一、 Simple RNN

第一步：嵌入（Embedding）

首先需要将 10 维的输入转化为 32 维的向量。可以使用 pytorch 中自带的 torch.nn.Embedding 来进行转化。这一部分的代码已经在 example 中给出。

第二步：RNN

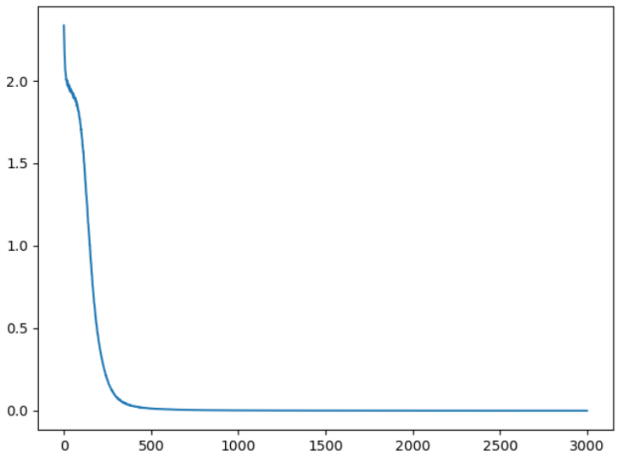
在 Embedding 结束之后，就可以将处理好的两个向量拼接起来，作为 Input，输入到 RNN 中。Pytorch 中自带了 RNN 的函数，只需要将参数设置好即可。这里的 Simple RNN 中使用的是单层的 RNN，也即只有一层隐含层的 RNN，所以在层数上设置为 1。另外再设置好维数等参数，就可以直接调用函数，获得 Output。这一部分代码放在 Forward()中，也有一部分在 example 中已经给出。

最后要将得到的结果重新降维，从 64 维降回到 10 维输出。使用 nn.dence 可以完成这个功能，代码在 example 中已经给出。

第三步：训练

Pytorch 中自带了很多用于训练的函数。包括一些损失函数和优化方法。这里采用了 example 中给出的交叉熵损失函数，以及 Adam 优化器。

下面是训练的结果：




```
step 100 : loss 1.6949189901351929
step 150 : loss 0.9802579283714294
step 200 : loss 0.4196445941925049
step 250 : loss 0.1849830001592636
step 300 : loss 0.08364889025688171
step 350 : loss 0.0463106743991375
step 400 : loss 0.027795664966106415
step 450 : loss 0.018594512715935707
step 500 : loss 0.013210367411375046
```

可以看到，loss 以极快的速度收敛于 0。最终也得到 accuracy 为 1 的结果。

二、 进一步的讨论与优化

1、使用 GPU 代替 CPU

在刚刚进行的学习中，3000 步的总耗时为 140 秒。

在网上学习 pytorch 用法时了解到，pytorch 自带功能，可以将 CPU 运行转化为 GPU 运行，可以在一定程度上提升运行的效率。

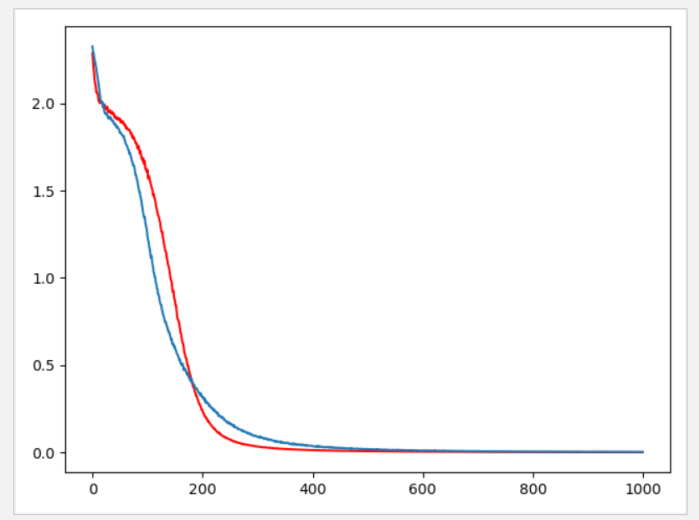
使用.cuda()可以完成从 CPU 到 GPU 的转化。需要对包括输入，模型以及输出三个方面转化到 cuda()运行。

转化完之后进行测试，准确率没有明显的变化，但是总的耗时则下降到了 55 秒左右，总效率提升了约 250%

2、使用不同的激活函数

查资料了解到，pytorch 中默认使用 tanh 函数作为激活函数。可以试图使用不同的激活函数，来观察是否能取得更好的效果。

于是在 RNN 模型中显式地给出了激活函数，进行测试。下面是测试结果，其中，红线代表默认的 tanh 函数，蓝线则是 relu 激活函数。



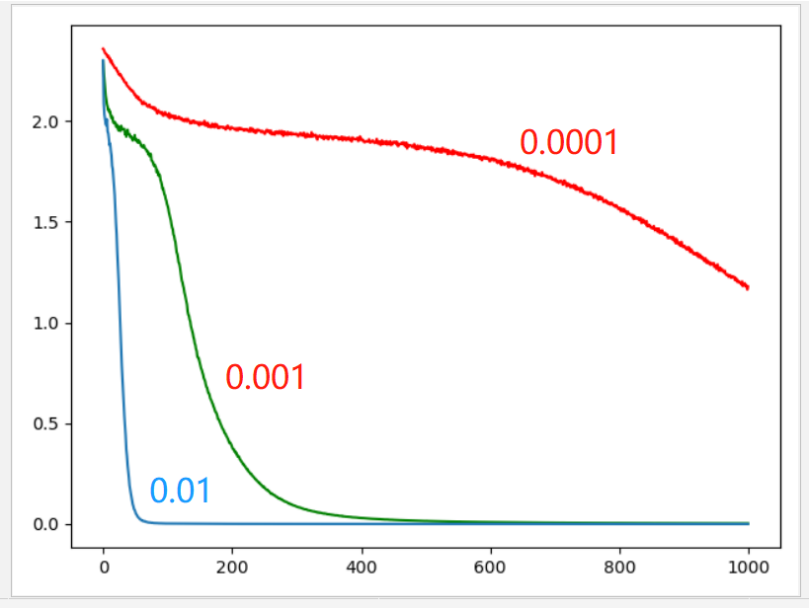
可以看到，二者性能相仿，relu 一开始的表现更好，但最终的收敛速度不及 tanh。

3、使用不同的学习率

下一步是探讨更好的学习率。在 example 中，给出的默认学习率是 0.001，在该学习率下，大概能在 500 步时收敛到 0。

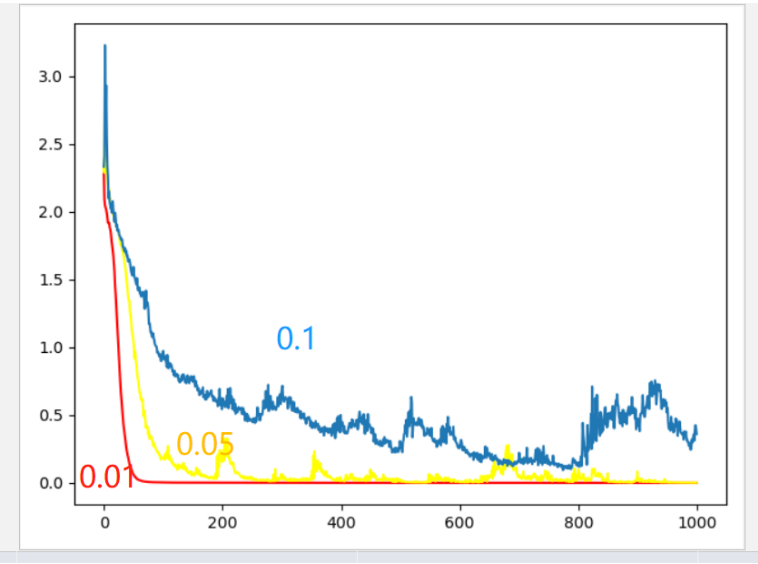
可以尝试更大或更小的学习率，来观察收敛速度以及收敛效果的变化。

首先设置了 3 组学习率：0.0001，0.001 和 0.01



很明显，0.01 的学习率远好于 0.001 和 0.0001。0.01 的学习率可以在 100 左右就收敛到 0。

接下来根据 0.01，又设置了 3 个学习率：0.01，0.05 和 0.1



可以看到，虽然 0.05 的收敛速度也较快，但是会出现比较大的波动，收敛效果并不好。0.1 的学习率波动极大，甚至无法收敛到 0。

综上所述，选择 0.01 附近的学习率较佳。

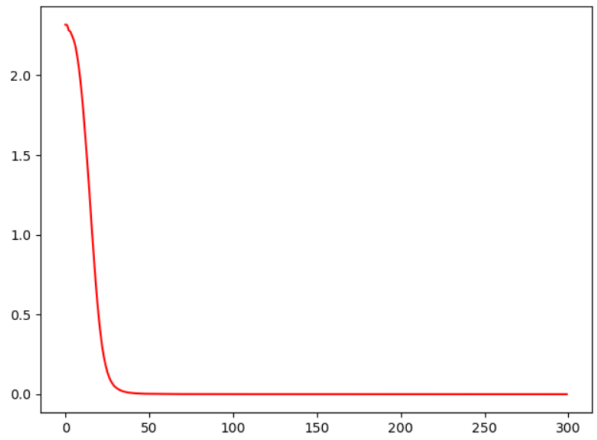
4、更困难的数据

在前一节的 Simple RNN 中，采用的数据是 10 位的。我们可以将数据的位数调大，来观测 RNN 的性能。

首先需要重写数据生成的代码。由于 numpy 的随机数范围不够，只能使用 python 自带的 random 模块中的随机数。

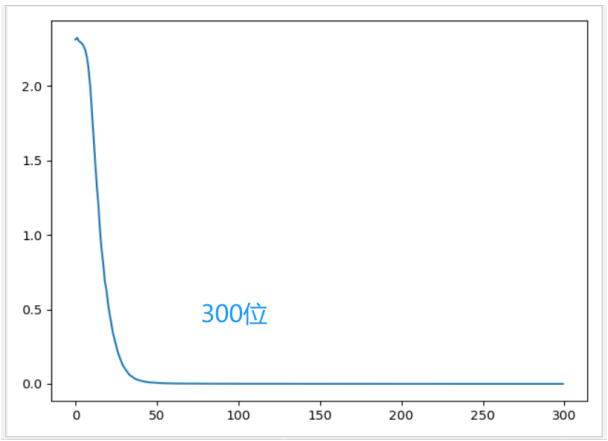
```
numbers_1 = [random.randint(start,end) for i in range(batch_size)]
numbers_2 = [random.randint(start,end) for i in range(batch_size)]
results = [i + j for i, j in zip(numbers_1, numbers_2)]
```

获得了新的随机数生成方法之后，我们将数据位数调整到 100 位。试图观测 RNN 的工作情况。



可以看到，在 100 位的情况下，依然能非常正常且快速的收敛到 0。不过，程序的运行时间有所增加。

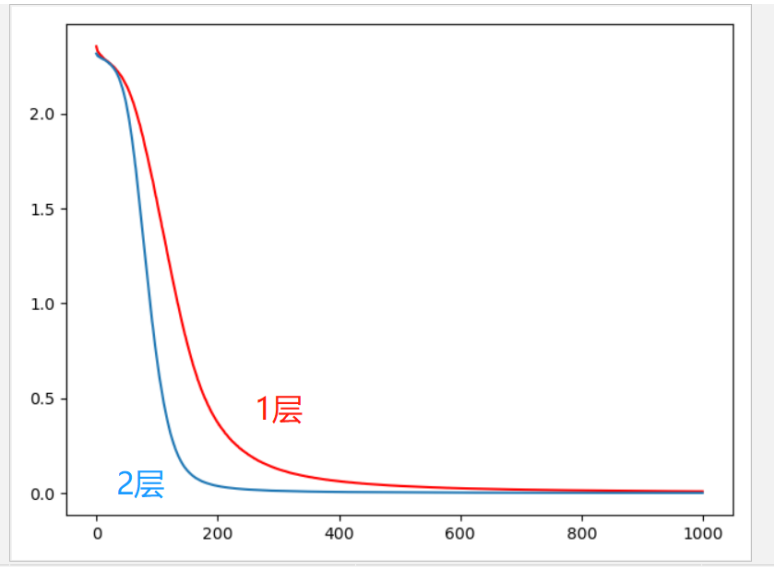
后续继续增大数据的位数，到 300 位仍工作正常。不过，由于电脑性能限制，在运行 500 位的时候发现显存不足，于是没有继续向上测试。



5、多层 RNN

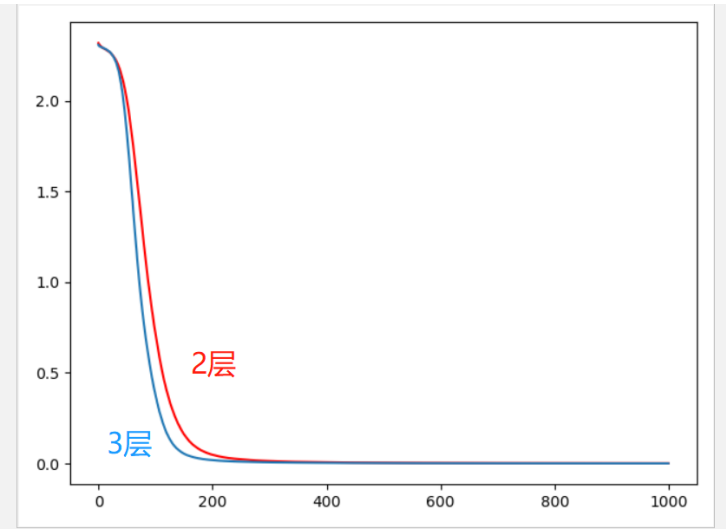
如果 RNN 拥有多个隐含层，那就是堆叠 RNN(Stacked RNN)。在 pytorch 中，RNN 自带的参数 num_layers 可以改变 RNN（默认为 1 层，example 中给的是 2 层）。于是我们修改 RNN 的层数，来观察层数对于收敛速度以及效果的影响。（为了更好观察，将学习率调回了 0.001）

首先测试了 1 层 RNN 和 2 层 RNN 的效果。



可以看到，2 层 RNN 的收敛速度明显要快于 1 层 RNN。

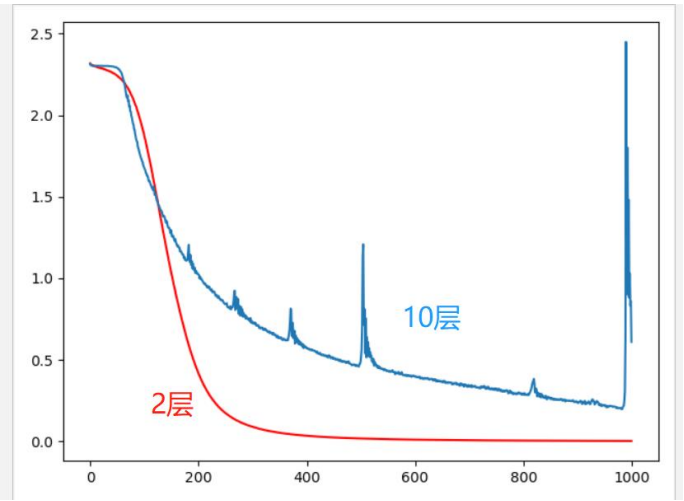
然后比较 2 层 RNN 和 3 层 RNN 的效果。



可以看到，3 层的 RNN 收敛速度要略快于 2 层 RNN，但是提升的幅度相较于

1 层到 2 层要少了很多。

为了观察层数非常多的 RNN 效果，又测试了 10 层的 RNN。



可以看到，10 层的 RNN 并没有出现很好的性能，反而是出现了非常大的波动，并且无法收敛到 0。

综上所述，选择 2 层或 3 层的 RNN 效果就已经可以达到较高水平。

三、 发现的其他问题

在实验中，还发现了一些其它问题。

1、Transpose 问题。最开始实验的时候，没有执行 transpose 这一步，发现无论如何都无法收敛且错误率很高。后来发现原来是没有转置，执行了 transpose()之后，就解决了这个问题。

2、pytorch 的版本问题：最开始安装的是 1.2 版本的 torch，在这个版本中，没有报出任何错误。但是后来重新安装了 1.5 版本的 torch，这个版本中，原先进行的 transpose()会出现报错，说 tensor 不连续。最后只能通过在 transpose 后面立即加上.contiguous()来保持其连续性，才解决了这个问题。目前怀疑是版本问题。

3、cuda 的使用问题。一开始发现无法使用 cuda，后来才知道是 torch 的版本不对，于是卸载了旧的用 pip 安装的 torch1.2，在 pytorch 的官网上，对照着 cuda 的版本重新安装了新版 torch1.5。后来使用 cuda 时，并没有将输入，模型以及输出都放在 cuda 上，出现了大量报错，将 3 者都用.cuda()

处理后问题解决。

四、 总结

在经过了上面的试验之后，得到了目前较好的执行方案：

1、使用 Cuda()用 GPU 来进行运行

2、学习率设置为 0.01 左右，可以达到最佳

3、使用 3 层 RNN 模型进行训练

4、采用默认的 tanh 激活函数

通过这几点，无论位数，都可以做到在 100 步左右收敛，并且收敛效果较好，没有出现明显波动。另外，使用 cuda()使用 gpu 运行，可以大幅减少运行时间。运行时间相比不使用 cuda 要快 2 倍以上。

实验配置：cuda10.2，torch1.5.0，python3.7。

文件组织：运行 source.py 即可运行两个模型，并附上了作图功能。其余文件放在文件夹 handout 中。