Writing-A-Poem

Here is a project on producing a Tang poem by using LSTM. Although the main target of this assignment is to better understand LSTM and a concise implementation of RNN, I'd like to pay some attention to the poem producing task. That's to say, *How to write a nice poem and how to write a nice poem in a short time?*.

To start with, we have to get the formula of the update of the parameters of the RNN.

Differentiate LSTM

The basic information

First we shall look at the normal update of a hidden state of a normal RNN.

$$h_t = f(h_{t-1}, x_t)$$

In fact, we could write it out in a more detailed way:

$$z_t = Uh_{t-1} + Wx_t + b$$

In the above formula, z_t will be the input of a activation function, x_t is the input of now, and b is a bias term. Based on the knowledge above, we'd look at the long short-term memory model (LSTM). LSTM handles the gradient disappears and explosions. LSTM introduces a new internal state c_t to deliver the information linearly and recurrently.

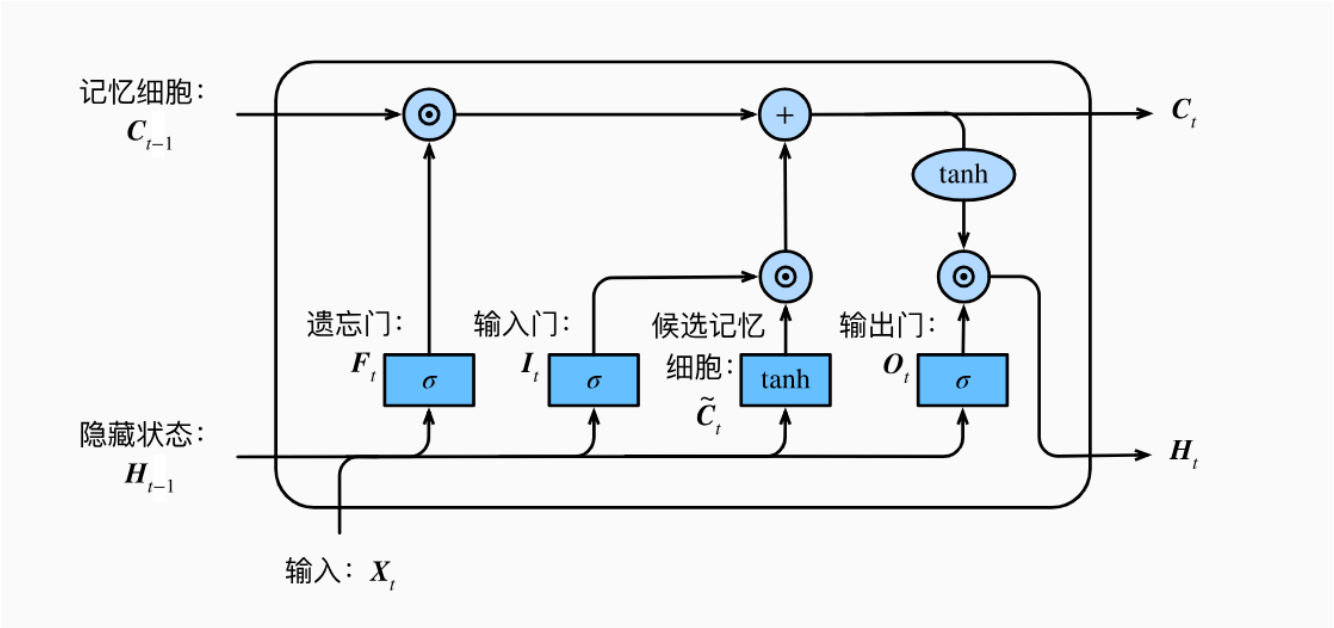
$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \hat{c}_t \\ h_t &= o_t \odot tanh(c_t) \\ \hat{c}_t &= tanh(W_c x_t + U_c h_{t-1} + b_c) \end{aligned}$$

where f_t, i_t, o_t are three gates to control the information flows. f_t, the forget gate, controls how much information RNN forgets. i_t, the input gate, controls how much information now should be kept. o_t, the output gate, controls how much information should be sent to the outside. And these vectors are computed as follows:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \end{aligned}$$

Computational graph and computing

And here we introduce the computational graph. To get the differentiates, we can do it by backpropagation from the top side. We can simply draw a computational graph:



And σ means the logistic function. And we now start to compute the differentiates.

$$\frac{\partial h_t}{\partial f_t} = \frac{\partial h}{\partial tanh(c_t)} \frac{\partial tanh(c_t)}{\partial c_t} \frac{\partial c_t}{\partial f_t}$$

while:

$$\frac{\partial h_t}{\partial (tanh(c_t))} = \mathrm{A}_{D_H \times D_H} = [\alpha^{ii} = o_t^i] \qquad \frac{\partial h_t}{\partial o_t} = \mathrm{A}_{D_H \times D_H} = [\alpha^{ii} = tanh^i(c_t)]$$

$$\frac{\partial tanh(c_t)}{\partial c_t} = \mathrm{A}_{D_H \times D_H} = [\alpha^{ii} = 1 - (tanh)^2(c_t^i)] \qquad \frac{\partial o_t}{\partial sig(\sigma_o)} = \mathrm{A}_{D_H \times D_H} = [\alpha^{ii} = \sigma^i(1 - \sigma^i)]$$

$$Assume \ \ \sigma_o = W_o \eta \qquad \frac{\partial \sigma_o}{\partial W_o} = [\frac{\partial o^i}{\partial W^{mn}} = \eta^n]$$

$$\frac{\partial \sigma_o}{\partial x_t} = [\frac{\partial \sigma_o^i}{\partial x_t^j}] = [W^{ij}] \qquad \frac{\partial \sigma_o}{\partial b} = I$$

$$\frac{\partial c_t}{\partial f_t} = [\alpha^{ii} = c_{t-1}^i] \qquad \frac{\partial c_t}{\partial c_{t-1}} = [\alpha^{ii} = f_t^i]$$

$$\frac{\partial c_t}{\partial i_t} = [\alpha^{ii} = \hat{c}_t^i] \qquad \frac{\partial c_t}{\partial \hat{c}_t^i} = [\alpha^{ii} = i_t]$$

As other terms all have their equivalent form above, we shall leave out these terms for conciseness.

$$
\begin{aligned}
\frac{\partial h_t}{\partial c_t} &= \frac{\partial h_t}{\partial tanh(c_t)} \frac{\partial tanh(c_t)}{\partial c_t} & \frac{\partial h_t}{\partial \sigma_o} &= \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial \sigma_o} \\
\frac{\partial h_t}{\partial W_o} &= \frac{\partial h_t}{\partial \sigma_o} \frac{\partial \sigma_o}{\partial W_o} & \frac{\partial h_t}{\partial U_o} &= \frac{\partial h_t}{\partial \sigma_o} \frac{\partial \sigma_o}{\partial U_o} \\
\frac{\partial h_t}{\partial b_o} &= \frac{\partial h_t}{\partial \sigma_o} \frac{\partial \sigma_o}{\partial b_o} & \frac{\partial h_t}{\partial f_t} &= \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial f_t} \\
\frac{\partial h_t}{\partial c_{t-1}} &= \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial c_{t-1}} & \frac{\partial h_t}{\partial i_t} &= \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial i_t} \\
\frac{\partial h_t}{\partial \hat{c}_t} &= \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial \hat{c}_t} & \frac{\partial h_t}{\partial \sigma_f} &= \frac{\partial h_t}{\partial f_t} \frac{\partial f_t}{\partial \sigma_f} \\
\frac{\partial h_t}{\partial W_f} &= \frac{\partial h_t}{\partial \sigma_f} \frac{\partial \sigma_f}{\partial W_f} & \frac{\partial h_t}{\partial U_f} &= \frac{\partial h_t}{\partial \sigma_f} \frac{\partial \sigma_f}{\partial U_f} \\
\frac{\partial h_t}{\partial b_f} &= \frac{\partial h_t}{\partial \sigma_f} \frac{\partial \sigma_f}{\partial b_f} & \dots
\end{aligned}
$$

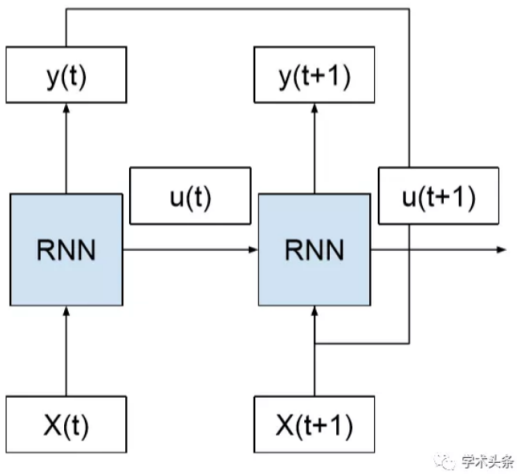
The other terms like U_c, W_c and so on all can be computed by the method above. So, the last we would like to mention is the compute of x and h_{t-1}. They are used in several terms, so we need to add them together to get the answer.

$$
\begin{aligned}
\frac{\partial h_t}{\partial x} &= \sum_{q=o,f,c,i} \frac{\partial h_t}{\partial \sigma_q} \frac{\partial \sigma_q}{\partial x} \\
\frac{\partial h_t}{\partial h_{t-1}} &= \sum_{q=o,f,c,i} \frac{\partial h_t}{\partial \sigma_q} \frac{\partial \sigma_q}{\partial h_{t-1}}
\end{aligned}
$$

Until now, we have all the differentiates settled.

Differentiate through time (BPTT)

BPTT in fact is a compressed FFNN. It could be extended to a long sequence of FFNN in a time sequence. The difference is that they share the same parameters. In a language model, a more reasonable way is to update the parameters through a whole sentence s with the length sl.



First we need to have a loss function, say l. Then the actual update of the parameters p goes:

$$\frac{\partial l_t}{\partial p} = \frac{\partial l_t}{\partial h_t}\frac{\partial h_t}{\partial p}$$

So the loss should be a number which describes the probability of a whole sentence:

$$L = \sum_{t=1}^{sl} l_t$$

So h_i here means the i-th output of LSTM on the sentence sequence.

$$\frac{\partial L}{\partial p} = \sum_{i=1}^{sl}\frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial p}$$
$$\frac{\partial L}{\partial h_i} = \sum_{j=1}^{sl}\frac{\partial l_j}{\partial h_i} = \sum_{j=i}^{sl}\frac{\partial l_j}{\partial h_i}$$

The last equation is valid because the loss in the history has no connection with the present state. For convenience, we mark the last term as L_t. So the target function could be transformed to

$$\frac{\partial L}{\partial p} = \sum_{i=1}^{sl}\frac{\partial L_i}{\partial h_i}\frac{\partial h_i}{\partial p}$$

According to the definition of L_t, $L_t = l_t + L_{t+1}$. Therefore,

$$\frac{\partial L_t}{\partial h_t} = \frac{\partial l_t}{\partial h_t} + \frac{\partial L_{t+1}}{\partial h_t}$$

So, if we have got the L_{t+1} term, we will be able to compute the L_t term. Again, apply the definition of L_t, we know that $\frac{\partial L_{sl}}{\partial h_{sl}} = \frac{\partial l_{sl}}{\partial h_{sl}}$.

$$\frac{\partial L_t}{\partial p} = \frac{\partial L_t}{\partial h_i}\frac{\partial h_i}{\partial p} = \frac{\partial l_t}{\partial h_t}\frac{\partial h_t}{\partial p} + \frac{\partial L_{t+1}}{\partial h_i}\frac{\partial h_i}{\partial p}$$
$$\frac{\partial L_{i+1}}{\partial h_i} = \frac{\partial L_{i+1}}{\partial h_{i+1}}\frac{\partial h_{i+1}}{\partial h_i}$$

The above formula means that if we get any loss grad of the next turn, we could easily get this step's grad, too. Besides, c_t also plays an important role like h_t. Because, c_t affects h_t and h_{t+1} directly:

$$\frac{\partial L_t}{\partial c_t} = \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial L_t}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial c_t}$$

And we have got $\frac{\partial L_t}{\partial h_t}$, $\frac{\partial h_t}{\partial c_t}$, and $\frac{\partial L_t}{\partial h_{h+1}} = \frac{\partial L_{t+1}}{\partial h_{t+1}}$, so this computing is also finished.

Implementations of LSTM

In this assignment, three major models are built:

- Autograd version: Simple use the autograd in the pytorch to implement LSTM
- Model version: Use the nn.Module to wrap a LSTM in a more formal way
- Double LSTM: An implement with two LSTM layers

Data Used and Preprocessing

About the data

In this implementation, I mainly use data from 全唐诗. To make the poems produced more regular (since a lot of poems in the `json` are not regular even to humans.), I picked out poems that have eight lines and 5 characters each sentence. I have used two datasets in this task. A normal one with about 1700 poems in training set and a large one with 10671 poems in the training set. For convenience and to get the data of models with different parameters and settings, I will always use the normal one to do the analysis work if the dataset used is not mentioned.

The dataset of wall is divided into training set and development set with the proportion 3:1.

About dictionary

A dictionary is built to memorize the frequency of each character in poems, and pick out 2000 the most frequent character to form the dictionary. `*, $` are two especial characters added to the dictionary. `*` means the beginning of this sentence, while `$` means the end. The two characters are added to the head and tail of each poem.

And `oov` means a character which shows in a poem but is not collected in the poem. In the dataset, such character occupies about 3.5% of all characters. But such character doesn't really appear in the poem. They are assigned to other characters in the dictionary randomly, which can also serve as masks. Another way is to just resemble all the unknown character as `oov`, but this may do harm to the training process, for the high frequency of `?` leads to wrong predictions.

Word to vector

To map the characters into vector, we need a embedding layer to translate characters from one-hots representations to vectors. Here I use Chinese Word Vectors 中文词向量 as the pretrain set. And the dataset trained by Classical Chinese is picked.

And two versions are built: with fine-tune or without fine-tune. In without fine-tune version, the word vectors are fixed, and the characters that are not in this dataset keep a random vector. And in the fine-tune version, there's a word-embedding layer and it loads the pretrain vectors and allows the net to adjust some vectors.

Structure

In this part, a detailed structure of the basic model and the double LSTM model are shown.

Notations

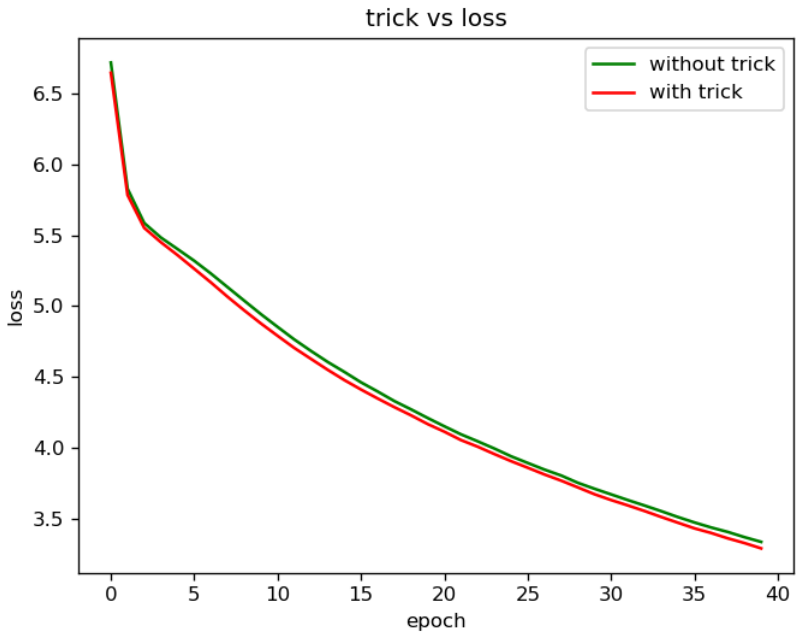
```
D_batch: the size of batch
D_input: the size of input to LSTM, also the output of Word Embedding layer
D_H:     the size of output of LSTM, also the input to the output layer
D_dict:  the size of dictionary, also the size of one-hots
len:     the length of a poem
```

LSTM Cell

LSTM Cell strictly follows the structure described in textbooks, with three gates and a hidden parameter c_t. W, U, b are main parameters keep the memory of the poems. LSTM layer takes word embedding as input and hidden state as output.

About Initializations, they are initialized randomly, but is divided by a term $\sqrt{2/n}$ (n means the number of parameters in the matrix). If these parameters are initialized with a too large number, it causes the $tanh$ always get 1 as the result, making h finally declines to 1, and c gets bigger and bigger. And the term above makes it more mild and solve the problem of sticking to 1. And I happened to find an interesting paper which provide a method to initialize LSTM: set the forget gate to 1 or 2 in the beginning.

And I had a little test on this result on Single LSTM model, and draw the following line with small weight initialization,1 initialization. However, the little trick doesn't show any obvious use in this task... Maybe it's a method that works when the dataset is huge enough?



And of course, they can't be initialized by 0, which leads the all the computing results to 0, and raises error when computing loss function.

PeomProducer

This is the main part of this assignment, which takes poems as dataset to train three layers: Word Embedding layer (if in fine-tune), LSTM layer and the output layer. Of course, for double LSTM, the only different is that there's one more LSTM.

Word Embedding layer

```
input:
one-hot represent:(D_batch, len, D_dict)

output:
embedding represent:(D_batch, len, D_input)
```

Word embedding layer is simply a Linear layer, with pretrained character vectors. The initialization of this layer is relatively free. Random initialization for characters that are not in dictionary is just fine. In all 2000 characters picked out, 1974 characters are hit by the dataset of Chinese Word Vectors.

One thing special is that we need to transform it into simplify Chinese first to get the vector, since most characters in the dataset are in simplify form.

LSTM layer

```
input:
character embedding: (D_batch,len,D_dict)

output:
hidden state:(D_batch,len,D_H)
```

The output of LSTM is surely a sequence of poem, so there's a loop to get the output layer, which is just the use of the differentiates through time that we have discussed above.

The initialization of LSTM layer has been discussed. In fact, I used to just use the random function provided by `pytorch` without realizing that too big initialization results in a disaster. It took me much time to find the grad of parameters in LSTM is always 0, after I have spent hours waiting for a nice poem. And the solution works just fine, which seems to be a nice trick when doing RNN work.

Output layer

```
input:
output of LSTM: (D_batch, len, D_H)

output:
output of this model: (D_batch, len, D_dict)
```

Another Linear layer like Word Embedding layer. The initialization of Output layer can just be random. Of course, 0 initialization results in errors.

Output of this layer will pass through a softmax layer before entering the loss layer. And output of this layer should be just fine to predict a character in dictionary.

Softmax layer And Loss layer

Softmax layer transforms the output of Output layer, and makes the computing of loss legal. Loss layer just compute the loss.

Dropout layer

This was not included in my initial model, but was added after I found the LSTM net began to overfit. In other words, it copies any poem starts with a assigned beginning character in stead of producing a new one. In fact, LSTM is quite easy to be **overfitting**, no matter one-layer or two-layer LSTM, so dropout layer is necessary. In dropout layers, they fixed some parameters to train the model. In this way, a completely copy of a poem will not be a easy task.

Dropout layers are added before the input of LSTM.

A Numpy version

The Numpy version is built according to the BPTT we have discussed before.Since the main topic is to discuss about LSTM, in this implementation, I will leave out the embedding layer to use fixed pre-train word vectors. Although it's a Numpy version, the only difference between the normal LSTM and Numpy LSTM is that the optimizer is changed to a manual version, and the `step()` and `backword()` functions are replaced.

And for a easier implementation of LSTM, I does not use `batch` method to train the model, which slows down the training process. And I do not if it's the way I implement LSTM, the calculation of a turn runs particularly slow, although the decline of the loss is obvious.

And the loss function here if still Cross Entropy, the optimizer I use is our old friend SGD. Although SGD's unfitness to this task, our focus is on LSTM here. SGD will be enough and more explorations of the optimizers will be still finished with the PyTorch version.

Since the version convergences with a dramatically slow way, we will not put it with PyTorch version to compare the convergence speed.

Training

In this part, I will talk more about the details of the three models in the training process.

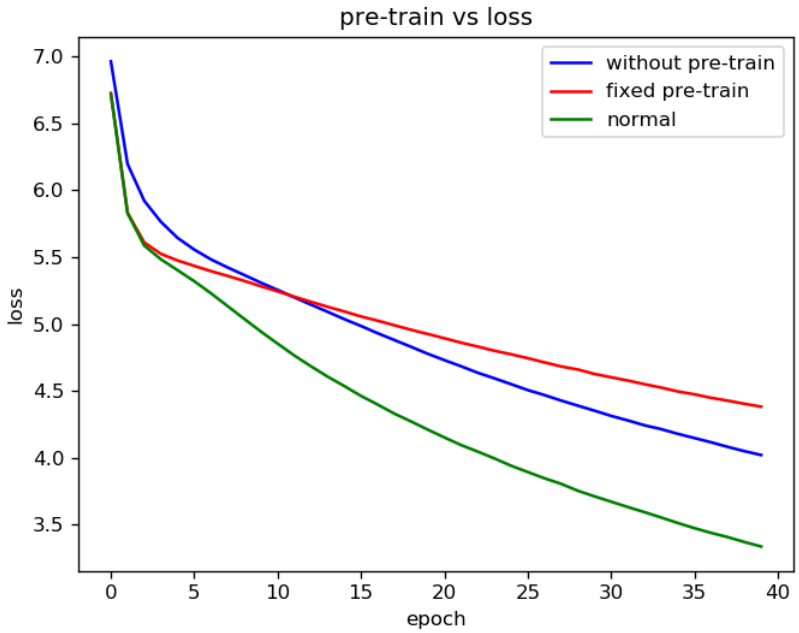
Initialization

The initialization of the shared part of the three models are the same. LSTMs' forget gates are all initialed to small weight. The main difference here is that I choose to use different dropout gate for the three models. Single version is 0.5, double is 0.40, and the triple is 0.35. I altered the values when I find that the convergence speed turns to be too slow for multi-layer LSTMs.

The embedding layers of the three all loaded pretrain results.

Does pre-train work? I want to know how the pre-train set affects the model or its convergence speed. So, a little set is necessary. The test is done on three versions :LSTM with pre-train loaded, one without pre-train, and one with fixed pre-train work.

In fact, the model with pre-train vectors actually works better. The fixed pretrain version's loss drops as fast as the normal one, but gets lower than the without pre-train version. So the ability to fine-tune pre-train is necessary.



Optimization

In this section, we will focus on the optimizer of PoemProducer. Some of the methods will be implemented, while some methods will be used by using the optimizer provided by PyTorch. But I will try my best to figure out how these methods or tricks work.

SGD

SGD has been our old friend. Unfortunately, SGD does not have nice performance on this task. SGD will easily fall in a local optimal and has no way to run out. Besides, SGD can't deal properly with Saddle points, which is common on 2D or 3D problem. So SGD stops convergence easily and in a short time even it's far from the optimal.

SGD with momentum

Momentum is a adaptation of SGD. It gets ideas from the physics, and considers the gradient to the direction of the history will affect the speed of this time. The update of parameters is a combined result of both the history information and this gradient.

$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$
$$W \leftarrow W + V_t$$

So the actual update will somehow keep the original direction which makes it be more possible to run away from the local optimal.

AdaGrad

Intuitively, the learning rate should vary according to the grad. So the basic idea of AdaGrad is to decide the learning rate by its grad. When grad is large, which is usually the case of the beginning of the training, the learning rate should be large. And after a while, learning rate should declines to get the optimal.

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$
$$n = \sum_{r=1}^{t} (\frac{\partial L_r}{\partial W_r})^2$$

n is a parameter that keeps the memory of the history grad. As the turn of the training grows, $\sqrt{n}$ will first declines naturally. ϵ as a term to prevent from $1/0$, and the other part is just the same as SGD.

It solves the problem of the small update in the late training and somehow has the ability to escape from the local optimal.

RMSprop

RMSprop just does a little trick on AdaGrad by adding a new parameter ρ, to balance the n. ρ simply decides how much grad of the history should be kept. See formulas below:

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$
$$n \leftarrow \rho n + (1-\rho)(\frac{\partial L_r}{\partial W_r})^2$$

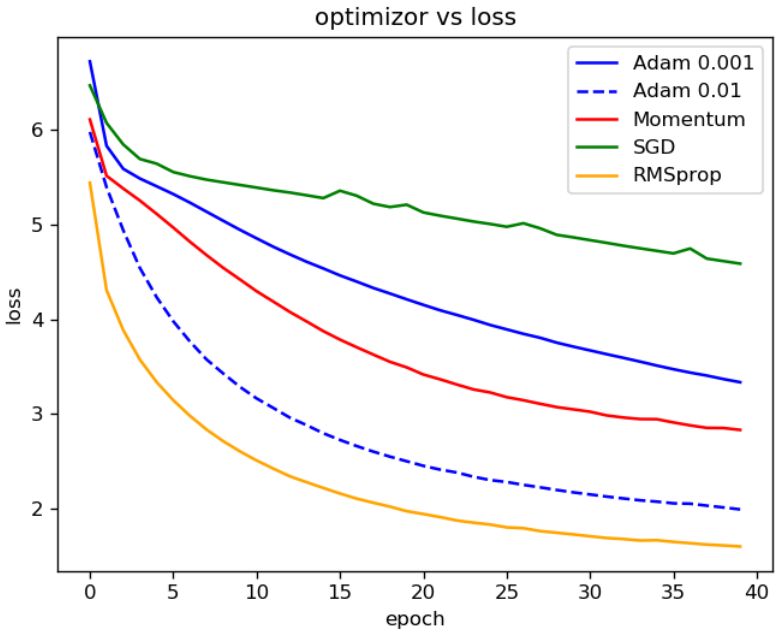
Adam

Adam is the combination of AdaGrad and Momentum. That's to say, it keeps the record of history grad to decides the grad of this update. Besides, it uses the history grad to decide the learning rate.

$$m_t = \beta_t m_{t-1} + (1-\beta_1)\frac{\partial L_t}{\partial W_t}$$
$$v_t = \beta_1 v_{t-1} + (1-\beta_2)(\frac{\partial L_t}{\partial W_t})^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

The last line above is just the combination of Momentum and AdaGrad. η is the learning rate as before. A new term $\hat{v}_t$ is to do the work of Adagrad. β here is the gate which balances the effects of the history information and the grad of this turn.

Performance of these optimizers



Data above uses the same training set, and runs to 40 epochs without checking the perplexity. The only different between Adam 0.001 and Adam 0.01 is their basic learning rate.We could see that SGD simple can't handle the work, while Momentum outperforms the Adam, which is out of expectation. But there's no doubt the RMSprop performs the best in the task. It's stable and fast.

Overfitting

As I have mentioned in *Structure*, overfitting is quite common in LSTM training. And I started to find ways to settle this problem after I found my model was overfitting , too. So, I added two dropout layers for double LSTM, and one for single LSTM.

Besides, I try to double the dataset to prevent overfitting. And of course, early-stopping or L2-regularization is necessary. Because the complicated structure of this model, L2-regularization will also be hard to implement. Therefore, I will choose early-stopping in this task. Thus I will need to add perplexity term to decide whether the train should stop or not.

$$PP(S) = P(s_1 s_2 \dots s_N | LM)^{-1/N} = (\Pi_{i=t}^{N} y_t^{s_t+1})^{-1/N}$$

Perplexity is quite like cross entropy , which we have used to get the loss and gradients.

We could see that as the number of epochs grows, training set and development appear to have the same PPL, and PPL declines obviously. A simple idea to implement *early stop* is to keep a record of the best performance on development set, and give a tolerant number χ. Stop the training when there are consequent χ turns without better performance than the record.

However, I do not support the idea of using perplexity in this task. For one thing, writing poems is a kind of an imitating work, so the poems we write will somewhat like the poems in the training set. But poets's style varies, which means every poet might have different styles. So there's no reason to say that a model is nice because the model can predict poems in development set. In other words, if a model grows to be general enough, it will lose a lot of important message for a poem. The point here is that styles of poems are important, and even more important than the grammars, so there's no reason to use training set as a

judgement of the early stop. An alternative way is to use only one poet's poems as both training set and development set.

But, the method will work well on other less personal-affected tasks like code-generating, for the really important goal is only the correctness, and if it's valid in both training set and development set, it's should be accepted.

In fact, we could raise examples in this task with different perplexity of the training set.

```
*红省三中宅，朱门醉楚阳。不知秋上去，云起更明寒。竹里风馀幌，天云入马微。君怜得此去，空为此时诗。$    ----
made by double LSTM with AvgPPL 466

*红陵风玉里，万里心不然。水色千云雨，水花鸟如时。白水晴上月，清香水遍山。不知时未去，何处不相妨。$    ----
made by double LSTM with AvgPPL 454

*红僻尘埃少，清去在中来。远风千水月，山色落中秋。水色春秋水，山花一秋山。不知无得意，何是此相情。$    ----
made by double LSTM with AvgPPL 277

*红僻清虚地，清花在水花。故风春日客，山日春山云。水色春山月，山山水秋云。不知不可在，何此不相情。$    ----
made by double LSTM with AvgPPL 264
```

In the first two poems, they keep perfect form of the training set and have nice sentences and even a hidden meaning. But the last one just doesn't do this task properly with lots of repeated characters and few meaningful words. And the last is around the turning point of the AvgPPL on the development set. In my opinion, the perplexity decides more on some patterns of rules and words, but does not care about meaning and style, which makes the output disappointing, even it also keeps a form of the poems.

However, we could find that with the AvgPPL a little higher than the turning point will keep a nice result.

The other way to prevent overfitting is to simply enlarge the dataset, which is quite a suitable method in this task, but not suitable for most other tasks with small datasets. So that's where the large dataset works.

Result

In fact, I have totally made three PoemProducer: the single LSTM one, double LSTM one and a triple LSTM one.

Triple LSTM

However, the triple one convergences much lower than the other two. Now that we mainly focus on the LSTM itself, the work to improve the triple layers will just be pushed off. But we could have a look at the poem written by it.

```
*日月南山里，来君去未行。江人千色客，水日故云烟。白色垂山里，乡山带路开。不须无我去，淦路与谁同。$
*何月清门处，江花万里秋。青人不可见，万里见南人。草色新山里，春光带花飞。不知如旧事，归取不相心。$
*夜月南山处，孤山野水流。松根云月月，白色夜云流。白浪寒山月，山山带水云。欲知无未去，无是不相情。$
*海里无何事，家居去故家。初风寒春水，水水向云山。客日春山雨，青山看袅烟。何朝不可见，相是不何情。$
*红水清山处，春山独不情。吹山天月水，草色向云云。白色开山水，山声白水云。思君不可见，骖驾不相间。$
*月月南山里，来君去未行。江人千色客，不是不何情。药穴飞山社，寒寒寒水，何人在已云。应君有有此，不是
*山水清门处，春云万里来。不须无见去，不此不相心。别去无相在，何人在自时。羡君从此去，无是向人情。$

----made by triple LSTM
```

We could notice that most poems keep a well-formed structure. However, one obvious problem is that it uses repeated characters too often. And with unnatural reduplications here and there.

Double LSTM

Double LSTM has a quite training speed, almost the same as the single version.

```
*日月春天月，春山独客归。风光犹复醉，归去去人飞。绿色无人在，风流不可闻。相怜有不见，不是不相情。$
*夜月一花草，青山水县楼。寒色连天色，白月照寒寒。白雪映玉烛，云间鬓紫寒。不知老相见，知君自相情。$
*红泽新绳岸，身汎百日东。东年不见客，复见故人归。白草三湘色，东春不可过。相知心得泪，此事更相情。$
*何处无年事，天心不得归。云开寒月去，山雨碧云寒。白水孤江下，江山欲易归。相知不可见，不见白云吟。$
*山月一江北，东山水县楼。云棹正中色，清香撼绿深。轻撼初犹远，水路夜孤舟。莫识一中人，相期一未知。$
*月色一离在，青山水下深。初阳不可见，回水望三行。四野三新色，青山不弄归。空须聊取去，其以鬓淹丝，$
*海上无为事，长家江上春。山光万里少，春色故人稀。独去风中去，青山草色归。殷勤闲日去，谁更鬓成丝。$

----made by double LSTM
```

Double LSTM shows powerful results much out of my expectations. I even doubt that the ”复见故人归“ was some sentence of some poem. However, it doesn't. “故人归” is a usual combination in Tang Poem. Also see "云开寒月去". What an impressing line! "春色故人稀", "青山草色归" all show great power of the PoemProducer. We can even feel the landscape and emotions of these poems through these simple words. I once thought it was a result of overfitting. But I have searched the Internet and found nothing but single words. It seems that the model combines the single words together to produce wonderful lines.

However, double LSTM shows its weakness of the memory of forms. See the poem begin with "月", it ends with a comma. What's more, double LSTM often get an early '$' just after the third period. Maybe it thought it was just the point to make and end of this poem?

Single LSTM

Final we shall look at the most common version, through which we can feel the power of LSTM the best.

```
*夜月江南寺，山川不可寻。云山云外寺，山水入山村。独夜风前落，秋深夜雨多。何当此惆怅，不得到青山。$
*日暮山川水，江山不可寻。云山不可见，春草又萋萋。莫道青山宅，何人更别离。何当此惆怅，不敢问人时。$
*红粉金闺夜，春风吹鬓丝。春风吹鬓发，春色满春风。别有青云路，春风白露生。莫言新薜萝，不敢问家蔬。$
*山川不可翫，江月照东田。日落风光满，春风落照沙。青山春草绿，江月照东田。莫怪临窗月，何人更此心。$
*月皎南昌尉，山川喜再游。乱山云外落，江月照江村。日暮云连月，山川入翠微。莫言新薜萝，不敢问家蔬。$
*海上春风起，春风满槛时。故山春水阔，江月照东林。草木秋风起，江山月满庭。何时得白头，不是有吾庐。$

----made by single LSTM
```

While training, the single version seems the easiest to be trained, and also the easiest to be overfitting. The turning point is not that obvious. So applying the early stop in the above way might be a good choice.

The worst problem of the single version is that it does not keep a stable form of the poem. It usually show bad shapes even when the model has grown obvious overfitting. So, it seems that it doesn't keep enough information needed or the single version is not strong enough compared to double version. Although simply improve the hidden-size solves this problem, and finally get the poems above. However, this version prefer to use repeated characters and even sentences, although these sentences are not any written poems of the poets.

In large dataset

In the *Overfitting* section, I have mentioned to use larger dataset to prevent the overfitting. And I will only show some of the poems.

```
*日日寻山远，山中有故人。山河连夜雨，江月入江城。野鹤栖书少，山花入夜深。何当一杯酒，不是一相宜。$
*夜来秋色晚，秋色暮霏霏。古道一时老，东山春草生。山川通海上，江月入江城。不见青山外，空馀白发生。$

----made by double LSTM on large dataset
```

However, it's seems hard to judge overfitting.

Conclusions

In the results, the double version performs the best, with the surprising performance and acceptable training set, also the ability to keep from overfitting. While the single version the worst, for it even fails to keep a regular form. Above all, the double LSTM model is considered the best to produce a poem in the regular form of “五言” poem.

A GPU version

There's also a GPU valid version on torch 0.4. But it seems that does not help that much on GTX1050. I only hear the roaring of my computer, **but** the speed of computing doesn't changes. I didn't know what was going wrong...

Besides, this version uses more formal structure of PyTorch. In this version, the batch is implemented by DataSet provided by PyTorch. **But** this change even slows the speed down, which is out of my expectation.

Since it's not much different from the single version, I will not make more comments on this.

Reference

1. Christopher M.Bishop, *Recognizing and Machine Learning*.
2. index
3. 《神经网络与深度学习》
4. LSTM推导参考博客