Assignment3 - LSTM 学习报告

2019 年 5 月 26 日

1 Task 1

1.1 隐藏层输出对相关状态的偏导

由前向传播，我们可以得出相关状态的递推式:

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)) = \sigma(W_f h * h_{t-1} + W_f x * x_t + b_f)$$

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)) = \sigma(W_i h * h_{t-1} + W_i x * x_t + b_i)$$

$$\bar{C}_t = tanh(W_{\bar{C}} * [h_{t-1}, x_t] + b_{\bar{C}})) = \sigma(W_{\bar{C}h} * h_{t-1} + W_{\bar{C}x} * x_t + b_{\bar{C}})$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)) = \sigma(W_o h * h_{t-1} + W_o x * x_t + b_o)$$

$$C_t = f_t * C_{t-1} + i_t * \bar{C}_t$$

$$h_t = o_t * tanh(C_t)$$

因此可以通过链式法则推导 h_t 对权值矩阵的偏导。首先求得对 o_t 和 C_t 的偏导:

$$\frac{\partial h_t}{\partial C_t} = o_t * (1 - tanh^2(C_t))$$

$$\frac{\partial h_t}{\partial o_t} = tanh(C_t)$$

$$\frac{\partial h_t}{\partial W_o h} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial W_o h} = tanh(C_t) * o_t * (1 - o_t) * h_{t-1}$$

$$\frac{\partial h_t}{\partial W_o x} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial W_o x} = tanh(C_t) * o_t * (1 - o_t) * x_t$$

$$\frac{\partial h_t}{\partial b_o} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial b_o} = tanh(C_t) * o_t * (1 - o_t)$$

之后，通过 C_t ，对其他的状态进行求导。下面以 f_t 和 W_f 为例进行推导，其中，为了推导书写的方便，把权值矩阵 Wf 横向拆分成两个矩阵 $W_f h$, $W_f x$ 分别表示对于 h_{t-1} 和 x_t 的权重。

$$\frac{\partial h_t}{\partial f_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t} = o_t * C_{t-1} * (1 - tanh^2(C_t))$$

$$\frac{\partial h_t}{\partial W_f h} = \frac{\partial h_t}{\partial f_t}\frac{\partial f_t}{\partial W_f h} = o_t * C_{t-1} * (1 - tanh^2(C_t)) * f_t * (1 - f_t) * h_{t-1}$$

$$\frac{\partial h_t}{\partial W_f x} = \frac{\partial h_t}{\partial f_t}\frac{\partial f_t}{\partial W_f x} = o_t * C_{t-1} * (1 - tanh^2(C_t)) * f_t * (1 - f_t) * x_t$$

$$\frac{\partial h_t}{\partial b_f} = \frac{\partial h_t}{\partial f_t}\frac{\partial f_t}{\partial b_f} = o_t * C_{t-1} * (1 - tanh^2(C_t)) * f_t * (1 - f_t)$$

相似地，其他状态的偏导数为：

$$\frac{\partial h_t}{\partial i_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t} = o_t * \bar{C}_t * (1 - tanh^2(C_t))$$

$$\frac{\partial h_t}{\partial W_i h} = \frac{\partial h_t}{\partial i_t}\frac{\partial i_t}{\partial W_i h} = o_t * \bar{C}_t * (1 - tanh^2(C_t)) * i_t * (1 - i_t) * h_{t-1}$$

$$\frac{\partial h_t}{\partial W_i x} = \frac{\partial h_t}{\partial i_t}\frac{\partial i_t}{\partial W_i x} = o_t * \bar{C}_t * (1 - tanh^2(C_t)) * i_t * (1 - i_t) * x_t$$

$$\frac{\partial h_t}{\partial b_i} = \frac{\partial h_t}{\partial i_t}\frac{\partial i_t}{\partial b_i} = o_t * \bar{C}_t * (1 - tanh^2(C_t)) * i_t * (1 - i_t)$$

$$\frac{\partial h_t}{\partial \bar{C}_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial \bar{C}_t} = o_t * i_t * (1 - tanh^2(C_t))$$

$$\frac{\partial h_t}{\partial W_{\bar{C}h}} = \frac{\partial h_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial W_{\bar{C}h}} = o_t * i_t * (1 - tanh^2(C_t)) * \bar{C}_t * (1 - \bar{C}_t) * h_{t-1}$$

$$\frac{\partial h_t}{\partial W_{\bar{C}x}} = \frac{\partial h_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial W_{\bar{C}x}} = o_t * i_t * (1 - tanh^2(C_t)) * \bar{C}_t * (1 - \bar{C}_t) * x_t$$

$$\frac{\partial h_t}{\partial b_{\bar{C}}} = \frac{\partial h_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial b_{\bar{C}}} = o_t * i_t * (1 - tanh^2(C_t)) * \bar{C}_t * (1 - \bar{C}_t)$$

1.2 目标函数对隐藏层输出的偏导

这次 lab 使用的是 softmax 交叉熵损失函数作为目标函数

$$C = -\sum y_i' \log(y_i)$$

其中 y_i' 是实际的输出，其中有某一维是 1，其他维是 0。$y_t = softmax(Z_t) = softmax(W_y * h_t + b_y)$ 是从网络输出层第 t 个神经元得到的输出。考虑 C 关于 h_t 的偏导数：

$$\frac{\partial C}{\partial h_t} = \frac{\partial C}{\partial Z_t} * \frac{\partial Z_t}{\partial h_t} = \sum_j (\frac{\partial C_j}{\partial y_j} * \frac{\partial y_j}{\partial Z_j}) * W_y = (y_t \sum_j y_j' - y_t') * W_y$$

因为对于实际输出 y_i' 只有一维为 1，所以上式最后可以转化为：

$$\frac{\partial C}{\partial h_t} = (y_t - y_t') * W_y$$

1.3 求梯度

有了 1.1、1.2 与链式法则，目标函数对所有权值矩阵的偏导数都可以被求得，对于除了 C_t 之外的中间状态 W

$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial W} = (y_t - y_t') * W_y * \frac{\partial h_t}{\partial W}$$

特别地，对于 $C_t = f_t * C_{t-1} + i_t * \bar{C}_t$，因为 C_t 记录了所有细胞的状态，所以在反向传播时，其梯度也要随着时间反向传递，即：

$$\frac{\partial C}{\partial C_t} = \sum_{i=t}^{T}\frac{\partial C}{\partial h_i}\frac{\partial h_i}{\partial C_i} = \sum_{i=t}^{T}(y_i - y_i') * W_y * o_i * (1 - tanh^2(C_i))$$

2 唐诗生成器

2.1 Preproccess and Word Embedding

生成器以全唐诗作为输入数据。为了方便学习，去掉了唐诗中的所有标点符号，即把所有汉字拉成一个大串。

为了模型计算的效率，处理时仅保留了每首诗的前 100 个字符。即将长度过长（大于 100）的诗截取前 100 个。同时，对于长度不足 100 的诗，在诗的末尾用特殊符号补足。

这样处理之后，我们得到了大小为 6857 的单词字典，若将每个字直接变成 6857 维的 OneHot 向量有些过于大，因此考虑 Embedding 之后再作为输入向量进行深度学习。

这次我们只进行了简单的 Word Embedding，即直接使用 pytorch 内置的 Embedding 方法，将 6857 个单词 Embedding 成 512 维的向量。并以此作为 lstm 的 input_layer 输入维度与 hidden_layer 输出维度。最后，通过一个线性层将 512 维再扩大成 6857 维，得到最后的输出。

2.2 Optimization

本次使用了 Adam 和 Adagrad 两种优化器分别对模型进行了训练，并得到输出。

Adagrad Adagrad 是一种改进的随机梯度下降算法，可以自适应地调节学习率。以前的算法中，每一个参数都使用相同的学习率 α。Adagrad 算法能够在训练中自动对 learning_rate 进行调整，出现频率较低参数采用较大的 更新．出现频率较高的参数采用较小的 α 更新．根据描述这个优化方法很适合处理稀疏数据。

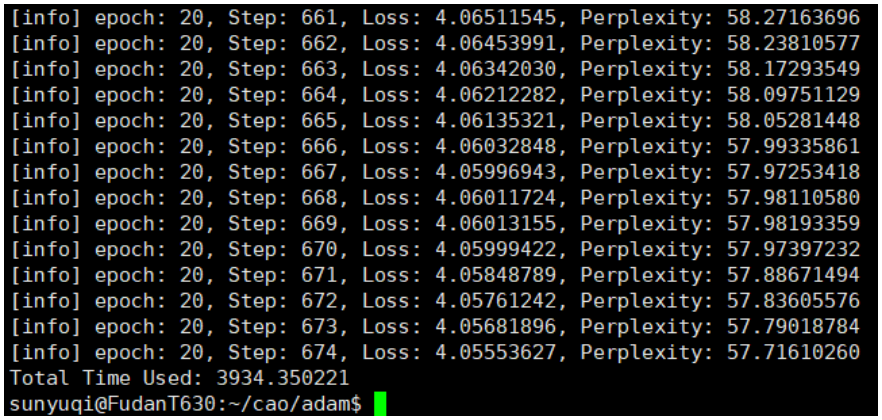

图 1: result of Adam

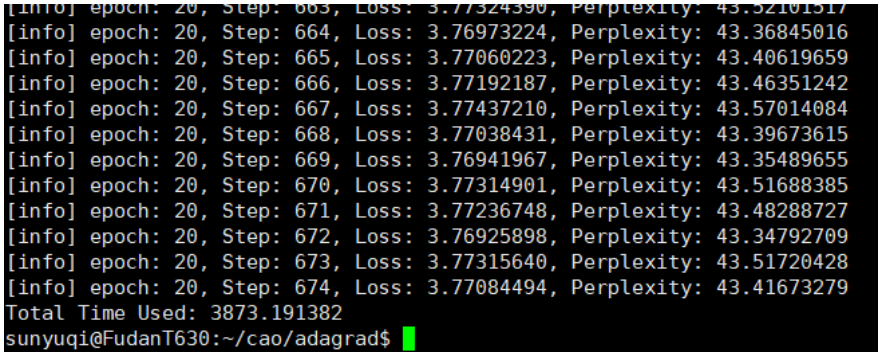

图 2: result of Adagrad

Adam Adam 算法和传统的随机梯度下降不同。随机梯度下降保持单一的学习率 α 更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率。其是 Adagrad 和 RMSProp 两种算法的优秀结合。

训练结果比较 用两种优化器设置 batch size=64，分别迭代了 20 轮，目标函数如上文采用了 softmax 交叉熵损失函数。下图是两种方法的结果。

可以看出，两种方法得出的 loss 都还比较不错。但其中相比之下 Adagrad 效果反而更好。初步分析是函数性质较好，亦或者是因为将没有学习率、batch size 等并没有调整到最优的情况。不过因为时间等原因本次 lab 没有进一步调参，这也是之后可以继续优化的方向所在。

2.3 Generating Tang Poem

对于训练完之后的模型，我们可以开始生成唐诗了。除了给定第一个字之外，我们需要给定诗的格律 (七言还是五言)，以及诗的长度（多少句话）。生成器会先生成一个符合大小的大字符串，然后按格律进行断句。

以下是 Adagrad 优化之后的结果（Adam 的结果相对较差就没有放出来）：

日：日晚东风动天池，月中秋日下无穷。秋草色不成霜气，满长沙风流光彩。

红：红树风流满树秋，雨来江色似相依。青春不得春来事，几年不如故园今。

山：山中多士事无语，白日闲人到江湖。客有花无人不到，白云水到头城外。

夜：夜夜月光秋雨雪，霜秋不是愁心苦。独坐秋云散未知，天涯一梦中有恨。

湖：湖山不在碧江上，万条千尺一万峰。前人未必归江口，更得一生心未回。

海：海棠柳里人生别，旧宅春深意已衰。不如不觉身犹有，泪满东城暮自伤。

月：月上清光里中流，水不流水声随客。处无事不知时独，去孤帆落风沙起。

2.4 lstm with numpy

除 pytorch 版本之外，我用 Numpy 手动实现了 lstm 的前向传播和反向传播逻辑，包括求梯度、梯度下降等。整体的大致逻辑类似，主要是求梯度的过程差别较大。先做一遍前向传播，得出所有函数的值，然后通过 $\delta h_t = \frac{\partial C}{\partial h_t}$，枚举所有时刻对权值矩阵的梯度进行更新。具体算法如下。

Algorithm 1: LSTM with Numpy

Input: 一组样本输入 X, y

对于给定输入 x 做前向传播

$\delta C_t = 0$

求得 δh_t

for *k=T downto 1* **do**

    累加 δC_t

    根据 $\delta C_t, \delta h_t$ 和向量的值更新权重 $W_f, W_i, W_o, W_{\bar{C}}$

$return W_f, W_i, W_o, W_{\bar{C}}$

运行 numpy 版本的 lstm，可以明显的感觉到梯度下降的速度非常慢——大概 20 分钟进行一次梯度下降。经过分析之后，我认为主要原因有两个：

1. 手写 numpy 版本没有利用 GPU 来加速运算，因此关于矩阵的运算效率较低

2. 手写 numpy 版本并发性较差，相比 pytorch 内置并行的版本效率要更差

不过，尽管这次手动实现的结果离一个优秀 lstm 模块还有很多距离，我也从中对 lstm 运行原理和梯度的理解更加深刻。这也是一次非常有意义的尝试