727 ppl = 929.574
Epoch: 0001 Batch: 500 /603 loss = 6.285437 ppl = 536.699
Epoch: 0001 Batch: 600 /603 loss = 6.421124 ppl = 614.694
Epoch: 0001 Batch: 604 /603 loss = 5.910603 ppl = 368.928
Valid 5504 samples after epoch: 0001 loss = 6.282809 ppl = 535.29
Epoch: 0002 Batch: 100 /603 loss = 6.007695 ppl = 406.545
Epoch: 0002 Batch: 200 /603 loss = 5.938262 ppl = 379.275
Epoch: 0002 Batch: 300 /603 loss = 6.171190 ppl = 478.756
Epoch: 0002 Batch: 400 /603 loss = 6.479310 ppl = 651.521
Epoch: 0002 Batch: 500 /603 loss = 6.048641 ppl = 423.537
Epoch: 0002 Batch: 600 /603 loss = 6.114794 ppl = 452.503
Epoch: 0002 Batch: 604 /603 loss = 5.626049 ppl = 277.563
Valid 5504 samples after epoch: 0002 loss = 6.122969 ppl = 456.217
Epoch: 0003 Batch: 100 /603 loss = 5.834625 ppl = 341.936
Epoch: 0003 Batch: 200 /603 loss = 5.622142 ppl = 276.481
Epoch: 0003 Batch: 300 /603 loss = 5.920704 ppl = 372.674
Epoch: 0003 Batch: 400 /603 loss = 6.206333 ppl = 495.879
Epoch: 0003 Batch: 500 /603 loss = 5.859174 ppl = 350.435
Epoch: 0003 Batch: 600 /603 loss = 5.888068 ppl = 360.708
Epoch: 0003 Batch: 604 /603 loss = 5.397559 ppl = 220.867
Valid 5504 samples after epoch: 0003 loss = 6.014072 ppl = 409.146
Epoch: 0004 Batch: 100 /603 loss = 5.652003 ppl = 284.862
Epoch: 0004 Batch: 200 /603 loss = 5.322270 ppl = 204.848
Epoch: 0004 Batch: 300 /603 loss = 5.683544 ppl = 293.99
Epoch: 0004 Batch: 400 /603 loss = 5.961515 ppl = 388.198
Epoch: 0004 Batch: 500 /603 loss = 5.671916 ppl = 290.591
Epoch: 0004 Batch: 600 /603 loss = 5.682182 ppl = 293.589
Epoch: 0004 Batch: 604 /603 loss = 5.209263 ppl = 182.959
Valid 5504 samples after epoch: 0004 loss = 5.942752 ppl = 380.982
Epoch: 0005 Batch: 100 /603 loss = 5.470592 ppl = 237.601
Epoch: 0005 Batch: 200 /603 loss = 5.065292 ppl = 158.427
Epoch: 0005 Batch: 300 /603 loss = 5.463360 ppl = 235.889
Epoch: 0005 Batch: 400 /603 loss = 5.733167 ppl = 308.946
Epoch: 0005 Batch: 500 /603 loss = 5.507286 ppl = 246.481
Epoch: 0005 Batch: 600 /603 loss = 5.484253 ppl = 240.869
Epoch: 0005 Batch: 604 /603 loss = 5.040599 ppl = 154.563
Valid 5504 samples after epoch: 0005 loss = 5.899819 ppl = 364.971

Test the LSTMLM……
Test 768 samples with models/LSTMlm_model_epoch5.ckpt……
loss = 5.782712 ppl = 324.638
```

自己实现 层数为3

```
,
Epoch: 0001 Batch: 100 /603 loss = 6.681307 ppl = 797.361
Epoch: 0001 Batch: 200 /603 loss = 6.486771 ppl = 656.4
Epoch: 0001 Batch: 300 /603 loss = 6.606194 ppl = 739.663
Epoch: 0001 Batch: 400 /603 loss = 6.883464 ppl = 976.001
Epoch: 0001 Batch: 500 /603 loss = 6.419839 ppl = 613.904
Epoch: 0001 Batch: 600 /603 loss = 6.615551 ppl = 746.616
Epoch: 0001 Batch: 604 /603 loss = 6.068656 ppl = 432.1
Valid 5504 samples after epoch: 0001 loss = 6.453108 ppl = 634.672
Epoch: 0002 Batch: 100 /603 loss = 6.242204 ppl = 513.99
Epoch: 0002 Batch: 200 /603 loss = 6.219300 ppl = 502.351
Epoch: 0002 Batch: 300 /603 loss = 6.398063 ppl = 600.68
Epoch: 0002 Batch: 400 /603 loss = 6.667985 ppl = 786.809
Epoch: 0002 Batch: 500 /603 loss = 6.295272 ppl = 542.003
Epoch: 0002 Batch: 600 /603 loss = 6.362342 ppl = 579.602
Epoch: 0002 Batch: 604 /603 loss = 5.866087 ppl = 352.865
Valid 5504 samples after epoch: 0002 loss = 6.341941 ppl = 567.898
Epoch: 0003 Batch: 100 /603 loss = 6.114986 ppl = 452.59
Epoch: 0003 Batch: 200 /603 loss = 6.043525 ppl = 421.376
Epoch: 0003 Batch: 300 /603 loss = 6.174054 ppl = 480.128
Epoch: 0003 Batch: 400 /603 loss = 6.463264 ppl = 641.151
Epoch: 0003 Batch: 500 /603 loss = 6.132443 ppl = 460.56
Epoch: 0003 Batch: 600 /603 loss = 6.120615 ppl = 455.144
Epoch: 0003 Batch: 604 /603 loss = 5.589432 ppl = 267.584
Valid 5504 samples after epoch: 0003 loss = 6.215983 ppl = 500.688
Epoch: 0004 Batch: 100 /603 loss = 5.926125 ppl = 374.7
Epoch: 0004 Batch: 200 /603 loss = 5.839946 ppl = 343.761
Epoch: 0004 Batch: 300 /603 loss = 6.005939 ppl = 405.832
Epoch: 0004 Batch: 400 /603 loss = 6.296723 ppl = 542.79
Epoch: 0004 Batch: 500 /603 loss = 6.002092 ppl = 404.274
Epoch: 0004 Batch: 600 /603 loss = 5.957596 ppl = 386.68
Epoch: 0004 Batch: 604 /603 loss = 5.402689 ppl = 222.003
Valid 5504 samples after epoch: 0004 loss = 6.151445 ppl = 469.395
Epoch: 0005 Batch: 100 /603 loss = 5.770391 ppl = 320.663
Epoch: 0005 Batch: 200 /603 loss = 5.647890 ppl = 283.692
Epoch: 0005 Batch: 300 /603 loss = 5.864576 ppl = 352.333
Epoch: 0005 Batch: 400 /603 loss = 6.125545 ppl = 457.394
Epoch: 0005 Batch: 500 /603 loss = 5.886577 ppl = 360.17
Epoch: 0005 Batch: 600 /603 loss = 5.796171 ppl = 329.037
Epoch: 0005 Batch: 604 /603 loss = 5.251020 ppl = 190.761
Valid 5504 samples after epoch: 0005 loss = 6.117557 ppl = 453.755
```

api

对比可知：

1. 我们在训练速度上已经和api基本持平（甚至更快）
2. 自己实现的一层的LSTM与api在预测准确度上差不多，但是多层的LSTM比api稍差一些，应该是训练的轮数不够