Assignment 3 of PRML

16307130158 李 政

Part 1

Ⅰ. 求解梯度下降（单次）

- 经过前向传播，可以得到

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

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

  $\bar{C}_t = tanh(W_c * [h_{t-1}, x_t] + b_c)) = tanh(W_{ch} * h_{t-1} + W_{cx} * x_t + b_c)$

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

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

  $h_t = o_t * tanh(C_t)$

  其中：

  W_{*h} 是大小为 $(hidden\_dim, hidden\_dim)$ 的矩阵，与上一层输出量h_{t-1} 相乘；

  W_{*x} 是大小为 $(hidden\_dim, embedding\_dim)$ 的矩阵，与当前输入量x_t相乘；

  二者横向连接组成W_*

- 为了求解参数对于h_t的倒数，需要逐步对中间量求导。举例：

  1. $\frac{\partial h_t}{\partial C_t} = o_t * (1 - tanh^2(C_t))$
  2. $\frac{\partial C_t}{\partial f_t} = C_{t-1}$
  3. $\frac{\partial f_t}{\partial W_{fx}} = f_t * (1 - f_t) * x_t$
  4. $\frac{\partial f_h}{\partial W_{fh}} = f_t * (1 - f_t) * h_{t-1}$
  5. $\frac{\partial f_t}{\partial b_f} = f_t * (1 - f_t)$
  6. $\frac{\partial h_t}{\partial o_t} = tanh(C_t)$

  对1、2、3式，由链式法则可得到

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

  对1、2、4式，由链式法则可得到

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

  对1、2、5式，由链式法则可得到

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

  同理，对于其余状态可求偏导：

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

- $\frac{\partial h_t}{\partial W_{oh}} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial W_{oh}} = tanh(C_t) * o_t * (1 - o_t) * h_{t-1}$
- $\frac{\partial h_t}{\partial W_{ox}} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial W_{ox}} = tanh(C_t) * o_t * (1 - o_t) * x_t$
- $\frac{\partial h_t}{\partial b_o} = \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial b_o} = tanh(C_t) * o_t * (1 - o_t)$
- $\frac{\partial h_t}{\partial i_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t} = o_t * \bar{C}_t * (1 - tanh^2(C_t))$
- $\frac{\partial h_t}{\partial C_{t-1}} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial C_{t-1}} = o_t * (1 - tanh^2(C_t)) * f_t$
- $\frac{\partial h_t}{\partial \bar{C}_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial \bar{C}_t} = o_t * i_t * (1 - tanh^2(C_t))$
- $\frac{\partial h_t}{\partial W_{Ch}} = \frac{\partial h_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial W_{Ch}} = o_t * i_t * (1 - tanh^2(C_t)) * \bar{C}_t * (1 - \bar{C}_t) * h_{t-1}$
- $\frac{\partial h_t}{\partial W_{Cx}} = \frac{\partial h_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial W_{Cx}} = o_t * i_t * (1 - tanh^2(C_t)) * \bar{C}_t * (1 - \bar{C}_t) * x_t$
- $\frac{\partial h_t}{\partial b_C} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial b_C} = o_t * i_t * (1 - tanh^2(C_t)) * C_t * (1 - C_t)$
- $\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial h_{t-1}} = o_t * (1 - tanh^2(C_t)) * C_{t-1} * f_t * (1 - f_t) * W_{fh}$
- $\frac{\partial h_t}{\partial x_t} = \frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial x_t} = o_t * (1 - tanh^2(C_t)) * C_{t-1} * f_t * (1 - f_t) * W_{fx}$

Ⅱ. 在时间序列上求解BPTT

- 由于在LSTM的模型中，误差是在前向传播中累加的，那么在沿时间序列反向传播时，仅需要将每个神经元的梯度反向累加即可；
- 在使用softmax交叉熵损失函数作为目标函数的情况下：

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

  其中 y_i' 是实际输出，为one-hot向量。

  $y_t = softmax(W_y * h_t + b_y)$ 为第t个神经元的输出。
- 损失E对于h_t 的偏导：

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

  由于one-hot 向量相当于仅取其中一列元素，也即：

  $\frac{\partial E}{\partial h_t} = (y_t - y_t') * W_y$
- 综合 I 中的公式，由链式法则：

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

  对于细胞状态C_t，有：

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

Part 2

Ⅰ. 如何初始化权重

- 首先，权重不可以全部初始化为0。这是因为，如果权重全部初始化为0，则对于任何输入量x，每个隐藏层的神经元输出都是相同的，每次梯度的更新也是相同的，也即出现了网络的对称性，这样的训练是无意义的（类似于线性模型）；
- 于是，采用随机初始化的方法。以W_{xi}, b_i举例如下：

```
self.wxi = torch.nn.Parameter(torch.rand(embedding_dim, hidden_dim) * np.sqrt(2 /
(embedding_dim + hidden_dim)))
self.bi = nn.Parameter(torch.rand(1, hidden_dim))
```

II. 建立LSTM模型生成唐诗

概览

- **数据集**：**全唐诗**，共40000+首唐诗，来源于网络中收集的数据
    - 采用fastNLP建立数据集字典，词频最低为两次，词典大小为6741；
    - 预处理时，将唐诗中的所有逗号删去，但保留句号（作为学习诗句断句的依据）和可能存在的问号、书名号；
    - 经过多次测试，决定将诗句序列长度定为80，对超过80字的唐诗，取前80个字符；对不足80字的唐诗，在之前补足若干空格直到变为80（在诗句前补足空格是因为，在诗句后补足会浪费序列后半部分的数据）；
- **模型构建**：主要学习了torch，涉及到`torch.nn.Module`，`torch.autograd`，`torch.optim`，`torch.nn.Embedding`
    - 继承torch.nn.Module父类，建立自己的LSTM模型；
    - 手动重定义forward函数，nn.Module会自动生成相应的backward函数求解梯度；
- **训练**：
    - 调用数据预处理，按4：1的比例划分为训练集和测试集；
    - 调用LSTM模型，并将该模型移动至cuda连接的显卡上进行运算；
    - 引入训练集，输入为前79个字符，标准输出为后79个字符。同样将训练集移动至显卡上进行运算；
    - 调用`torch.optim.SGD()`进行梯度运算，在研究优化时，也可以调用`torch.optim`的其他函数；
    - 每次梯度下降，打印一次当前loss以及测试集得到的perplexity。
- **吟诗**：
    - 以七言诗为例，输入开头单个字；
    - 对于其中单个字的生成：找出预测结果中概率最大的前10个字，首先在它们的前5个中随机取出一个字，如果这个字不是特殊字符，则保留；否则将其舍弃，之后在这10个字中再随机挑选一个字，重复该循环直到满足要求；
    - 通过上述行为，直至预测27个字，形成一首七言四句唐诗。
    - 以下是一些结果：

```
C:\Users\admin\AppData\Local\Programs\Python\Python36\python.exe C:/Users
日落天际来山僧      红水清秋风月明      山头有客行人去      夜幕清秋月光辉
一半秋风雪更深      万里何人现离情      江海无穷一两曲      照灼芳辰丽景时
草叶无水尘土遥      忆君莫羡他日后      秋月明天寒日雨      微彩轻惹罗绮帐
相识离心自知身      知今夜思梦中迎      高台空断不知心      寒日凝云独居回


湖水南风流不得      海内三年春水边      月华不在山河畔
万壑寒花木下稀      草树黄河一片寒      夜长松门闭竹间
相思梦魂遥断阁      云雨中有馀杭天      烟水深思无路傍
来路难忘乱世依      下山归路长相见      西林秋色如云里

Process finished with exit code 0
```

超参数

- Vocabulary size, $|V| = 6741$
- Batch size, $bs = 64$
- Sentence length, $sl = 79$
- Hidden size, $hs = 128$
- Input size, $is = 128$
- Learning rate, $lr = 0.01$

Ⅲ. 优化方法探究

本次实验主要探究了下面4个方法:

- **SGD**

  SGD是最基础的优化方法，即把整套数据集分为若干批数据投入网络中训练。虽然每批数据不能反映整体的情况，但是相比把整套数据进行训练，SGD的速度会加快很多，而且并不会损失很多准确度。

- **SGD with Momentum**

  即具有惯性的SGD。在梯度下降时，保留上一次梯度下降的一部分趋势。用数学表达式表示$dW_t = \mu * dW_{t-1} + learning_rate * \frac{\partial loss}{\partial W}$。在torch中，可以调用`torch.optim.SGD(momentum = μ)`实现惯性SGD，该模型的好处是可以加速梯度的下降，有时也可避免局部最优（因惯性防止落入局部小凹槽）；

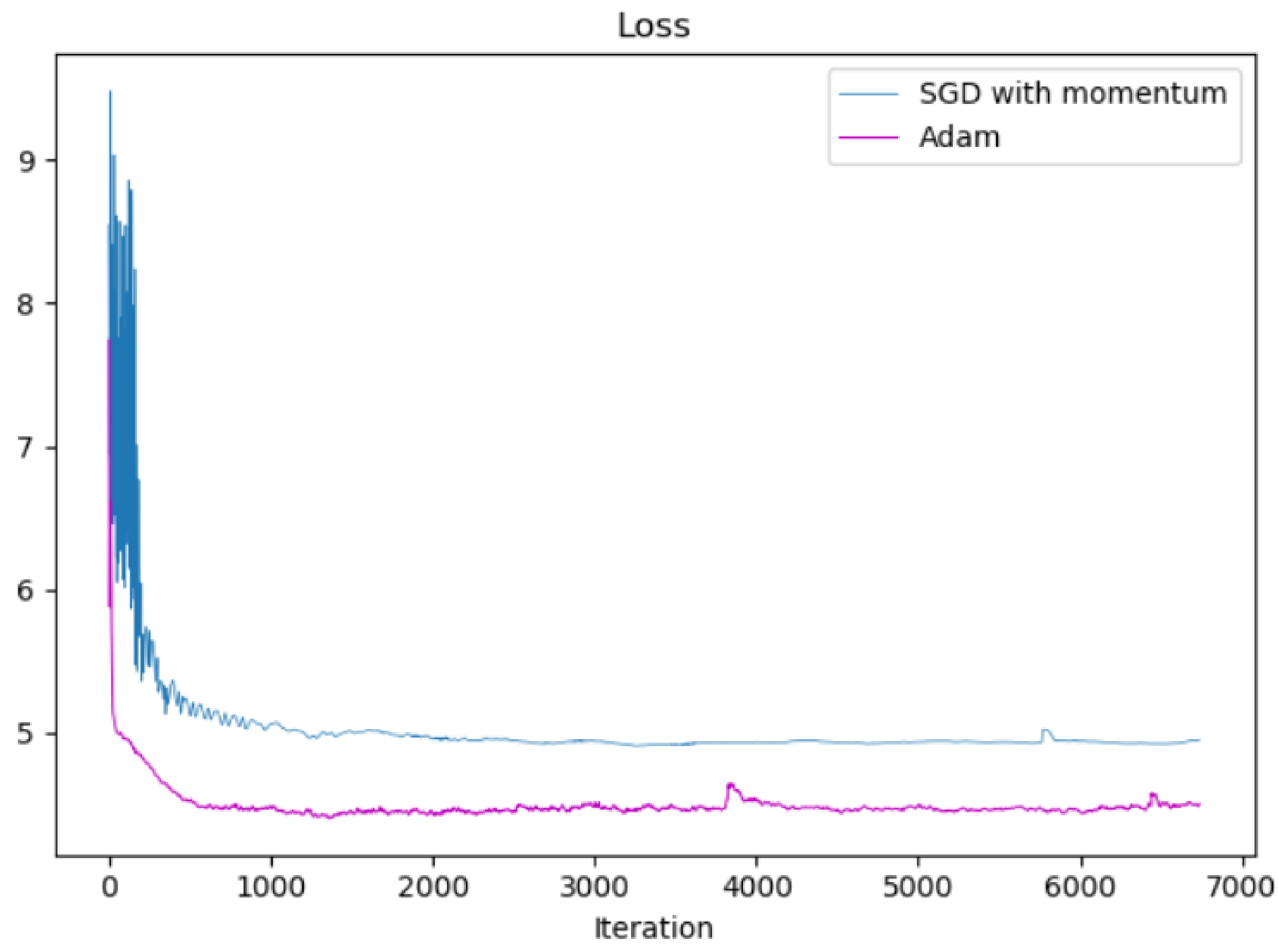
- **AdaGrad**

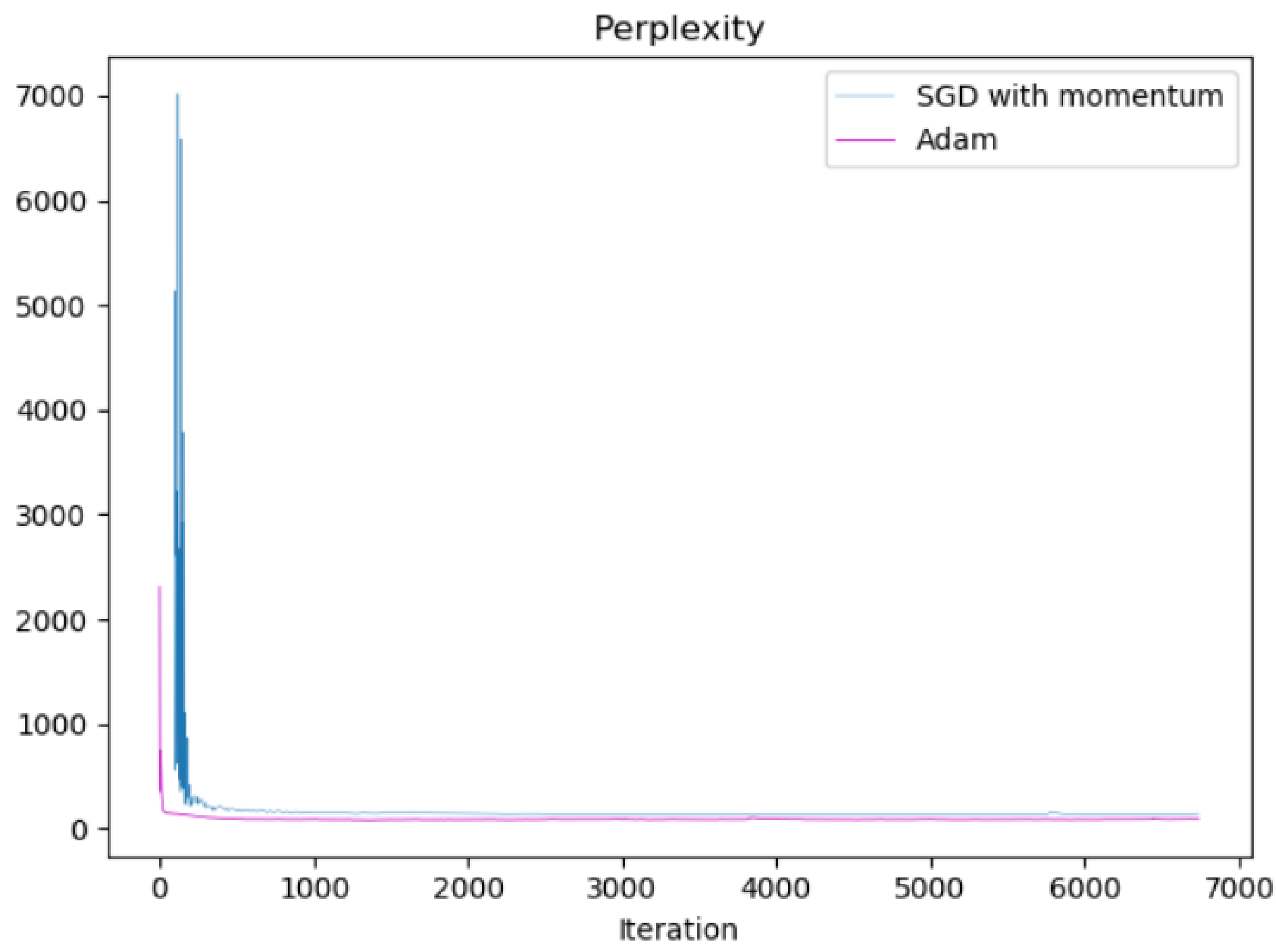
  Adagrad优化方法则是优化学习率，每一次参数的更新都有互不相同的学习率。用数学表达式表示$v+ = dx^2$，$dW = \frac{-learning\_rate}{\sqrt{v}} * dx$。其实是相当于，当梯度下降不准确时，加入惩罚系数修正其下降方向。

- **Adam**

  Adam优化则是结合了Momentum与AdaGrad两种方法，不但能快速下降，又能防止方向不准确，可以既好又快地收敛。

下面以SGD with Momentum和Adam为例，查看具体的效果:


Loss
SGD with momentum
Adam
9
8
7
6
5
0
1000
2000
3000
4000
5000
6000
7000
Iteration

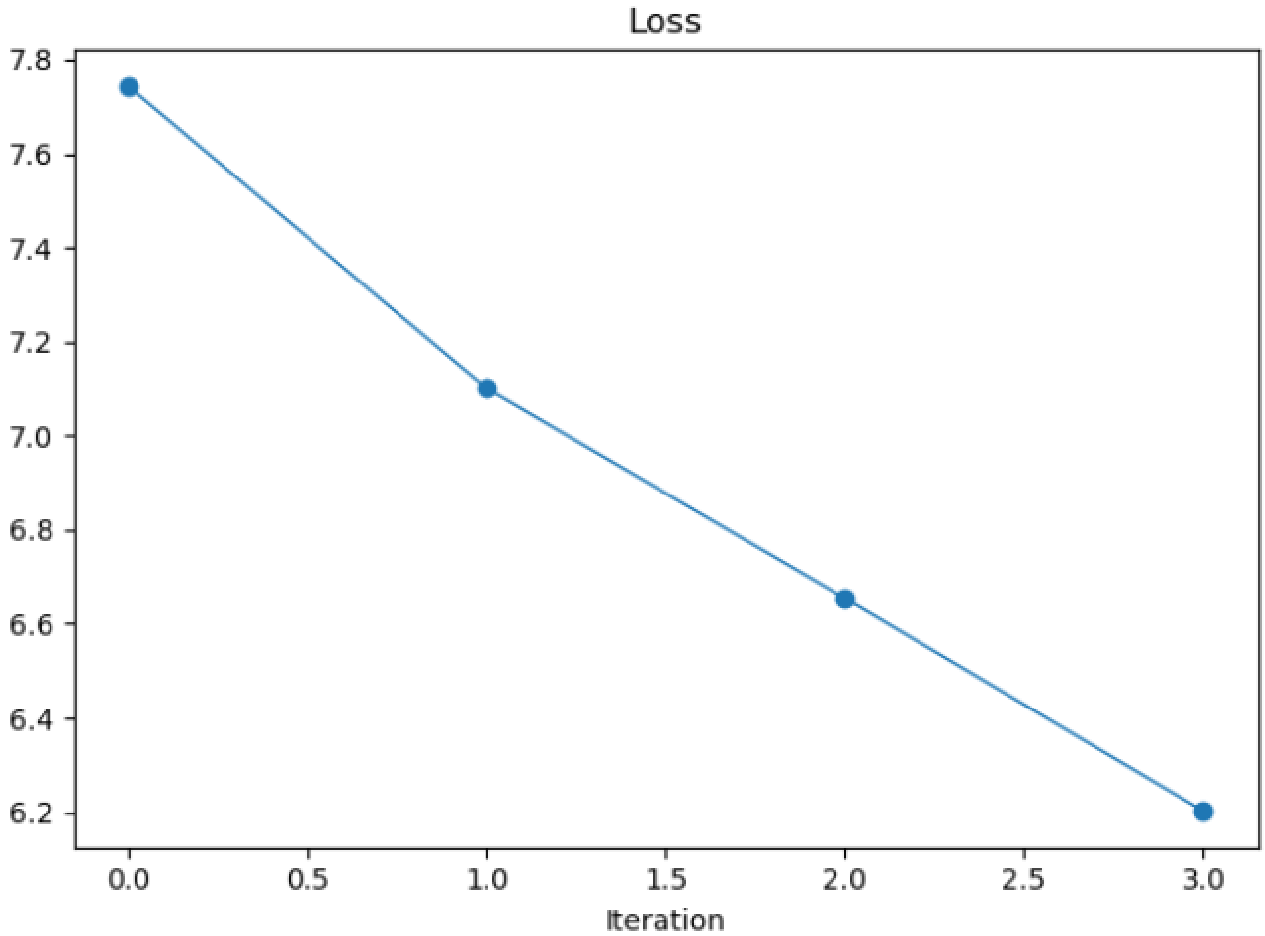

Perplexity
SGD with momentum
Adam
7000
6000
5000
4000
3000
2000
1000
0
0
1000
2000
3000
4000
5000
6000
7000
Iteration

上图是10个epoch（共6740个iteration）中，SGD和Adam的具体实践效果。可以看到，无论是每次梯度下降后的Loss，还是用于检测效果的Perplexity，Adam总是收敛的速度比SGD要快，而且在经过10个epoch后最终达到的收敛值会更小。由此可见，Adam对于SGD的优化是显著的。

Numpy手写梯度下降

在使用torch的自动梯度下降完成了作业后，开始尝试用Numpy手写梯度下降。

主要结合PART Ⅰ中的推导公式，用numpy手写。主干方法是在每一次前向传递，将每个门的状态都记录到一个量中，并在后向传递中手写求导。由于不能像torch那样在显卡上多线程地跑，这个下降速度大概半个小时一次，于是仅有4次下降的结果，和上图相比，结果大致接近：



附上numpy手写LSTM部分的代码（已封装为LSTM类）：

```
from preprocess import build_data_vocab
from LSTM_torch import LSTM_torch as lstm_model
import torch.nn as nn
import torch
from torch.autograd import Variable
import json
import fastNLP
import numpy as np
from fastNLP import DataSet

from fastNLP import Instance
```

```
from fastNLP import Vocabulary
from fastNLP import Trainer
from fastNLP import Tester

def softmax(x):
    x = np.array(x)
    max_x = np.max(x)
    return np.exp(x-max_x) / np.sum(np.exp(x-max_x))

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))

class LSTM:
    def __init__(self, data_dim, hidden_dim):
        self.data_dim = data_dim
        self.hidden_dim = hidden_dim

        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.wha, self.wxa, self.ba = self._init_wh_wx()
        self.wy, self.by = np.random.uniform(-np.sqrt(1.0/self.hidden_dim),
np.sqrt(1.0/self.hidden_dim),
                                             (self.data_dim, self.hidden_dim)), \
                           np.random.uniform(-np.sqrt(1.0/self.hidden_dim),
np.sqrt(1.0/self.hidden_dim),
                                             (self.data_dim, 1))


    def _init_wh_wx(self):
        wh = np.random.uniform(-np.sqrt(1.0 / self.hidden_dim), np.sqrt(1.0 / self.hidden_dim),
                               (self.hidden_dim, self.hidden_dim))
        wx = np.random.uniform(-np.sqrt(1.0 / self.data_dim), np.sqrt(1.0 / self.data_dim),
                               (self.hidden_dim, self.data_dim))
        b = np.random.uniform(-np.sqrt(1.0 / self.data_dim), np.sqrt(1.0 / self.data_dim),
                              (self.hidden_dim, 1))
        return wh, wx, b

    def init_gate(self, T):
        iss = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # input gate
        fss = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # forget gate
        oss = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # output gate
        ass = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # current inputstate
        hss = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # hidden state
        css = np.array([np.zeros((self.hidden_dim, 1))] * (T + 1))  # cell state
        ys = np.array([np.zeros((self.data_dim, 1))] * T)  # output value
        return {'iss': iss, 'fss': fss, 'oss': oss,
                'ass': ass, 'hss': hss, 'css': css,
                'ys': ys}
```

```
    def cal_gate(self, wh, wx, b, ht_pre, x, activation_func):
        e = wh.dot(ht_pre)
        f = wx[:,x]
        res = e + f + b
        return activation_func(res)
        # return activation_func(wh.dot(ht_pre) + wx[:, x] + b)

    def forward(self, x):
        T = len(x)
        stats = self.init_gate(T)

        for t in range(T):
            ht_pre = np.array(stats['hss'][t-1]).reshape(-1, 1)

            # input gate
            stats['iss'][t] = self.cal_gate(self.whi, self.wxi, self.bi, ht_pre, x[t], sigmoid)
            # forget gate
            stats['fss'][t] = self.cal_gate(self.whf, self.wxf, self.bf, ht_pre, x[t], sigmoid)
            # output gate
            stats['oss'][t] = self.cal_gate(self.who, self.wxo, self.bo, ht_pre, x[t], sigmoid)
            # current inputstate
            stats['ass'][t] = self.cal_gate(self.wha, self.wxa, self.ba, ht_pre, x[t], tanh)

            # cell state, ct = ft * ct_pre + it * at
            stats['css'][t] = stats['fss'][t] * stats['css'][t - 1] + stats['iss'][t] *
stats['ass'][t]
            # hidden state, ht = ot * tanh(ct)
            stats['hss'][t] = stats['oss'][t] * tanh(stats['css'][t])

            # output value, yt = softmax(self.wy.dot(ht) + self.by)
            stats['ys'][t] = softmax(self.wy.dot(stats['hss'][t]) + self.by)

        return stats

    def loss(self, x, y):
        cost = 0
        for i in range(len(y)):
            stats = self.forward(x[i])
            pre_yi = stats['ys'][range(len(y[i])), y[i]]
            cost -= np.sum(np.log(pre_yi))

        N = np.sum([len(yi) for yi in y])
        return cost/N

    def init_wh_wx_grad(self):
        dwh = np.zeros(self.whi.shape)
        dwx = np.zeros(self.wxi.shape)
        db = np.zeros(self.bi.shape)

        return dwh, dwx, db

    def cal_grad_delta(self, dwh, dwx, db, delta_net, ht_pre, x):

        dwh += delta_net * ht_pre
```

```
        dwx += delta_net * x
        db += delta_net

        return dwh, dwx, db

    def bptt(self, x, y):
        dwhi, dwxi, dbi = self.init_wh_wx_grad()
        dwhf, dwxf, dbf = self.init_wh_wx_grad()
        dwho, dwxo, dbo = self.init_wh_wx_grad()
        dwha, dwxa, dba = self.init_wh_wx_grad()
        dwy, dby = np.zeros(self.wy.shape), np.zeros(self.by.shape)

        delta_ct = np.zeros((self.hidden_dim, 1))

        stats = self.forward(x)
        delta_o = stats['ys']
        delta_o[np.arange(len(y)), y] -= 1

        for t in np.arange(len(y))[::-1]:

            dwy += delta_o[t].dot(stats['hss'][t].reshape(1, -1))
            dby += delta_o[t]

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


            delta_ot = delta_ht * tanh(stats['css'][t])
            delta_ct += delta_ht * stats['oss'][t] * (1 - tanh(stats['css'][t]) ** 2)
            delta_it = delta_ct * stats['ass'][t]
            delta_ft = delta_ct * stats['css'][t - 1]
            delta_at = delta_ct * stats['iss'][t]

            delta_at_net = delta_at * (1 - stats['ass'][t] ** 2)
            delta_it_net = delta_it * stats['iss'][t] * (1 - stats['iss'][t])
            delta_ft_net = delta_ft * stats['fss'][t] * (1 - stats['fss'][t])
            delta_ot_net = delta_ot * stats['oss'][t] * (1 - stats['oss'][t])


            dwhf, dwxf, dbf = self.cal_grad_delta(dwhf, dwxf, dbf, delta_ft_net, stats['hss'][t
- 1], x[t])
            dwhi, dwxi, dbi = self.cal_grad_delta(dwhi, dwxi, dbi, delta_it_net, stats['hss'][t
- 1], x[t])
            dwha, dwxa, dba = self.cal_grad_delta(dwha, dwxa, dba, delta_at_net, stats['hss'][t
- 1], x[t])
            dwho, dwxo, dbo = self.cal_grad_delta(dwho, dwxo, dbo, delta_ot_net, stats['hss'][t
- 1], x[t])

        return dwhf, dwxf, dbf, dwhi, dwxi, dbi, dwha, dwxa, dba, dwho, dwxo, dbo, dwy, dby


    def sgd(self, x, y, learning_rate):

        dwhf, dwxf, dbf, dwhi, dwxi, dbi, dwha, dwxa, dba, dwho, dwxo, dbo, dwy, dby =
```

```
self.bptt(x, y)
        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.wha, self.wxa, self.ba = self.update_wh_wx(learning_rate, self.wha, self.wxa,
self.ba, dwha, dwxa, dba)
        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

    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

    def train(self, x_train, y_train, learning_rate = 0.005, n_epoch = 5):
        losses = []
        num_examples = 0

        for epoch in range(n_epoch):
            for i in range(len(y_train)):
                self.sgd(x_train[i], y_train[i], learning_rate)
                num_examples += 1

            loss = self.loss(x_train, y_train)
            losses.append(loss)
            print('epoch {0}: loss = {1}'.format(epoch + 1, loss))
            if len(losses) > 1 and losses[-1] > losses[-2]:
                learning_rate *= 0.5
                print('decrease learning_rate to ', learning_rate)
```