Assignment 3

Assignment 3

方煊杰

复旦大学

16307130335@fudan.edu.cn

准备

package requirements:

- torch
- numpy
- fastNLP
- json

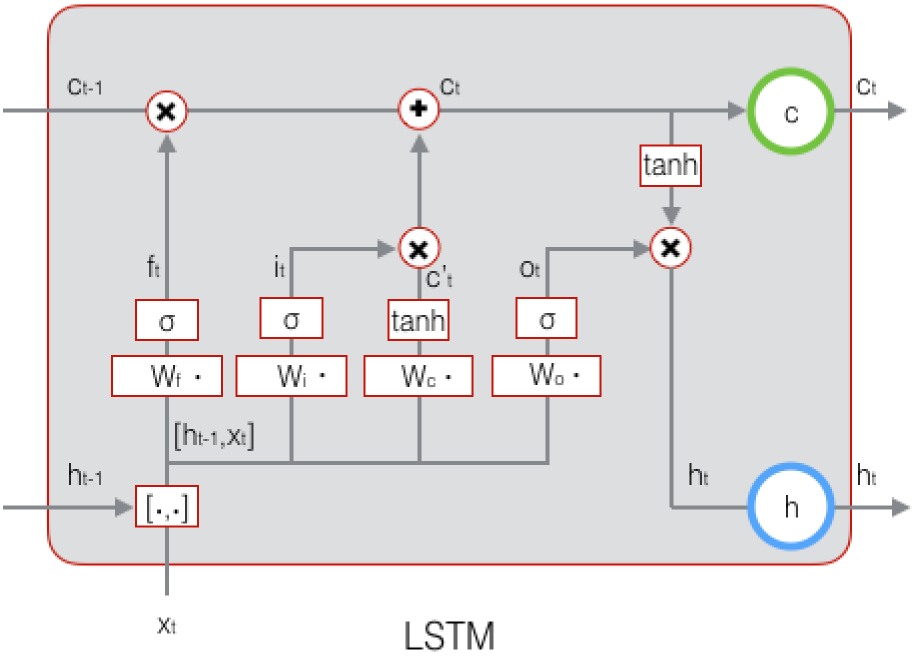
```
#install packages
$pip install torch
$pip install numpy
$pip install fastNLP
$pip install json
```

实现

Part1 LSTM基本内容

ps: 为了更清楚的展示微分过程，在这里也将具体的求导过程用代码实现

1. 前向计算 (forward)



LSTM

对于LSTM中的每一扇门，实际就是一层全连接层，W作为权重，b为偏置项，f为激活函数，那么门可以表示为：

$$g(x) = f(Wx + b)$$

```
# 门的基础函数
def gate(W, x, b, activation):
    return activation(W.dot(x) + b)
```

1.1 遗忘门 (forget gate)

$$f_t = sigmoid(W_f \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_f)$$

为了简便计算，可以将权重矩阵也写为两部分：

$$W_f \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} = [\, W_{fh} \quad W_{fx} \,] \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$
$$\text{维 度：} \; W_{fx} : \; d_c \times d_h; \;\; W_{fx} : \; d_c \times d_x$$

所以遗忘门的最终输出可以写为：

$$f_t = sigmod(W_{fh} \times h_{t-1} + W_{fx} \times x_t + b_f)$$

```
# 修改gate函数, h_pre代表上一时刻的h值
def new_gate(Wh, Wx, h_pre, x, b, activation):
    return activation(Wh.dot(h_pre) + Wx.dot(x) + b)

def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

ft = new_gate(Whf, Wxf, h_pre, x, bf, sigmoid)
```

1.2 输入门 (input gate)

$$
\begin{aligned}
i_t &= sigmoid(W_i \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_i) \\
&= sigmoid(W_{ih} \times h_{t-1} + W_{ix} \times x_t + b_i)
\end{aligned}
$$

```
it = new_gate(Whi, Wxi, h_pre, x, bi, sigmoid)
```

1.3 单元状态

根据上一次的输出和本次输入计算当前输入的单元状态:

$$
\begin{aligned}
\widetilde{c_t} &= tanh(W_c \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_c) \\
&= tanh(W_{ch} \times h_{t-1} + W_{cx} \times x_t + b_c)
\end{aligned}
$$
再计算当前时刻的单元状态:

再计算当前时刻的单元状态:

$$
c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c_t}
$$

```
def tanh(x):
    return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))

# c_pre表示上一时刻c的值
ct =  ft * c_pre + it * new_gate(Whc, Wxc, h_pre, x, bc, tanh)
```

1.4 输出门 (output gate)

输出门受到长期记忆的影响:

$$
\begin{aligned}
o_t &= sigmoid(W_o \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o) \\
&= sigmoid(W_{oh} \times h_{t-1} + W_{ox} \times x_t + b_o)
\end{aligned}
$$

而最终输出由输出门以及单元状态共同决定:

$$
h_t = o_t \circ tanh(c_t)
$$

```
ot = new_gate(Who, Wxo, h_pre, x, bo, sigmoid)
ht = ot * tanh(ct)
```

1.5 交叉熵损失函数层

*事实上**LSTM中并没有这一层**，但是在之后的古诗生成中需要用到，因此在这边也将它加上*

$$y_t = softmax(W_y \times x + b_y)$$

这里线性变换的目的是得到和词典长度相同的向量，从而和目标预测值相比较进行对损失函数的计算。

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))

yt = softmax(np.dot(Wy, x) + by)
```

2. 反向传播 (backward)

首先考虑 sigmoid以及tanh激活函数的导数：

$$sigmoid'(x) = sig(x)(1 - sig(x))$$
$$tanh'(x) = 1 - tanh(x)^2$$

2.1 ht对ot和ct的求导

我们已知ht关于输出门以及单元状态的函数：

$$h_t = o_t \circ tanh(c_t)$$

由于对多维向量的求导我们希望能够拥有 **Wx + b**形式，这样对多维向量x的求导结果就是变量系数W, 于是通过数学观察可以将上述运算改写为下面的形式：

$$h_t = diag[tanh(c_t)] \times o_t = diag[o_t] \times tanh(c_t)$$

由此可得ht关于ot和ct的导数：

$$\begin{aligned} \frac{\partial h_t}{\partial o_t} &= diag[tanh(c_t)] \\ \frac{\partial h_t}{\partial c_t} &= \frac{\partial h_t}{\partial tanh(c_t)} \frac{\partial tanh(c_t)}{\partial c_t} \\ &= diag[o_t \circ (1 - tanh(c_t)^2)] \end{aligned}$$

2.2 ct对ft, it, ~ct的求导

我们已知ct与门输出相关的等式：

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c_t}$$

与2.1描述的相同，可以改写为对角矩阵乘积的形式，从而可以方便地得到偏导：

$$\begin{aligned} \frac{\partial c_t}{\partial f_t} &= diag[c_{t-1}] \\ \frac{\partial c_t}{\partial i_t} &= diag[\tilde{c}_t] \\ \frac{\partial c_t}{\partial \tilde{c}_t} &= diag[i_t] \end{aligned}$$

2.3 考虑gate输出对各自网络层的求导

在前向计算中，每个门的输出是由激活函数作用与网络层上所得到，因此，每个门的输出对网络层的偏导可以转化为对激活函数的求导。

根据前向计算中的公式，先对每个网络层进行定义：

$$
\begin{aligned}
net_{ft} &= W_{fh}h_{t-1} + W_{fx}x_t + b_f \\
net_{ot} &= W_{oh}h_{t-1} + W_{ox}x_t + b_o \\
net_{it} &= W_{ih}h_{t-1} + W_{ix}x_t + b_i \\
net_{\tilde{c}t} &= W_{ch}h_{t-1} + W_{cx}x_t + b_c
\end{aligned}
$$

再考虑各自的激活函数：

$$
\begin{aligned}
f_t &= sigmoid(net_{ft}) \\
o_t &= sigmoid(net_{ot}) \\
i_t &= sigmoid(net_{it}) \\
\tilde{c}_t &= tanh(net_{\tilde{c}t})
\end{aligned}
$$

由此得到偏导：

$$
\begin{aligned}
\frac{\partial f_t}{\partial net_{ft}} &= diag[f_t \circ (1 - f_t)] \\
\frac{\partial o_t}{\partial net_{ot}} &= diag[o_t \circ (1 - o_t)] \\
\frac{\partial i_t}{\partial net_{it}} &= diag[i_t \circ (1 - i_t)] \\
\frac{\partial \tilde{c}_t}{\partial net_{\tilde{c}t}} &= diag[1 - \tilde{c}^2)]
\end{aligned}
$$

2.4 考虑网络层对h_pre的求导

根据2.3中每个网络层的定义，对h_pre的求导即为其系数矩阵：

$$
\begin{aligned}
\frac{\partial net_{ft}}{\partial h_{t-1}} &= W_{fh} \\
\frac{\partial net_{ot}}{\partial h_{t-1}} &= W_{oh} \\
\frac{\partial net_{it}}{\partial h_{t-1}} &= W_{ih} \\
\frac{\partial net_{\tilde{c}t}}{\partial h_{t-1}} &= W_{ch}
\end{aligned}
$$

2.5 考虑网络层对参数矩阵的求导

同样根据2.3中网络层的定义，得到对参数矩阵的求导：(对偏置b的求导即为单位矩阵，所以这里不写出)

$$\frac{\partial net_{ft}}{\partial W_{fht}} = h_{t-1}^T \qquad \frac{\partial net_{ft}}{\partial W_{fxt}} = x_t^T$$
$$\frac{\partial net_{ot}}{\partial W_{oht}} = h_{t-1}^T \qquad \frac{\partial net_{ot}}{\partial W_{oxt}} = x_t^T$$
$$\frac{\partial net_{it}}{\partial W_{iht}} = h_{t-1}^T \qquad \frac{\partial net_{it}}{\partial W_{ixt}} = x_t^T$$
$$\frac{\partial net_{\tilde{c}t}}{\partial W_{cht}} = h_{t-1}^T \qquad \frac{\partial net_{\tilde{c}t}}{\partial W_{cxt}} = x_t^T$$

2.6 通过链式法则求出损失函数对参数的导数

在t时刻，LSTM的输出值为ht, 定义t时刻的误差项为:

$$\delta_t = \frac{\partial E}{\partial h_t}$$
$$\delta_{ft} = \frac{\partial E}{\partial net_{ft}} \qquad \delta_{ot} = \frac{\partial E}{\partial net_{ot}}$$
$$\delta_{it} = \frac{\partial E}{\partial net_{it}} \qquad \delta_{\tilde{c}t} = \frac{\partial E}{\partial net_{\tilde{c}t}}$$

根据链式法则:

$$\delta_{ft} = \frac{\partial E}{\partial net_{ft}} = \frac{\partial E}{\partial h_t} \frac{\partial h_t}{\partial f_t} \frac{\partial f_t}{\partial net_{ft}}$$

根据2.1-2.5中已经计算的偏导结果可得:

$$\delta_{ot}^T = \delta_t^T \circ tanh(c_t) \circ (1 - o_t)$$
$$\delta_{ft}^T = \delta_t^T \circ o_t \circ (1 - tanh(c_t)^2) \circ c_{t-1} \circ f_t \circ (1 - f_t)$$
$$\delta_{it}^T = \delta_t^T \circ o_t \circ (1 - tanh(c_t)^2) \circ \tilde{c}_t \circ i_t \circ (1 - i_t)$$
$$\delta_{\tilde{c}t}^T = \delta_t^T \circ o_t \circ (1 - tanh(c_t)^2) \circ i_t \circ (1 - \tilde{c}^2)$$

2.7 通过链式法则计算梯度偏导

在2.6中我们已经求得了误差项，于是接下来我们就很容易地表示出t时刻梯度:

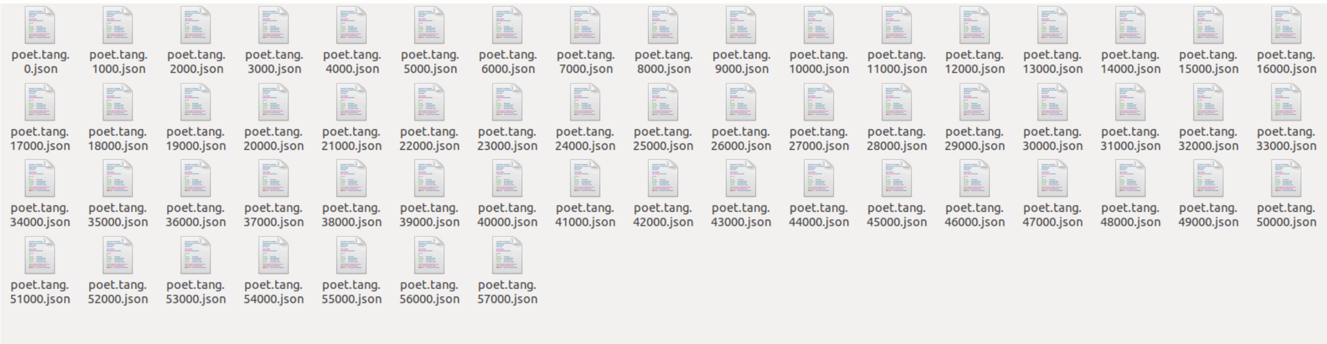
$$
\begin{aligned}
\frac{\partial E}{\partial W_{oht}} &= \frac{\partial E}{\partial net_{ot}} \frac{\partial net_{ot}}{\partial W_{oxt}} = \delta_{ot} h_{t-1}^T & \frac{\partial E}{\partial W_{oxt}} &= \delta_{ot} x_t^T \\
\frac{\partial E}{\partial W_{fht}} &= \frac{\partial E}{\partial net_{ft}} \frac{\partial net_{ft}}{\partial W_{fht}} = \delta_{ft} h_{t-1}^T & \frac{\partial E}{\partial W_{fxt}} &= \delta_{ft} x_t^T \\
\frac{\partial E}{\partial W_{iht}} &= \frac{\partial E}{\partial net_{it}} \frac{\partial net_{it}}{\partial W_{iht}} = \delta_{ot} h_{t-1}^T & \frac{\partial E}{\partial W_{ixt}} &= \delta_{it} x_t^T \\
\frac{\partial E}{\partial W_{cht}} &= \frac{\partial E}{\partial net_{\tilde{c}t}} \frac{\partial net_{\tilde{c}t}}{\partial W_{iht}} = \delta_{\tilde{c}t} h_{t-1}^T & \frac{\partial E}{\partial W_{cxt}} &= \delta_{\tilde{c}t} x_t^T
\end{aligned}
$$

而对于偏置项的求导即为每个时刻的误差项。

Part2 唐诗生成

1. 数据集预处理

1.1 提取全唐诗



```
[
  {
    "author": "太宗皇帝",
    "paragraphs": [
      "秦川雄帝宅，函谷壯皇居。",
      "綺殿千尋起，離宮百雉餘。",
      "連甍遙接漢，飛觀迥凌虛。",
      "雲日隱層闕，風煙出綺疎。"
    ],
    "strains": [
      "平平平仄仄，平仄仄平平。",
      "仄仄平平仄，平平仄仄平。",
      "平平平仄仄，平仄仄平平。",
      "平仄仄平仄，平平仄仄平。"
    ],
    "title": "帝京篇十首 一"
  },
  {
    "author": "太宗皇帝",
    "paragraphs": [
      "巖廊罷機務，崇文聊駐輦。",
      "玉匣啓龍圖，金繩披鳳篆。",
      "韋編斷仍續，縹帙舒還卷。",
      "對此乃淹留，欹案觀墳典。"
```

一开始尝试使用作业中给的唐诗数据集，但是由于量太少所以效果不好，因此考虑使用**全唐诗**，总共57000首古诗，将其提取:

```
# 根据json文件的名字，遍历所有的唐诗文件，并使用json模块提取
for i in range(0, 58000, 1000):
    file = "json/poet.tang." + str(i)+".json"
        with open(file, 'r') as load_f:
            load_dict = json.load(load_f)
```

1.2 使用fastNLP获得数据集

看了一下fastNLP的文档，感觉这个库对于数据集预处理帮助挺大的:

```
from fastNLP import Vocabulary, DataSet, Instance
def get_dataset(file):
    # 构建dataset数据集
    dataset = DataSet()
    # 获得数据集
    with open(file, "r") as load_f:
        load_dict = json.load(load_f)
        for poem in load_dict:
```

```
            data = poem['paragraphs']
            # 将诗句加入dataset, dataset中存在形式为
            # [{'peom':"xxxx"}, {"peom":'xxxxx'}]
            dataset.append((Instance(poem=data)))
    # 将数据集划分为8:2, 训练集为8, 扩展集为2
    train_data, dev_data = dataset.split(0.2)
    return train_data, dev_data
```

- 调用fastNLP中的`Instance` 和 `DataSet` 来封装诗句
- 调用DataSet中自带的`split()`函数，将数据集分割成训练集和扩展集

1.3 生成数据字典与词向量

```
def get_vocabulary(train_data, test_data):
    # 构建词表, Vocabulary.add(word)
    vocab = Vocabulary(min_freq=2, unknown='<unk>', padding='<pad>')
    train_data.apply(lambda x: [vocab.add(word) for word in x['poem']])
    vocab.build_vocab()
    # index句子, Vocabulary.to_index(word)
    train_data.apply(lambda x: [vocab.to_index(word) for word in x['poem']],
new_field_name='words')
    test_data.apply(lambda x: [vocab.to_index(word) for word in x['poem']],
new_field_name='words')
    return vocab, train_data, test_data
```

- 通过fastNLP中的vocabulary模块来构建词典，设定每个子至少出现两次才被计入：`min_freq=2`
- 然后看一下生成的词典以及训练集形式：

  ```
  # python console
  >>>print(vocab.word2idx)
  {'<pad>': 0, '<unk>': 1, '不': 2, '人': 3, '山': 4, '一': 5, '風': 6.......}
  >>>print(train_data[0])
  {"poem":'xxxxxxxxx', 'words':[int,int,int,int....]}
  # 即把存储的poem中每个字用index保存, 存在words字段里
  ```

2. 构建LSTM神经网络

ps:由于第一部分已经要求了我们推导每一步的导数公式，因此构建神经网络直接全部使用 numpy手动求导

为了方便以及清晰地实现LSTM，我们需要构建一个神经网络模型，用class进行封装：

```
class LSTM:
    def __init__():
        ........
```

接下来就是实现这个模型的具体步骤。

2.1 初始化参数矩阵

```
    def _init_w(self, shape, input_dim):
        w = np.random.uniform(-np.sqrt(1.0/input_dim), np.sqrt(1.0/input_dim), shape)
        return w

    def _init_wh_wx(self):
        wh = self._init_w((self.hidden_dim, self.hidden_dim), self.hidden_dim)
        wx = self._init_w((self.hidden_dim, self.input_dim), self.input_dim)
        b = self._init_w((self.hidden_dim, 1), self.input_dim)
        return wh, wx, b
```

- 构造函数 `_init_w(shape)` 生成维度为shape的随机矩阵
- 封装函数`_init_wh_wx()`生成每个门所需要的Wh, Wx, b

```
def __init__(self, input_dim, hidden_dim, vocab_dim)
    # input_dim: 词向量维度; hidden_dim:隐藏层以及状态层维度; vocab_dim:词典维度
    self.input_dim = input_dim
    self.hidden_dim = hidden_dim
    self.vocab_dim = vocab_dim
    self.embedding = nn.Embedding(vocab_dim, input_dim)
    #初始化参数矩阵
    self.whi, self.wxi, self.bi = self._init_wh_wx()
    self.whf, self.wxf, self.bf = self._init_wh_wx()
    self.who, self.wxo, self.bo = self._init_wh_wx()
    self.whc, self.wxc, self.bc = self._init_wh_wx()
    self.wy = self._init_w((vocab_dim, hidden_dim), hidden_dim)
    self.by = self._init_w((vocab_dim, 1), hidden_dim)
```

- 在class LSTM init时，初始化所有需要的参数矩阵，在这里要加上最后线性变换为词典长度的参数矩阵wy , by.

2.2 初始化一个序列的状态向量

```
def _init_list(self, shape, length):
    return np.array([np.zeros(shape)]*lenth)

def _init_state(self, T, hist = {'h':0, 'c': 0}):
    gate_shape = (self.hidden_dim, 1)
    output_shape = (self.vocab_dim, 1)
    i = _init_list(gate_shape, T+1) # 输入门
    f = _init_list(gate_shape, T+1) # 遗忘门
    o = _init_list(gate_shape, T+1) # 输出门
    h = _init_list(gate_shape, T+1) # 隐藏层输出
    c = _init_list(gate_shape, T+1) # 单元状态
    c_ = _init_list(gate_shape, T+1)# 当前输入状态 ~c
    y = _init_list(output_shape, T) # 最终输出
    # 将上一次的状态存放在（T+1)时刻
    h[-1] = hist['h']
    c[-1] = hist['c']
```

```
    return {'i': i, 'f': f, 'o': o, 'h': h, 'c': c, 'c_': c_, 'y': y}
```

2.3 前向计算 (forward)

2.3.1 在gate函数中增加对输入x的embedding

```
# 再次更改gate函数，在其中加入对输入x的embedding
def new_gate(self, Wh, Wx, h_pre, x, b, activation):
    tensor_x = self.embedding(Variable(torch.tensor(x)))
    x = tensor_x.tolist()
    x = np.array(x)
    return activation(Wh.dot(h_pre) + Wx.dot(x.reshape(-1, 1)) + b)
```

- 回顾所有需要用到的计算函数:

```
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def tanh(x):
    return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))

def softmax(x):
    x = np.exp(x)
    return x/np.sum(x)
```

- 根据第一部分中所写的，每个状态变量的计算表达式计算各个变量:

```
def forward(self, x, hist={'h':0, 'c': 0}):
    # 向量长度
    T = len(x)
    # 初始化各个状态向量
    state = self._init_state(T, hist)
    for t in range(T):
        # h(t-1), reshape(-1, 1)代表将其转换为列向量
        h_pre = np.array(state['h'][t-1]).reshape(-1, 1)
        # 输入门
        state['i'][t] = new_gate(self.whi, self.wxi, self.bi, h_pre, x[t], sigmoid)
        # 遗忘门
        state['f'][t] = new_gate(self.whf, self.wxf, self.bf, h_pre, x[t], sigmoid)
        # 输出门
        state['o'][t] = new_gate(self.who, self.wxo, self.bo, h_pre, x[t], sigmoid)
        # 输入状态c~
        state['c_'][t] = new_gate(self.whc, self.wxc, self.bc, h_pre, x[t], tanh)
        # 单元状态 ct = ft * c_pre + it * ~ct
        state['c'][t] = state['f'][t]*state['c'][t-1] + state['i'][t] * state['c_'][t]
        # 隐藏层输出
        state['h'][t] = state['o'][t] * tanh(state['c'][t])
        # 最终输出 yt = softmax(self.wy.dot(ht) + self.by)
        state['y'][t] = softmax(self.wy.dot(state['h'][t]) + self.by)
    return state
```

这里也是第一部分要求我们所做的，根据时间序列来计算下一个状态量，返回的是状态字典，其中每一个变量都是这个序列长度的向量。

2.4 梯度更新(反向传播)

- **初始化所有的梯度矩阵**

```
# shape与参数矩阵相同, 值全为0 (np.zeros)
def _init_delta_gate(self):
    dwh = np.zeros((self.hidden_dim, self.hidden_dim))
    dwx = np.zeros((self.hidden_dim, self.input_dim))
    db = np.zeros((self.hidden_dim, 1))
    return dwh, dwx, db
```

- **反向传播**

```
def backward(self, x, y):
    #初始化所有参数矩阵梯度
    dwhi, dwxi, dbi = self._init_delta_gate()
    dwhf, dwxf, dbf = self._init_delta_gate()
    dwho, dwxo, dbo = self._init_delta_gate()
    dwhc, dwxc, dbc = self._init_delta_gate()
    dwy, dby = np.zeros(self.wy.shape), np.zeros(self.by.shape)

    # 初始化 delta_ct, 因为后向传播过程中, 此值需要累加
    delta_ct = np.zeros((self.hidden_dim, 1))

    # 前向计算
    state = self.forward(x)

    # 目标函数(CrossEntryLoss)对输出 y 的偏导数
    delta_y = state['y']
    delta_y[np.arange(len(y)), y] -= 1

    for t in np.arange(len(y))[::-1]:
        # 输出层wy, by的偏导数
        dwy += delta_y[t].dot(state['h'][t].reshape(1, -1))
        dby += delta_y[t]

        # 目标函数对隐藏状态的偏导数
        delta_ht = self.wy.T.dot(delta_y[t])

        # 各个门及状态单元的偏导数
        delta_ot = delta_ht * tanh(state['c'][t]) * state['o'][t] * (1-state['o'][t])
        delta_ct += delta_ht * state['o'][t] * (1-tanh(state['c'][t])**2)
        delta_it = delta_ct * state['c_'][t] * state['i'][t] * (1-state['i'][t])
        delta_ft = delta_ct * state['c'][t-1] * state['f'][t] * (1-state['f'][t])
        delta_c_t = delta_ct * state['i'][t] * (1-state['c_'][t]**2)

        # 计算各权重矩阵的偏导数
```

```
            dwhf, dwxf, dbf = self.delta_grad(dwhf, dwxf, dbf, delta_ft, state['h'][t-1],
x[t])
            dwhi, dwxi, dbi = self.delta_grad(dwhi, dwxi, dbi, delta_it, state['h'][t-1],
x[t])
            dwhc, dwxc, dbc = self.delta_grad(dwhc, dwxc, dbc, delta_c_t, state['h'][t-1],
x[t])
            dwho, dwxo, dbo = self.delta_grad(dwho, dwxo, dbo, delta_ot, state['h'][t-1],
x[t])

      # 更新权重矩阵
      self.whf, self.wxf, self.bf = self.update_wh_wx(learning_rate, self.whf, self.wxf,
self.bf, dwhf, dwxf, dbf)
      self.whi, self.wxi, self.bi = self.update_wh_wx(learning_rate, self.whi, self.wxi,
self.bi, dwhi, dwxi, dbi)
      self.whc, self.wxa, self.ba = self.update_wh_wx(learning_rate, self.whc, self.wxc,
self.bc, dwhc, dwxc, dbc)
      self.who, self.wxo, self.bo = self.update_wh_wx(learning_rate, self.who, self.wxo,
self.bo, dwho, dwxo, dbo)
      self.wy, self.by = self.wy - learning_rate * dwy, self.by - learning_rate * dby
```

- 在这里要着重讲一下，我们所定义的损失函数是**交叉熵损失函数**，而这个函数对输出y的求导为:

  $$\frac{\partial E}{\partial yi} = yi - 1$$

  所以`delta_y[np.arange(len(y)), y] -= 1`语句代表的是将计算出的预测值减去目标值（目标)看做词典向量单热点所以只用减去对应位置的1）

- 关于**输出层对于参数矩阵**的求导，即交叉熵损失函数对wy, yb的导数，其中t为目标向量:

  $$\frac{\partial E}{\partial W} = (y - t) * x^T$$

  $$\frac{\partial E}{\partial b} = (y - t)$$

  `dwy += delta_y[t].dot(state['h'][t].reshape(1, -1))`

  `dby += delta_y[t]`

- **目标函数对隐藏层输出的导数**

  $$\delta_t = \frac{\partial E}{\partial h_t} = W_y^T \times \delta_{net_t}$$

  `delta_ht = self.wy.T.dot(delta_o[t])`

- **目标函数对门状态变量的导数**
  - 单元状态:

    $$\delta_{ct} = \delta_t \circ o_t \circ (1 - tanh(c_t)^2)$$

    `delta_ct += delta_ht * state['o'][t] * (1-tanh(state['c'][t])**2)`

  - 输出门:

    $$\frac{\partial E}{\partial net_{ot}} = \delta_{ot} = \delta_t \circ tanh(c_t) \circ (1 - o_t)$$

    `delta_ot = delta_ht * tanh(state['c'][t]) * (1-state['o'][t])`

- 遗忘门:

$$\frac{\partial E}{\partial net_{ft}} = \delta_{ft} = \delta_{ct} \circ c_{t-1} \circ f_t \circ (1 - f_t)$$

`delta_ft = delta_ct * state['c'][t-1] * state['f'][t] * (1 - state['f'][t])`

- 输入门:

$$\frac{\partial E}{\partial net_{it}} = \delta_{it} = \delta_{ct} \circ \tilde{c}_t \circ i_t \circ (1 - i_t)$$

`delta_it = delta_ct * state['c_'][t] * state['i']['t'] * (1 - state['i'][t])`

- 输入状态:

$$\frac{\partial E}{\partial net_{\tilde{c}t}} = \delta_{\tilde{c}t} = \delta_{ct} \circ i_t \circ (1 - \tilde{c}_t^2)$$

`delta_c_t = delta_ct * state['i'][t] * (1 - state['c_'][t]**2)`

- **计算梯度:**

  根据第一部分2.7中的公式,构造更新梯度函数:

```
def delta_grad(self, dwh, dwx, db, delta, h_pre, x):
    dwh += delta * h_pre
    dwx += delta * x
    db += delta
    return dwh, dwx, db
```

```
dwhf, dwxf, dbf = self.delta_grad(dwhf, dwxf, dbf, delta_ft, state['h'][t-1], x[t])
...
...
```

- 更新参数矩阵

```
def update_wh_wx(self, learning_rate, wh, wx, b, dwh, dwx, db):
    wh -= learning_rate * dwh
    wx -= learning_rate * dwx
    b -= learning_rate * db
    return wh, wx, b
```

2.5 预测与损失

```
def predict(self, x, hist={'h':0, 'c': 0}):
    state = self.forward(x, hist)
    # 取出sortmax后值最大的index
    pre_y = np.argmax(state['y'].reshape(len(x), -1), axis=1)
    hist['h'] = state['h'][len(x)-1]
    hist['c'] = state['c'][len(x)-1]
    return pre_y, hist
```

```python
# 交叉熵损失函数
def loss(self, x, y):
    loss_sum = 0
    for i in range(len(y)):
        state = self.forward(x[i])
        # 将目标y[i]作为index,取出pre_y相应位置的值, 因为只有目标值为1的部分对交叉熵函数才有效
        pre_yi = state['y'][range(len(y[i])), y[i]]
        loss_sum -= np.sum(np.log(pre_yi))
    # 统计所有y中词的个数, 计算平均损失
    N = np.sum([len(yi) for yi in y])
    ave_loss = loss_sum / N
    return ave_loss
```

- hist代表的上一次运行后h 和 c的值，在预测的时候要传入从而获得长短时记忆的效果。

2.6 总结

由上述2.1-2.5，因为用numpy实现代码数量较多，因此为了更加清晰地阐述我的思路，在这里将整个LSTM网络模型的结构进行总结:

```python
class LSTM:
    def __init__(input_dim, hidden_dim, vocab_dim):
        """
        初始化LSTM
        """
    def _init_w(self, shape):
        """
        初始化参数矩阵
        """
    def _init_list(self, shape, length):
        """
        初始化向量
        """
    def _init_state(self, T):
        """
        初始化一个时间序列中的状态向量
        """
    def _init_delta_gate(self):
        """
        初始化梯度
        """
    def forward(self, x):
        """
        前向计算, x为单个序列(即一条诗句)
        """
    def backward(self, x):
        """
        反向传播, 更新梯度
        """
    def predict(self, x):
        """
        预测单个输出(x为一句诗)
```

```
        """
    def loss(self, x, y):
        """
        计算损失, x,y输入规模为batch
        """
```

3. 训练网络

3.1 训练步骤

```
def train(lstm, X, y, learning_rate, epoch_num):
    losses = []
    for epoch in range(epoch_num):
        for i in range(len(y)):
            lstm.backward(X[i], y[i], learning_rate)
        global step
        step += 1
        loss = lstm.loss(X, y)
        print("step"+str(step)+":loss =" + str(loss))
        losses.append(loss)
    return lstm
```

- train函数参数包括lstm对象，训练数据X（一个batch)和对应目标y, 学习率learning_rate, 迭代次数epoch_num
- 对每一句诗调用backward()更新梯度
- 计算单个batch的损失

3.2 模型保存

由于用numpy实现并不能像torch一样可以直接保存模型，所以我考虑训练好之后的lstm内部的所有参数矩阵存储到json文件中（超级大，生成大小超过50M的文件...):

```
def generate_model(lstm, name):
    model = {}
    model['whi'] = lstm.whi.tolist()
    model['wxi'] = lstm.wxi.tolist()
    model['who'] = lstm.who.tolist()
    model['wxo'] = lstm.wxo.tolist()
    model['whc'] = lstm.whc.tolist()
    model['wxc'] = lstm.wxc.tolist()
    model['whf'] = lstm.whf.tolist()
    model['wxf'] = lstm.wxf.tolist()
    model['bf'] = lstm.bf.tolist()
    model['bo'] = lstm.bo.tolist()
    model['bc'] = lstm.bc.tolist()
    model['bi'] = lstm.bi.tolist()
    model['wy'] = lstm.wy.tolist()
    model['by'] = lstm.by.tolist()
    with open('model_' + str(name) + '.json', 'w') as f:
        json.dump(model, f)
        print("生成model")
```

The file is too large: 64.42 MB. Showing a read-only preview of the first 2.56 MB.

This document contains very long lines. Soft wraps were forcibly enabled to improve editor performance.

1 {"whi": [[-0.06813248559872787, -0.001718372479384513, -0.03647011
.015552777701015475, 0.042077258576279884, -0.060769195818068604
0.02242143642881387, -0.08825128031208737, -0.06372094916787929,
-0.09502309780185762, -0.020655390282571247, -0.0336781426200603
0.060404848106415464, 0.05628692609822788, 0.0776898202565213, 0
.010799756716159391, 0.09302211807577834, 0.027176501380937114,
.09338420661265198, -0.09564363070600972, -0.006803363576974594,
-0.01805969964641909, 0.03546679286146961, 0.08887488379559869,
.09153291845229625, 0.0940058030044755, -0.09666247687053088, 0.
.04514035401656006, -0.018076571922993966, -0.03693068721027971,
-0.05686323004040659, -0.0761456070093513, 0.06334585917374355,
-0.06077329780603565, 0.042721889479134766, -0.05849467335547085
-0.058001005573950916, -0.05796428528852652, 0.00557156174549809
-0.0672753760013256, 0.030274819757973033, -0.034636013724943536
-0.030646182416989274, -0.04234531009841581, 0.09075139352343592
0.015175173150993761, -0.07320718937729706, 0.027403195304988824
-0.07587612756342518, -0.0439682257967871, 0.0041847864514697386
-0.07384144755834267, -0.08203048875140152, 0.06762206894522463,
-0.026722154413601852, -0.09171105855771178, -0.0370131023565613

4. 唐诗生成

4.1 载入模型

```
def load_model(file):
    with open(file, 'r') as load_f:
        m = json.load(load_f)
        hidden_dim = len(m['whi'])
        input_dim = len(m['wxi'][0])
        vocab_dim = len(m['by'])
        lstm = LSTM(input_dim, hidden_dim, vocab_dim)
        lstm.whc = np.array(m['whc'])
        lstm.wxc = np.array(m['wxc'])
        lstm.bc = np.array(m['bc'])
        lstm.whf = np.array(m['whf'])
        lstm.wxf = np.array(m['wxf'])
        lstm.bf = np.array(m['bf'])
        lstm.whi = np.array(m['whi'])
        lstm.wxi = np.array(m['wxi'])
        lstm.bi = np.array(m['bi'])
        lstm.who = np.array(m['who'])
        lstm.wxo = np.array(m['wxo'])
        lstm.bo = np.array(m['bo'])
        lstm.wy = np.array(m['wy'])
        lstm.by = np.array(m['by'])
```

```
    return lstm
```

- 将已经保存好的模型(json文件) load 出来， 用于初始化lstm的各个参数矩阵

4.2 循环调用

```
def next_word(lstm, x, hist, vocab):
    # x为单个汉字
    index = vocab.to_index(x)
    pre_y, hist = lstm.predict(index, hist)
    y = vocab.to_word(pre_y[0])
    return y, hist
```

- 输入为lstm模型，前一个汉字，前一个状态，以及词典
- 循环调用生成下一个

Part 3 问题解答

1. 参数矩阵初始化问题

1.1 为什么不能将矩阵初始化为0

如果设定权重为0，则使用了sigmoid和tanh激活函数的隐藏层所有的神经元敏感度和权值梯度都相同，也就是每一个节点更新后和其他节点值相同，一群节点做着相同的计算，所以并没有实现不同节点学习到不同特征的效果。

1.2 用什么方式初始化

使用Xavier initialization，令输入和输出的方差保持一致：

`W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)`

我这里做了一些改变，使用`np.random.uniform`实现正态的分布：

```
    def _init_w(self, shape, input_dim):
        w = np.random.uniform(-np.sqrt(1.0/input_dim), np.sqrt(1.0/input_dim), shape)
        return w
```

2. 优化

2.1 随机梯度下降算法 (SGD)

对于批量梯度、随机梯度、小批量梯度下降，在这里只要改变batch值就可已得到不同的算法，当然如果将batch设为1，会跑得很慢，但是可以有效地收敛到比较好的结果。

2.2 Adagrad

在代码中，还可以使用了Adagrad来进行梯度更新的优化：

```
def update_wh_wx(self, learning_rate, wh, wx, b, dwh, dwx, db, cache_wh):
    cache_wh += dwh ** 2
    wh -= learning_rate * dwh / (np.sqrt(cache_wh) + 2e-7)
```

通过增加cache值来改变学习率

2.3 无脑衰减

最简单的优化办法是对产生的loss进行判断，如果后一次>前一次，则使learning_rate衰减：

```
        if loss > last_loss:
            learning_rate *= 0.5
            print('decrease learning_rate to', learning_rate)
```

使用

1. train.py

1.1 使用方法

使用`argparse`工具来封装整个训练过程，便于改变训练参数

```
def arg():
    parser = argparse.ArgumentParser()
    parser.add_argument("--input_dim", "-i", default=100, type=int)
    parser.add_argument("--hidden_dim", "-l", default=100, type=int)
    parser.add_argument("--embedding_dim", "-e", default=100, type=int)
    parser.add_argument("--epoch_num", "-n", default=5, type=int)
    parser.add_argument("--learning_rate", "-r", default=0.001, type=float)
    parser.add_argument("--output", "-o", default="2019")
    parser.add_argument("--batch", "-b", default=20, type=int)
    args = parser.parse_args()
    return args
```

1.2 示例输入

```
# 默认输入维度(即embedding维度):100，隐藏层维度: 100，迭代次数: 3， 学习率: 0.005， batch:20
# 生成模型的名字为"model_2019.json"
$ python test.py

# 更改参数
$ python test.py -i 200 -l 200 -e 5 -r 0.01 -b 100 -o "new"
```

1.3 示例结果

```
fang@fang-HP-Pavilion-Notebook:~/桌面/LSTM/lstm$ python train.py
step1:loss =8.7346024852155
step2:loss =8.726057612302487
step3:loss =8.713127206135281
step4:loss =8.687962517046687
step5:loss =8.634049771530439
```

2. generate.py

1.1 使用方法

- 使用`argparse`工具获得模型路径

```
def arg():
    parser = argparse.ArgumentParser()
    parser.add_argument("--file", "-f", default="model_29.json")
    args = parser.parse_args()
    return args
```

- 运行过程中提示输入首字以及诗歌格式获得不同形式的诗句:

```
    while True:
        print("请选择: \n"
              "0: 五言\n"
              "1: 七言\n")
        choice = int(input())
        if choice == 0:
            num = 5
        elif choice == 1:
            num = 7
        print("请输入句数: ")
        sent_num = int(input())
        if sent_num > 0:
            print("请输入诗句首字: ")
            start_words = str(input())
            poem = start_words
```

1.2 示例结果

使用训练次数比较小的模型，可以看到，生成的诗句几乎都是频率高的词语的重复：

```
fang@fang-HP-Pavilion-Notebook:~/桌面/LSTM/lstm$ python generate.py
请选择：
0: 五言
1: 七言

1
请输入句数：
4
请输入诗句首字：
春
生成诗句:
 春日不不不不不，不不不不不不不。
 不不不不不不不，不不不不不不不。
 不不不不不不不，不不不不不不不。
 不不不不不不不，不不不不不不不。
```

结果

1. 诗词结果:

为了尽可能展示不同，每个字生成时输入的参数都随机改变了一下（比如句数）

- 日

```
日
生成诗句:
日月照天地，天地无人间。天地有所有，天地无所为。天地有所有，天地无所为。我闻有余者，不用为其伦。一朝不可见，一室无所为。我闻不得道，不得不相求。我今不得意，不得不相求。我亦不得意，不知无所求。我亦不得意，不知无所求。我亦不得意，不知无所求。
```

- 红

```
红
生成诗句:
红颜不得意，白发不相知。一朝不得意，一日无人知。不知何处去，不见一枝新。
```

- 山

```
山
生成诗句:
山水有余波，苍苍无所似。我来有余物，不见无人识。白日无人识，青山有路岐。山川连北阙，山水入南山。山色连山出，山阴出石连。山川连海屿，山鸟入山川。石壁连山色，松花落石桥。石桥通海屿，山石出江村。石壁连山色，松花落石桥。
```

- 夜

```
夜
生成诗句:
夜夜闻风吹，风吹落日斜。风吹一片雨，日落一千年。水汽连山色，山阴入海田。野寺行人少，江村旧客稀。旧友多游客，新诗不见诗。故园无限意，归去在何时。
```

- **湖**

```
湖
生成诗句:
湖上春风起，江南春草生。风流一日别，日暮一年同。旧国多新暇，新年不见春。风尘如有
意，云雨不知春。旧国多新事，新年不见人。风尘如有意，云雨不知春。旧国多新事，新诗
不见诗。相逢不相见，今日又何为。旧国多新事，新诗不见诗。故人今日少，今日在江东。
别后无人识，春风不可留。别离多少事，别后不知春。别后无人识，无言不可寻。
```

- **海**

```
海
生成诗句:
海天色兮天地中，天地遥兮天地长。天地见兮人不见，天地人兮不可见。我不见兮君不见，
君不见兮人不见。我今来兮不得君，我不知兮君不得。
```

- **月**

```
月
生成诗句:
月下西陵水，日落西南望。年年不见君，只是无人识。
```

2. 总结

为了能输出每一句长度一样的唐诗，在输出的时候做了处理，思路就是处理累积到5/7个字才输出标点符合，不然就跳过。

可以看到，生成的效果并不好，没有押韵，而且有大量低频词重复堆砌，可能是因为迭代次数不够的问题，还有用numpy实现的LSTM并没有用torch实现的好......

当然我也用pytorch写了一下，因为代码相对单纯用numpy实现简单了很多，只需要定义参数矩阵，反向传播都不需要自己写，所以这篇报告中并没有贴出代码。（好吧我承认是为了bonus）

写了蛮久的，虽然网上有很多别人的代码，但是用numpy实现的几乎没有能跑成功的（而且代码组织比较冗杂），所以在复现的过程中遇到了很多坑，差点砸电脑......

比如总是出现tanh(x)运算时出现 **RuntimeWarning: invalid value encountered in true_divide**，然后就出现loss=nan .调了很久参数都没用。后来把embedding处x向量放小了一千倍才成功避免：

```
    def new_gate(self, Wh, Wx, b, h_pre, x, activation):
        tensor_x = self.embedding(Variable(tensor(x)))
        x = tensor_x.tolist()
        # x = np.array(x).reshape(-1, 1)
        x = np.array(x).reshape(-1, 1)/1000
        return activation(Wh.dot(h_pre) + Wx.dot(x.reshape(-1, 1)) + b)
```

对于报告，代码确实长所以我思考了很久怎么才能把自己的思路写清楚，然后写公式真的累了累了......

总之虽然效果不是那么好，但总归还是自己实现了一遍，也算是加深了自己对神经网络实现过程的理解吧！