

# PRML assignment 3

## PART 1

### 1. 计算LSTM单元的单步梯度

相对于常规的RNN，LSTM增加了保存较长短期记忆的单元状态——Cell State，同时使用更多的门控单元来控制信息的流动。

LSTM用两个门来控制单元状态cell的内容，一个是遗忘门 $\mathbf{f}_t$  (forget gate)，它决定了上一时刻的单元状态有多少保留到当前时刻；另一个是输入门 $\mathbf{i}_t$  (input gate)，它决定了当前时刻网络的输入有多少保存到单元状态。LSTM用输出门 $\mathbf{o}_t$  (output gate) 来控制单元状态有多少输出到LSTM的当前输出值。

LSTM的前向计算过程如下：

$$\begin{aligned}\mathbf{z}_t &= [\mathbf{h}_{t-1}, \mathbf{x}_t] \\ \mathbf{f}_t &= \sigma(W_f \cdot \mathbf{z}_t + \mathbf{b}_f) \\ \mathbf{i}_t &= \sigma(W_i \cdot \mathbf{z}_t + \mathbf{b}_i) \\ \overline{\mathbf{C}}_t &= \tanh(W_c \cdot \mathbf{z}_t + \mathbf{b}_c) \\ \mathbf{C}_t &= \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \overline{\mathbf{C}}_t \\ \mathbf{o}_t &= \sigma(W_o \cdot \mathbf{z}_t + \mathbf{b}_o) \\ \mathbf{h}_t &= \mathbf{o}_t * \tanh(\mathbf{C}_t)\end{aligned}$$

其中 $W_o$ 。（其余权重矩阵同理）：

$$W_o = \begin{bmatrix} W_{oh} \\ W_{ox} \end{bmatrix}$$

为方便表示，令 $\mathbf{net}_{o,t} = W_o \cdot \mathbf{z}_t + \mathbf{b}_o$ ，其余线性层表示方法同理。

为了更方便地推导LSTM的反向传播梯度公式，我们先计算LSTM在单步中产生的梯度公式：

$$\begin{aligned}\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} &= \text{diag}(\tanh(\mathbf{C}_t)) \\ \frac{\partial \mathbf{h}_t}{\partial \mathbf{C}_t} &= \text{diag}(\mathbf{o}_t \odot (1 - \tanh(\mathbf{C}_t)^2)) \\ \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} &= \frac{\partial \mathbf{h}_t}{\partial \mathbf{C}_t} \frac{\partial \mathbf{C}_t}{\partial \mathbf{f}_t} = \text{diag}(\mathbf{o}_t \odot (1 - \tanh(\mathbf{C}_t)^2) \odot \mathbf{C}_{t-1})\end{aligned}$$



$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_f} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{f,t}} \frac{\partial \mathbf{net}_{f,t}}{\partial \mathbf{b}_f} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{f,t}} = \text{diag}(\mathbf{o}_t \odot (1 - \tanh(\mathbf{C}_t)^2) \odot \mathbf{C}_{t-1} \odot \mathbf{f}_t \odot (1 - \mathbf{f}_t))$$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_i} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{i,t}} \frac{\partial \mathbf{net}_{i,t}}{\partial \mathbf{b}_i} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{i,t}} = \text{diag}(\mathbf{o}_t \odot (1 - \tanh(\mathbf{C}_t)^2) \odot \bar{\mathbf{C}}_t \odot \mathbf{i}_t \odot (1 - \mathbf{i}_t))$$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_c} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{\bar{c},t}} \frac{\partial \mathbf{net}_{\bar{c},t}}{\partial \mathbf{b}_c} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{net}_{\bar{c},t}} = \text{diag}(\mathbf{o}_t \odot (1 - \tanh(\mathbf{C}_t)^2) \odot \mathbf{i}_t \odot (1 - \bar{\mathbf{C}}_t^2))$$

## 2. 描述LSTM的反向传播

在训练过程中，我们需要计算Loss对各个参数的导数。通过BPTT算法，从loss和最终时刻T开始，沿时间反向计算梯度（如果是多层LSTM，还要沿层反向计算）。设输入句子长度为T：

1. 由于我们之前已经计算过 $h_t$ 对各个参数的导数，因此现在只要找出 $L$ 与 $h_t$ 之间的导数关系，就能通过链式求导法则求出 $L$ 与各个t时刻参数的偏导数：

$$\text{设 } \hat{y}_t = \text{softmax}(\mathbf{net}_{y,t})$$

$$\mathbf{net}_{y,t} = V \cdot \mathbf{h}_t + \mathbf{b}_q$$

$$L = -\frac{1}{T} \sum_{t=1}^T \sum_{i=1}^N y_{t,i} \log(\hat{y}_{t,i})$$

$$\frac{\partial L}{\partial \mathbf{net}_{y,t}} = \hat{y}_t - y_t$$

$$\frac{\partial L}{\partial \mathbf{h}_t} = \left( \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right)^T \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \left( \frac{\partial \mathbf{net}_{y,t}}{\partial \mathbf{h}_t} \right)^T \frac{\partial L}{\partial \mathbf{net}_{y,t}}$$

在最后时刻T（也是反向传播开始之处）：

$$\frac{\partial L}{\partial \mathbf{h}_T} = \left( \frac{\partial \mathbf{net}_{y,T}}{\partial \mathbf{h}_T} \right)^T \frac{\partial L}{\partial \mathbf{net}_{y,T}}$$

观察前向计算可知，反向传播中 $C_t$ 之前的节点还还有 $C_{t+1}$ ，因此：

$$\frac{\partial L}{\partial \mathbf{C}_t} = \frac{\partial L}{\partial \mathbf{C}_{t+1}} \frac{\partial \mathbf{C}_{t+1}}{\partial \mathbf{C}_t} + \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{C}_t}$$

2. 对于各个W和b，我们知道它们的梯度是各个时刻梯度之和：

$$\frac{\partial L}{\partial \mathbf{W}_o} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_o}$$

$$\frac{\partial L}{\partial \mathbf{W}_f} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_f}$$

$$\frac{\partial L}{\partial \mathbf{W}_i} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_i}$$

$$\frac{\partial L}{\partial \mathbf{W}_c} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_c}$$

$$\frac{\partial L}{\partial \mathbf{b}_o} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_o}$$

$$\frac{\partial L}{\partial \mathbf{b}_f} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_f}$$

$$\frac{\partial L}{\partial \mathbf{b}_i} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_i}$$

$$\frac{\partial L}{\partial \mathbf{b}_c} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_c}$$

3. 对于embedding层的矩阵 $E$ , 其偏导数同理:

$$X = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T]$$

$$X = E \cdot \text{sentence} \quad (\text{sentence为单热点码的序列})$$

$$\text{sentence} = [\text{word1}, \text{word2}, \dots, \text{word}_T]$$

$$\frac{\partial L}{\partial E} = \sum_t \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial E}$$

4. 可轻松计算出全连接层参数的偏导数:

$$\frac{\partial L}{\partial \mathbf{b}_q} = \sum_t \frac{\partial L}{\partial \text{net}_{y,t}}$$

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L}{\partial \text{net}_{y,t}} \mathbf{h}_t^T$$

## PART 2

### 1. 处理数据集与初始化模型

本次实验使用了handout提供的唐诗小数据集，以及来自于《神经网络与深度学习》练习中的唐诗大数据集“poems.txt”（近5万首唐诗）。处理数据集的方法如下：

1. 设定输入单句的最大长度sentence\_length（结尾标识符不计算在内）
2. 逐行检查原始数据集中的诗句（或检查每一首诗），若长度太小（会造成padding过多）则舍去，若长度超过设定最大长度则按最大长度截断。为筛选出的每一个诗句末尾添加结尾标识符E。
3. 使用fastNLP的DataSet和Vocabulary：将所有筛选出的诗句添加进数据集中，根据该数据集生成词表（添加unknown word标识符和padding标识符），对词表中的词作筛选（本实验选取词频>2的所有词汇，控制词表大小在6000以下）

4. 将数据集中的每个诗句的文字映射到词表中的索引，更新数据集。通过在单条数据末尾加padding的方法，将每条数据变为等长（使用fastNLP的Trainer训练时，fastNLP的Batch会帮助我们完成padding）
5. 将数据集按8:2的比例分为训练集和验证集

之后我们需要构建模型并初始化参数。本实验采用的网络结构是“embedding层--激活函数--单层LSTM”，其中LSTM为自己编写，只使用了Pytorch的autograd模块。

在开始训练前我们要对embedding层和权重进行初始化。参数不能全部初始化为0，原因：

1. 初始参数需要破坏神经网络各节点间的对称性。如果初始化参数为0，则对于有相同输入的节点，它们会一直以相同的梯度更新参数，且更新后参数矩阵中也有大量相同的值。随着迭代次数增加，各个参数的值相等，使得网络失去提取不同特征的能力。因此权重不能全部初始化为0。
2. embedding层的初始化与权重初始化同理。若做零初始化，则embedding矩阵中每个向量中的值均相同（每一行或每一列的值相等）。

通常初始化LSTM参数的方法有**高斯分布初始化**，**均匀分布初始化**和**正交初始化**。为避免引入超参，这里直接使用torch.nn默认的方法——**均匀分布初始化**：

对于m个输入，n个输出的全连接层，权重初始化为 $U(-\sqrt{\frac{1}{m}}, \sqrt{\frac{1}{m}})$ ，对偏置采取相同的均匀分布初始化

对于embedding层，采用默认的高斯分布初始化： $N(0, 0.1)$

## 2. 用LSTM生成唐诗

### 2.1 利用FastNLP的Trainer模块和自建trainer函数训练

使用fastNLP的Trainer及其封装的方法，对模型进行训练。我们需要按trainer的要求来定义自己的模型，如模型forward和predict的输出必须是一个词典，且键必须与trainer中的默认设置相同或建立映射。

fastNLP提供了很好的early stop方法，帮助我们自动重新载入在给定评价方法下最好的模型。

```
trainer = Trainer(  
    train_data=train_data,  
    model=lstm_model,  
    loss=CrossEntropyLoss(pred='pred', target='target'),  
    metrics=AccuracyMetric(),  
    n_epochs=50,  
    batch_size=batch_size,  
    print_every=10,  
    validate_every=-1,  
    dev_data=dev_data,  
    use_cuda=True,  
    optimizer=Adam(lr=learning_rate, betas=(0.5, 0.99)),  
    check_code_level=2,  
    metric_key='acc',  
    use_tqdm=False,  
)
```

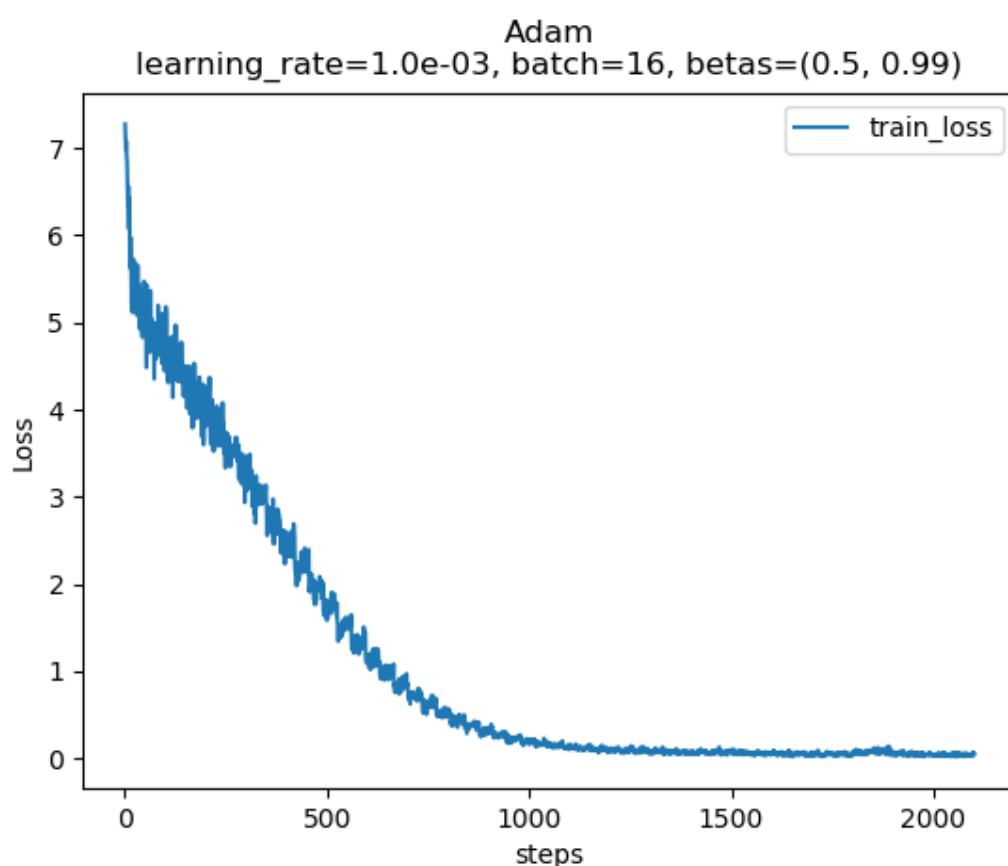
目前trainer.py中封装的优化算法、准确率等接口很少。如果要使用我们所需的metrics方法——perplexity，必须自己建立一个继承fastNLP中MetricBase的类。

出于练习目的，我也使用了自己定义的trainer函数，结合fastNLP中的Batch模块，进行更加个性化的训练。

对于给出的小数据集，超参数设定：

```
Vocabulary size = 1445
Batch size = 16
Sentence length = 25 (包含一个结尾标识符)
Hidden size = 512
Input size = 256
Adam, learning_rate = 0.001, betas=(0.5, 0.99)
```

训练过程：



在给出的小数据集上尝试多次后，发现：

1. 由于数据集过小，如果在数据集中包含标点符号，则模型不能很好地学习到唐诗中标点符号的合适位置。而且生成的诗句往往达不到设定的sentence length。
2. 在小数据集上，模型十分容易过拟合，这体现为在训练过程中，经常出现：模型在训练集上的交叉熵损失下降，但在验证集上的困惑率（perplexity）单调上升。
3. 当模型刚能生成可阅读的诗句时 $perplexity = 525$ ， $loss = 0.6$ 。而在 $loss < 0.03$ 时， $perplexity > 4000$ 。

基本上只会生成数据集中有的诗句，如：

日：日夕故园意，汀洲春草生。何时一杯酒，与倚春风弄。

海：海亭秋日望，委曲见江山。染翰聊题壁，倾壶一解颜。

总之，该简单的模型在小数据集上的诗歌生成表现并不好，很容易过拟合，在训练的后期基本上只会生成内容相近的诗句，缺乏多样性。

## 2.2 更大的数据集

我使用了来自于《神经网络与深度学习》练习中的唐诗大数据集“poems.txt”，这之中大概有近50000首唐诗。处理后得到包含42110首诗的数据集。然后按1.1中的数据集处理方式拆分为训练集和验证集。

[https://github.com/nndl/exercise/blob/master/for\\_chapter\\_6\\_RNN/poems.txt](https://github.com/nndl/exercise/blob/master/for_chapter_6_RNN/poems.txt)

超参数设定：

```
Vocabulary size = 5416
Batch size = 64
Sentence length = 49 (包含一个结尾标识符)
Hidden size = 512
Input size = 512
Adam, learning_rate = 0.001, betas=(0.5, 0.99)
```

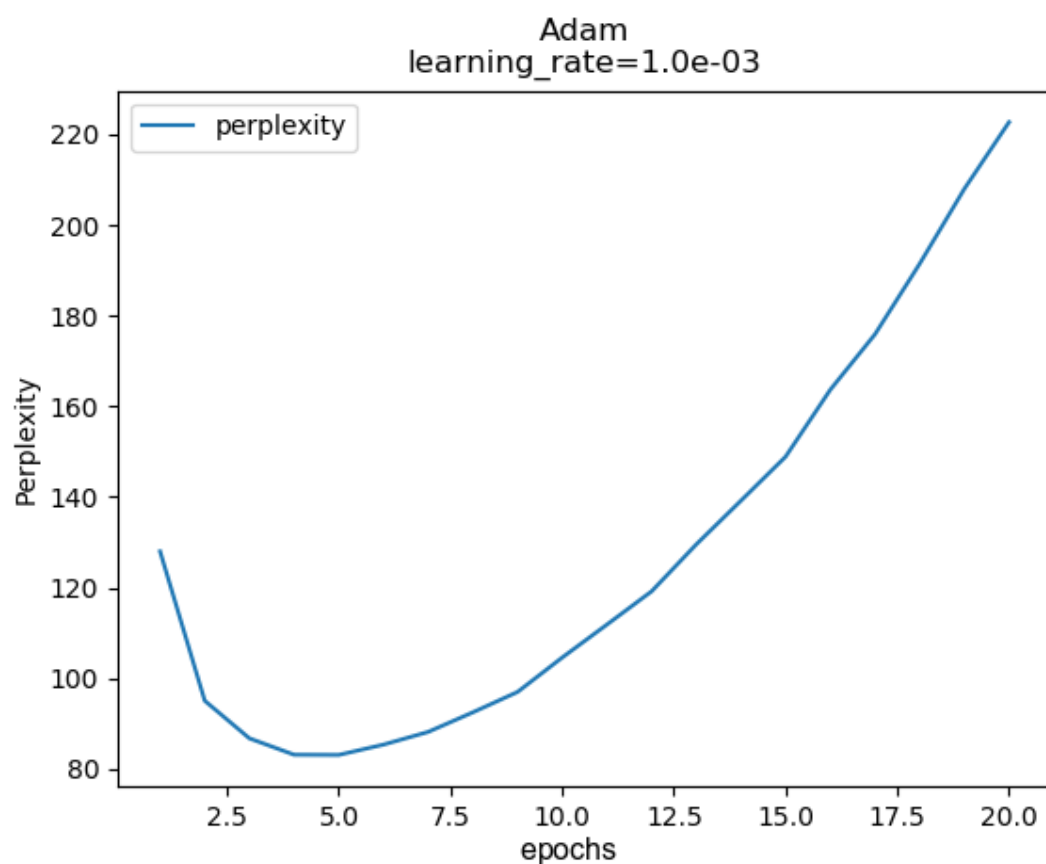
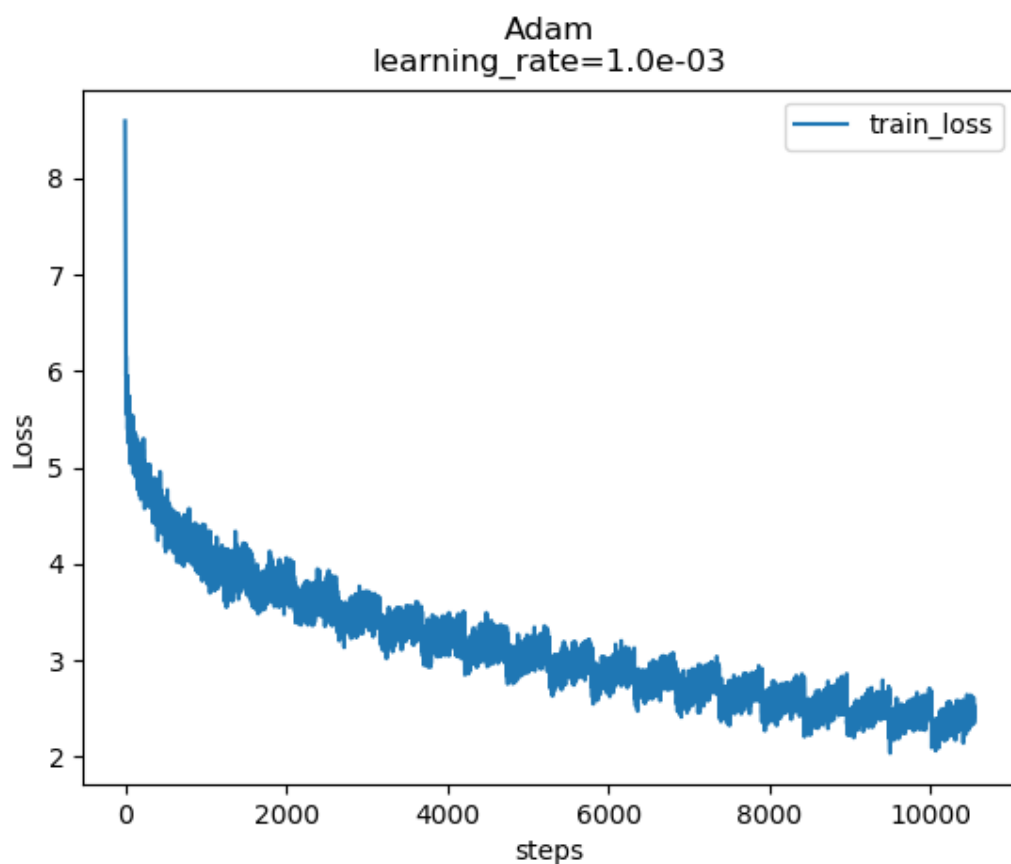
使用GPU在大数据集上训练的注意事项：

1. 在训练时，不涉及到反向传播的代码部分，例如验证、测试部分，请关闭autograd（比如用"with torch.no\_grad():"），否则显存很容易溢出。
2. 在显存允许的情况下可以适当增大Batch，增加并行计算，更快地拟合模型。

在大数据集上，在跑了2个epoch后，模型便能较好地学到标点符号的位置，生成工整的诗句。还能学习到结尾标识符，生成长短不同的诗句。

跑了5个epoch后，模型便在验证集上达到最低的困惑率。

训练过程：



选取完成5个epoch时的模型。

在[0.6, 1.0]之间调整temperature term来使得诗句生成多样化，生成的诗句如下：

*Perplexity* = 87.2, *temperture* = 0.8



日：

日日出门时，登高望远天。云山初出岫，云雨入天山。  
地僻人来久，山深水自闲。何当见山水，相对一相思。

红：

红烛下高楼，楼中望玉京。月明三峡晓，云里一阳台。  
万里无人到，千家万里馀。明年江汉使，应见此时心。

山：

山中有酒熟，此去何时还。独有东山意，无人知此心。

夜：

夜来江上月明时，一夜风吹万里秋。  
一片玉山千万里，一声风雨夜凄凉。

湖：

湖上春风满，春风日日长。绿苔生绿草，红树绿阴阴。

海：

海燕双飞去不回，一双飞去又回头。  
人间有酒唯多病，一夜无心一夜愁。

月：

月出照南山，南山临古城。风烟入户牖，水宿鹭群鸥。  
山色连云远，山川带夕阳。何时归故里，应见此中人。

## 2.3 使用numpy实现LSTM

由part1中关于前向计算和反向传播算法的推导，建立numpy实现的LSTM类。在反向传播中，用循环的方式实现公式中的“累加”，并注意在整个反向传播链上计算loss对 $h_t$ 和 $C_t$ 的偏导数。

随机生成输入，使用数值梯度对反向传播计算出的梯度进行验证。也可以实现pytorch的LSTM方法，给2个模型以相同的初始参数，在相同的随机输入下比较2个模型计算的梯度是否相同。方法包含在lstm\_numpy.py中。

(代码实现参考了 [http://blog.varunajayasiri.com/numpy\\_lstm.html](http://blog.varunajayasiri.com/numpy_lstm.html) )

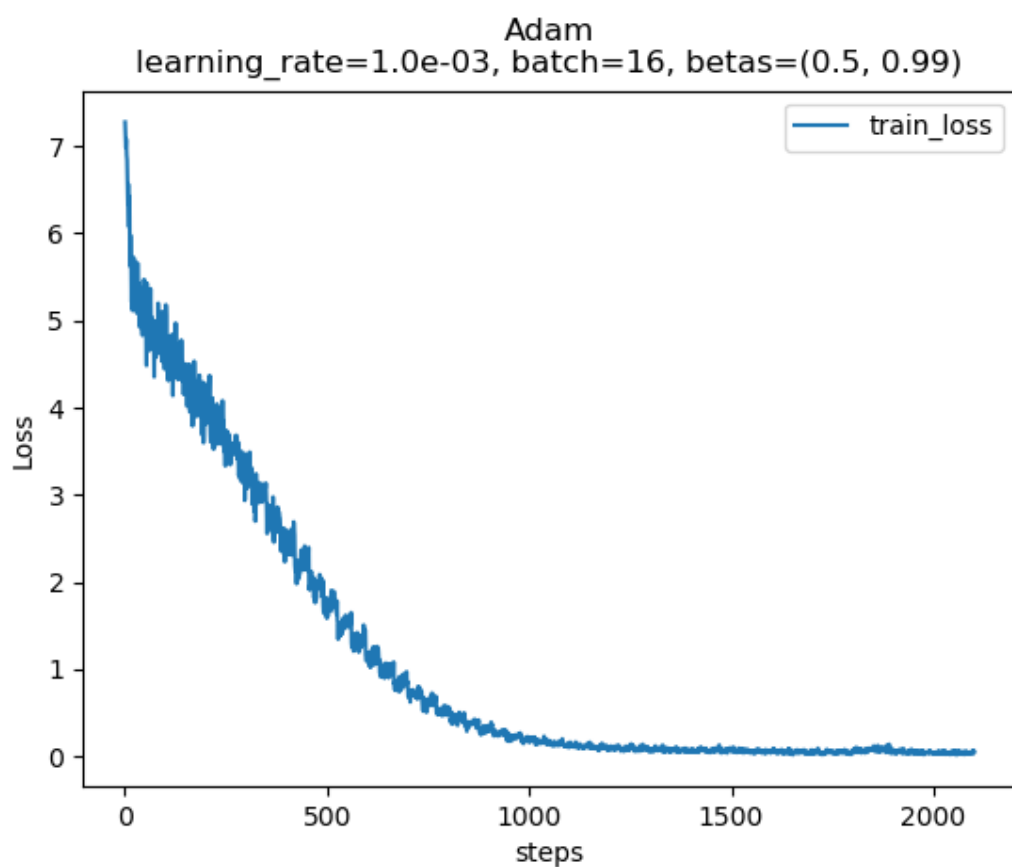
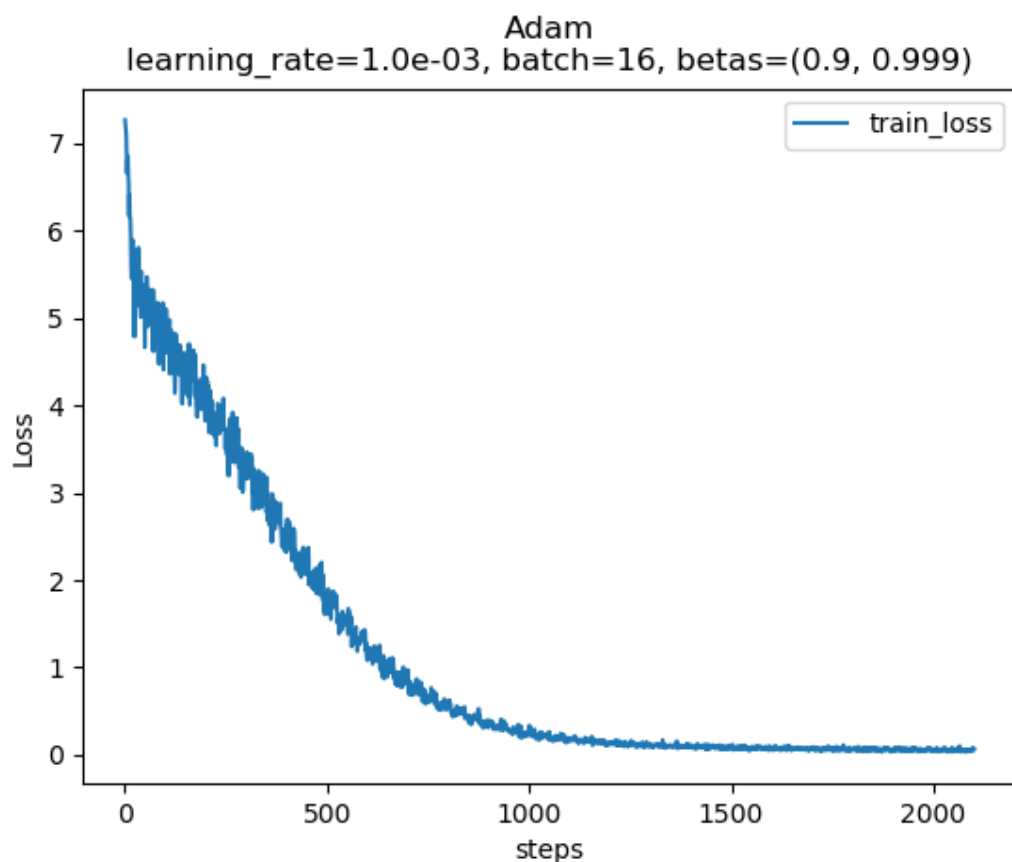
## 3. 梯度下降的优化算法

### 3.1 Adam

Adam算法是一种学习率自适应的优化算法，相当于结合了RMSprop和动量，利用梯度的一阶矩估计和二阶矩估计动态调整参数学习率，且包含偏置修正，使得参数变化比较平稳。Adam算法有默认的超参数设置，但它对超参数的选择鲁棒性也特别高，比如学习率取0.001和0.1，模型参数都不会发散，只是学习率较大时参数更新路径在极小值点附近的环绕区域会更大。一般取学习率为0.001，根据实际情况大致调整就行。

对于betas的选择，默认设置为(0.99, 0.999)。但根据实验，减小第一个超参数，会缩小更新路径在极小值附近的环绕区域；而减小第二个超参数，会使更新路径变得更加曲折。因此很多文献和项目中使用betas=(0.5, 0.99)

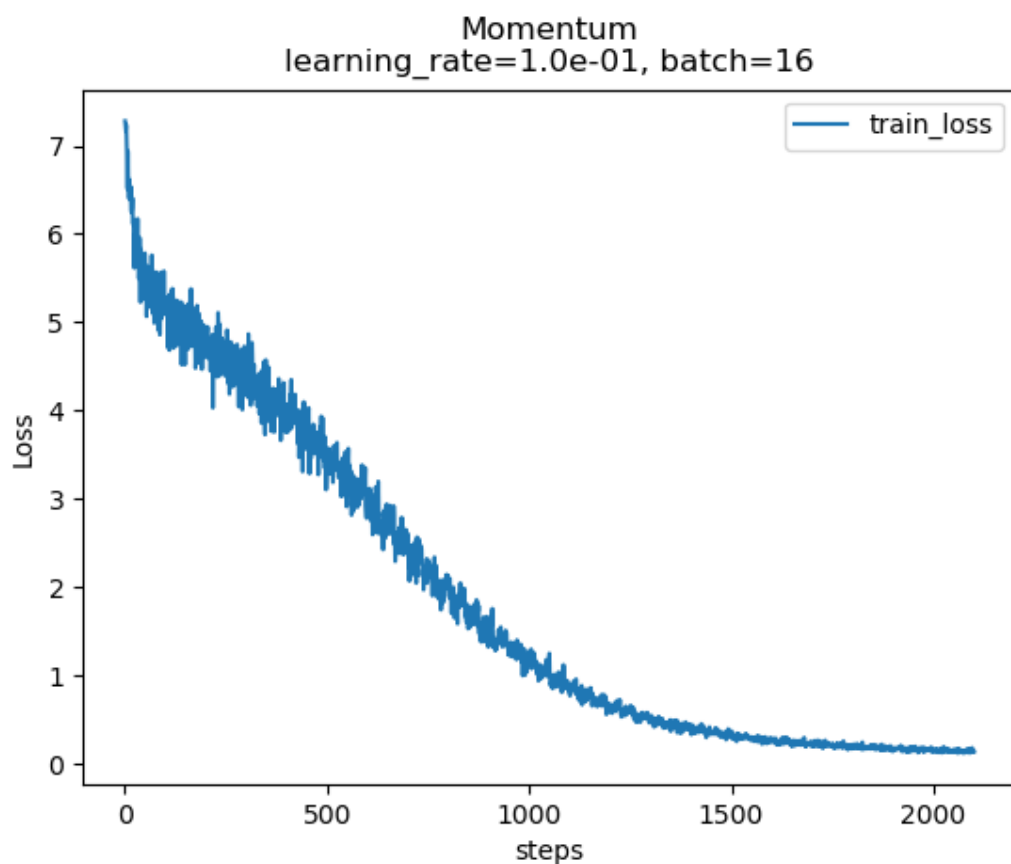
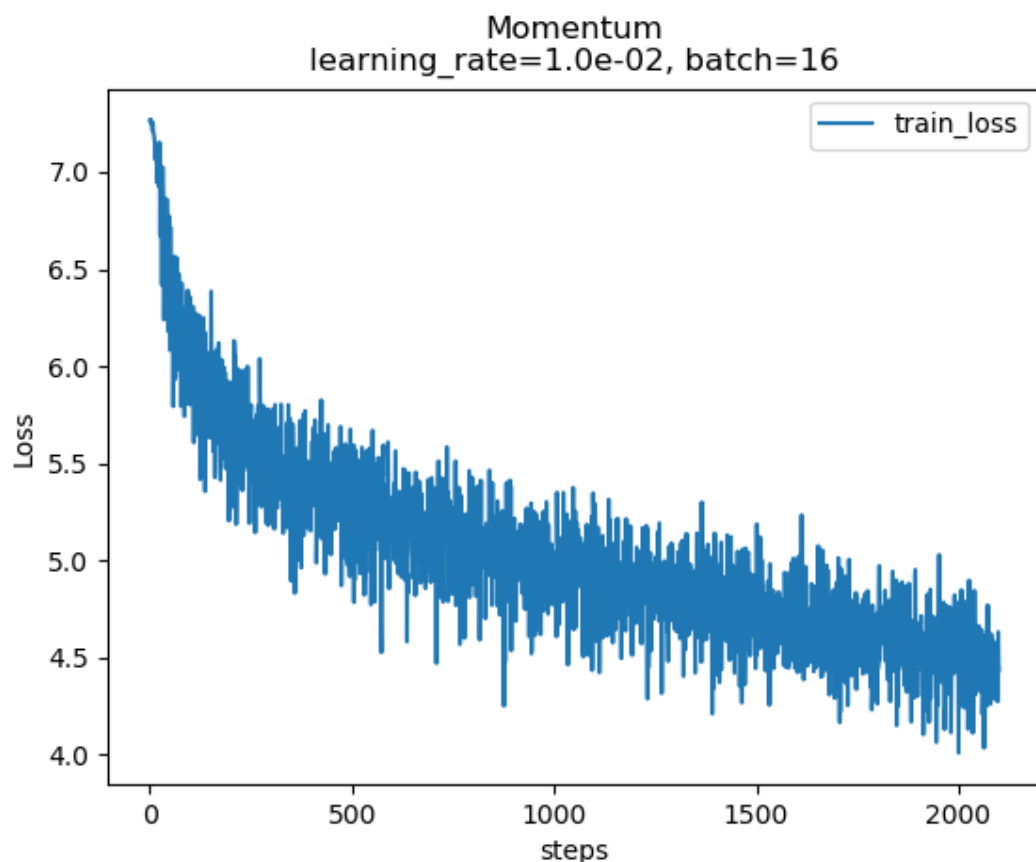
下面给出了2种Adam超参选择方法在小数据集上的loss下降曲线，可见超参的选择对Adam算法影响不大，但观察具体数据可以发现，第二种参数选择方法收敛速度略快且最后得到的loss更小。



### 3.2 带动量的SGD

带动量的SGD算法对学习率的选取要求比较高，相差10倍的学习率，训练效果可能十分不一样（如下2图）

一般的方法是选择较大的动量( $\text{momentum}=0.9$ )，和"小"的学习率( $\text{learning\_rate}\leq 0.1$ ),但这个学习率的选取还是依靠具体情况下的试验。动量也可以在训练过程中调整，往往大动量可以加速收敛。



### 3.3 Adagrad与RMSprop

Adagrad算法通过计算累积梯度平方，自适应地控制学习率，具有大偏导的参数学习率相对较低，具有小偏导的参数学习率较高，因此在参数空间中更平缓的倾斜方向会取得更大的进步。但这样会使得学习率一直衰减，如果在经过一定次数的迭代后依然没有找到最优解，便很难继续找到最优解。

RMSprop算法是Adagrad算法的改进版，它将Adagrad算法中梯度平方的累积计算改为了指数衰减移动平均计算，因此学习率可变小变大，克服了上述缺点。

这2种算法在小数据上的表现与Adam差不多，故不再展示小数据集上训练的损失曲线。

### 3.3 优化方法的numpy代码实现

大部分优化算法在计算一步梯度下降时，往往需要用到之前（所有）的梯度信息，所以需要在每个参数上增加额外的空间，记录之前一步梯度下降的计算信息（比如Adagrad对梯度平方的累加）。

SGD，Adagrad，RMSprop，带动量的SGD，Adam——代码实现如下：

```
# p是参数，p.v是参数值，p.d是梯度，p.m是记录之前梯度下降中的额外计算信息
def update_sgd(self, learning_rate):
    for p in self.all():
        p.v -= learning_rate * p.d

# p.m是累积平方梯度
def update_Adagrad(self, learning_rate, epsilon=1e-8):
    for p in self.all():
        p.m += p.d * p.d
        p.v -= learning_rate * p.d / np.sqrt(p.m + epsilon)

# p.m是加入衰减的改进的累积平方梯度
def update_RMSprop(self, learning_rate, beta, epsilon=1e-8):
    for p in self.all():
        p.m = beta * p.m + (1 - beta) * p.d * p.d
        p.v -= learning_rate * p.d / np.sqrt(p.m + epsilon)

# p.m是速度
def update_sgd_momentum(self, learning_rate, momentum=0.9):
    for p in self.all():
        p.m = momentum * p.m - learning_rate * p.d
        p.v += p.m

# p.g是新增的记录信息，代表Adam中的有偏二阶矩；p.m是有偏一阶矩；p.t用来记录步数
def update_Adam(self, learning_rate, beta1=0.9, beta2=0.999, epsilon=1e-8):
    for p in self.all():
        p.t += 1
        p.m = beta1 * p.m + (1 - beta1) * p.d
        p.g = beta2 * p.g + (1 - beta2) * p.d * p.d
        m_hat = p.m / (1 - np.power(beta1, p.t))
        g_hat = p.g / (1 - np.power(beta2, p.t))
        p.v -= learning_rate * m_hat / np.sqrt(g_hat + epsilon)
```

### 3.4 对5种优化方法的比较

采用以下5种优化方法，在大数据集上进行训练，观察损失下降情况：

Adam: learning rate=1e-3, betas=(0.5, 0.99)

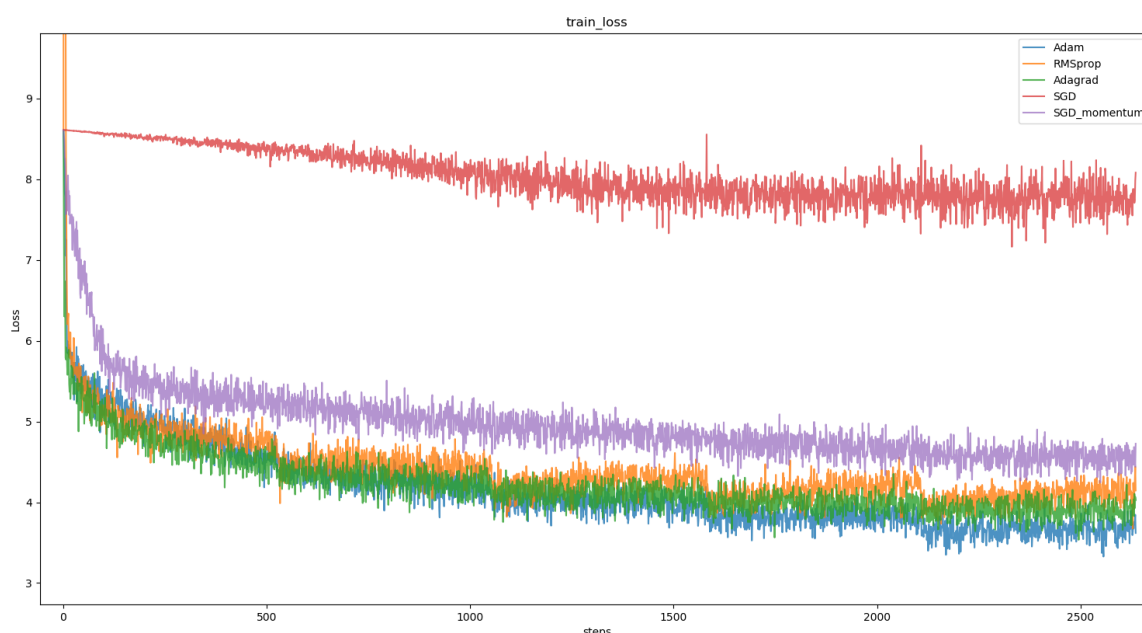
RMSprop: learning rate = 1e-2, alpha=0.99, eps=1e-8

Adagrad: learning rate=1e-2

SGD: learning rate=1e-3

SGD(momentum=0.9): learning rate=1e-1

训练集损失下降如下图所示：



可见：

1. 若固定学习率，则SGD方法收敛速度很慢；而带动量的SGD方法则比普通的SGD方法表现好很多，但仍比不上Adam、RMSprop和Adagrad。使用动量法时，若要取得更好的效果，则需要在训练过程中动态调整学习率和动量大小。
2. 在曲线所演示的前5个epoch中，Adam、RMSprop和Adagrad表现差不多。RMSprop和Adagrad在训练初期会让损失下降得很快。随着迭代次数增加，RMSprop和Adagrad逐渐陷入瓶颈，但Adam损失下降的持久力更强（在3个epochs后Adam得到的损失值便比其他方法都低，且仍在以比其他方法更快的速率下降）。
3. 在上面的实验中，RMSprop优化算法采用的是默认配置。经过调参，在调整学习率为1e-3后，RMSprop的表现变好，其训练的最终损失值与Adam得到的最终损失值基本相同，且初期损失下降更快。RMSprop需要调参反过来也说明Adam算法默认配置的鲁棒性比较高。
4. 我们知道，Adagrad算法的缺点是在经过一定次数的迭代依然没有找到最优点时，由于这时的学习率已经非常小，很难再继续找到最优点。这应该是Adagrad在后期乏力的原因之一。

## 附录

Python文件内容说明：

1. [source.py](#): 载入词表和数据集，训练方法trainer和生成唐诗

2. [lstm.py](#): 仅使用pytorch的autograd模块实现的LSTM模型，以及一个使用nn.LSTM实现的测试用模型
3. [lstm\\_numpy.py](#): numpy实现的LSTM，各种梯度下降优化算法实现，及梯度检测方法
4. [handle\\_poems.py](#): 处理原始数据，生成数据集和词表，保存在当前目录（大数据集和handout提供的小数据集的处理方法都有）