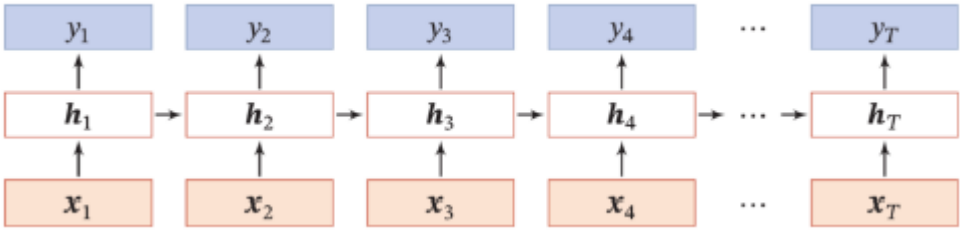
Report of Assignment 2

PRML-Spring20-FDU

Part 1：使用 PyTorch 的 RNN 实现加法器

描述：

在 PyTorch 框架下，使用循环神经网络实现加法器的功能。



网络的各层已经在 example 中定义好了，对于模型上，我们需要补完的就是 forward 的过程。对于加法器，我们的输入为 2 个整数 a、b，以**位的列表**表示，如 a=123，b=456 将被划分为[3,2,1]和[6,5,4]。输出也是整数的分位表示。

我们首先将整数 a、整数 b 通过 embed_layer，将单个数字映射到一个高维的空间（像我们处理 word embedding 一样）得到两者的 **embedding**，然后将两者的 embedding 在张量的**最后一个维度上进行连接**，好让 a、b 中的第 i 位的 embedding 能同时作为第 i 个 timestep 的输入。连接的结果就是 RNN 的输入，让它经过我们定义好的 rnn，得到的**第一个返回值即每个 timestep 的输出**。将 RNN 的输出过一个**线性层 dense**，使每一个 timestep 的维度重新压缩到 10，就得到了 logits。logits 通过 softmax 即可得到模型预测的结果，再与给定的标签作交叉熵得到 loss。

基础的功能实现到这里就完成了，并很快取得了**测试集上 100%**的准确率。

为了提高训练效率，使用 **GPU** 进行训练。

为了后续实验，尝试增加整数长度，新增参数 **maxlen**，表示数的长度。训练集中的数据在[0, int(maxlen*'5') -1]之间，测试集的数据在[int(maxlen*'5') , int(maxlen*'9')]之间。

Command Lines:

python source.py 从 0 开始训练网络并在每个 epoch 进行测试

分析：

循环神经网络和我们从小就学习的 2 位整数的加法在**逻辑上是非常接近**的：给定整数 a、b，我们总是将 a、b 的最低位**对齐**，从最低位向最高位**逐位**相加，得到每一位的结果，并传送这一位的**进位信息**。这个过程，从低到高每一位都是一个 timestep，**进位信息被包含在 timestep 间传递的隐状态中**，每一位的 2 数之和加上上一位的进位信息（隐状态）得到了该 timestep 的隐状态，取隐状态的个位即是该层的输出，再将隐状态传递给下一个 timestep，使下一个 timestep 能读取到之前的进位信息。

这个问题相对来说是对 RNN 是**非常简单**的——因为每一个 timestep 的隐状态除了输入，就**仅和上一个 timestep 的隐状态有关**，而不是之前所有的 timestep，**依赖距离太短**。因此，在训练充足的情况下，可以很容易地在测试集上达到 100%准确率。

Part 2：调整数的长度，分析模型表现并对模型进行优化

Part 2.1：调整数的长度，分析模型表现

描述:

根据题意，尝试增大数据的长度。我**先后将数据长度扩张到 250、1000、5000、25000 位**，模型均能在一定的训练后达到测试集 100%的准确率。

随着数的长度变大，每一个样本占用的内存增加，数据无法完整地放在 GPU 上，因而将数据进行分批，使用 **MGGD**，并将原始的一次性训练 3000 个 step 改为了训练若干个 epoch，**每个 epoch 进行一次 evaluate**。同时，训练时间变长，因而增加了**参数保存**的功能。在参数将被保存在一个 pt 文件中，可以通过命令行 --load 声明需要从 pt 文件读取参数。

Command Lines:

python source.py **--load --maxlen** 250 指定输入的长度为 250 位，并从 maxlen.pt 文件中读取训练过的参数（如果有的话）。

实验结果:

数据长度（位）	（初始）9	250	1000	5000	25000
测试集准确率	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**

输入数字长度在不同值时，模型在训练集上取得的准确率

分析:

可见，由于问题本身完全没有长距离依赖，对于 RNN 来说已经过于简单。**如果模型能够以 100%的准确率预测任何一位，那么无论数据长为多少，准确率都是 100%。**必须让模型对每一位预测正确的概率不等于 1。当每一位的预测准确率不为 1 时，对于**一个数预测的准确率，实际上是每一位的平均准确率之积**，数的长度越长，模型犯错的概率越大。数的长度为 250 位时，犯错的概率是数的长度为 9 位时的 25 次方。**增加数的长度并没有增加问题的难度，只是增加了模型犯错的机会。**

因此，**在后续中控制输入数的长度为 250**，这是一个相比 int 来说相当大的数，但不会导致训练代价过高。

Part 2.2：提升问题对模型的难度

分析:

为了进行下一步的实验，这里首先要**提升问题的难度**，即让现有模型在问题上的准确率下降。

考虑以下可能：

1. 是否能设计一个方法让模型更快的收敛?

2. **固定训练集**，反复在一个较小的训练集上训练，测试模型在数据有限的情况下的能力。

3. 将问题复杂化，如变成小数的加法、二进制相加输出十进制或十进制相加输出二进制等。

就以上选择来说，第一个问题往往可以通过增大学习率实现，讨论的意义不大。

第三个方法中，二进制与十进制的长度相差非常多，如果二进制作为输入，十进制作为输出，则根本无法在十进制的第 i 位得到需要的所有信息；即使十进制作为输入，二进制作为输出，也会遇到严重的长距离依赖问题；小数的加法需要以小数点进行对齐，而对齐后与整数加法没有本质差异。

因此，接下来考虑使用第二个方法，**固定训练集**，考察如何提升模型**在有限数据下的拟合能力和泛化能力**。

Part 2.3：固定训练集并调整大小以降低模型在测试集上的准确率

描述：

原始代码中，进行了 3000 次训练，每次训练有 200 个样本，总共生成了 600000 个训练样本。这里，我们在 pt_adv_main 中生成训练集和测试集，保证每次训练使用同一个集合。从 10000 开始不断下调训练集样本数量。

Command Lines:

python source.py --adv --maxlen 250 --train_dataset_size 13

--adv 表示使用固定训练集的更复杂的问题（但不代表使用 myAdvPTRNNModel）

--train_dataset_size 指定训练集大小

实验结果:

训练集大小	>=50	20	15	13	12	11
最优测试集准确率	1	0.998	0.998	0.933	0.302	0
取得最优准确率时的 EPOCH	-------	1200	9702	12084	1660	-----

不同数据集大小在 RNN 上取得的最优准确率及训练 EPOCH 数

依次将训练集样本数量修改为 10000、3000、1000、100、50、20、15、13、12、11、10，发现自己还是低估了模型的拟合能力，直到我将训练集样本数量修改为 20，训练集上才第一次没有达到 1%的准确率。当样本数量修改为 15 后，模型最终依然取得了 0.998 的准确率，但却多花了 7 倍的 epoch。有趣的是，当训练集大小在 11-15 间移动的时候，得到的测试集准确率惊人。当训练集有 13 个样本时，模型的表现已经离 100%有了较大差距，为 0.933。当模型有 12 个样本的时候，模型的准确率大大下降，只剩 0.302，但还是有 1/3 的情况能算准。当模型的样本减少到 11 个的时候，测试集上的准确率降低到了 0，即使我训练了 20000 个 epoch。

分析：

考虑到样本长度为默认值250,这个结果也是可以解释的。当样本数从13降低到12时，准确率变成了约三分之一。由于一次预测正确的概率等于251（2者的和多一位）次位预测正确的准确率的积。平均来说，总体准确率0.933，相当于每位准确率0.9997；总体准确率0.302，相当于每位准确率 0.9952；而要使总体准确率大于 0.0001，每位准确率依旧需要0.9640。在每一位上微小的变化，就会给预测的准确率带来巨大的差异。而训练集从11 增加到15 的过程，会给每一位的准确率带来可见的改变，因而导致最终的结果。

Part 2.4：改进模型

描述：

前面已经提到，我不想在模型的效率方向进行考虑，因为改变模型的学习率就会对效率产生很大影响。因此，这里考虑的改进，指**如何在数据集极其有限的情况下提高模型的泛化能力。**

因为本章要求使用 RNN-based neural networks，很自然地考虑到 RNN 的变体 LSTM。虽然我们说，对于加法器问题，RNN 的实现具备非常好的可解释性，但加入了 softgate 的 LSTM 似乎**具备更好的可解释性**。比如说，假设细胞状态 ci 表示第 i 位的两数之和 加上进位信息，hi 表示第 i 位的输出的话，**输出门**就可以控制 hi=ci%10，使**输出去掉进位的数字**，而**遗忘门**可以控制 ci = (ci-1) /10 忘掉上一时刻的最后一位，**只保留进位信息**。

在此之上，因为 LSTM 具有比 RNN 更强的能力，也就意味着在小数据集上 LSTM 更容易过拟合（而在前面的实验中，RNN 似乎很少出现过拟合现象），考虑加入 dropout 防止过拟合。

Command Lines:

python source.py --adv --maxlen 250 --train_dataset_size 11 **--rnn_adv**

--rnn_adv：使用 myAdvPTRNNModel。

实验结果:

Dropout	0	0.1	0.2	0.3	0.4	0.5
准确率	0	0.988	1.0	1.0	1.0	1.0
EPOCH	--------	9900	8580	4356	3168	7590

当训练集大小为 15 时，采用**不同 Dropout** 的 LSTM 的最优准确率及训练 EPOCH

数据集大小	15	13	12	11	8	7
准确率	1.0	1.0	0.984	0.994	0.391	0
EPOCH	7590	8588	15106	10980	27750	---------

dropout=0.5 时，**不同数据集大小**在 LSTM 上取得的最优准确率及训练 EPOCH 数

（作为对比，再次放入 Part. 2.3 中 RNN 的实验结果）

训练集大小	>=50	20	15	13	12	11

最优测试集准确率	1	0.998	0.998	0.933	0.302	0
取得最优准确率时的 EPOCH	-------	1200	9702	12084	1660	-----

不同数据集大小在 RNN 上取得的最优准确率及训练 EPOCH 数

分析:

出乎我意料的是，一开始将模型换为 LSTM 时，准确率一直是 0，让我有点不解。而在添加 dropout 后，模型的准确率就恢复了合理的范围。仔细想来，这是因为 **LSTM 具有比 RNN 更强的能力，在小数据集上更容易过拟合，因而必须使用 dropout 来保证模型的泛化能力**。当 dropout 升到 0.2 时，模型已经能够超越 RNN 达到 1.0 的准确率。在一定范围内，dropout 越大，训练出的子网的数量就越多，模型的泛化能力越强。但 dropout 过高，又会使训练过于困难。最终选择 dropout=0.5。

在加入了 dropout 后，可以发现，**对于数据非常有限的情况，LSTM 确实具有比 RNN 更强的能力**。当数据集大小为 15、13 时，RNN 准确率分别为 0.998、0.993，而 LSTM 准确率均为 1。当数据集大小为 12 时，RNN 准确率骤降至 0.302，而 LSTM 依然有 0.984。当数据集大小为 11 时，RNN 准确率已经为 0，而 LSTM 依然有接近 1 的准确率。LSTM 要到数据集降到 7 的时候，准确率才变为 0。而且数据集等于 8 的时候，依然能有 0.391 的准确率。