PRML Assignment3 Report

Part 1

Task 1

The derivation of the differentiation is as followed:

- $\frac{\partial h_t}{\partial o_t} = tanh(C_t)$ $(\because h_t = o_t * tanh(C_t))$
- $\frac{\partial h_t}{\partial C_t} = o_t * \frac{\partial tanh(C_t)}{\partial C_t} = o_t * (1 - tanh^2(C_t))$
- $\frac{\partial h_t}{\partial f_t} = \frac{\partial h_t}{\partial C_t} * \frac{\partial C_t}{\partial f_t} = o_t * (1 - tanh^2(C_t)) * C_{t-1}$
- $\frac{\partial h_t}{\partial i_t} = \frac{\partial h_t}{\partial C_t} * \frac{\partial C_t}{\partial i_t} = o_t * (1 - tanh^2(C_t)) * \bar{C}_t$
- $\frac{\partial h_t}{\partial \bar{C}_t} = \frac{\partial h_t}{\partial C_t} * \frac{\partial C_t}{\partial \bar{C}_t} = o_t * (1 - tanh^2(C_t)) * i_t$
- $\frac{\partial h_t}{\partial C_{t-1}} = \frac{\partial h_t}{\partial C_t} * \frac{\partial C_t}{\partial C_{t-1}} = o_t * (1 - tanh^2(C_t)) * f_t$
- $\frac{\partial h_t}{\partial W_o} = \frac{\partial h_t}{\partial o_t} * \frac{\partial o_t}{\partial W_o} = tanh(C_t) * o_t * (1 - o_t) * z$
- $\frac{\partial h_t}{\partial b_o} = \frac{\partial h_t}{\partial o_t} * \frac{\partial o_t}{\partial b_o} = tanh(C_t) * o_t * (1 - o_t)$
- $\frac{\partial h_t}{\partial W_i} = \frac{\partial h_t}{\partial i_t} * \frac{\partial i_t}{\partial W_i} = o_t * (1 - tanh^2(C_t)) * \bar{C}_t * i_t * (1 - i_t) * z$
- $\frac{\partial h_t}{\partial b_i} = \frac{\partial h_t}{\partial i_t} * \frac{\partial i_t}{\partial b_i} = o_t * (1 - tanh^2(C_t)) * \bar{C}_t * i_t * (1 - i_t)$
- $\frac{\partial h_t}{\partial W_C} = \frac{\partial h_t}{\partial \bar{C}} * \frac{\partial \bar{C}}{\partial W_C} = o_t * (1 - tanh^2(C_t)) * i_t * (1 - \bar{C}^2) * z$
- $\frac{\partial h_t}{\partial b_C} = \frac{\partial h_t}{\partial \bar{C}} * \frac{\partial \bar{C}}{\partial b_C} = o_t * (1 - tanh^2(C_t)) * i_t * (1 - \bar{C}^2)$
- $\frac{\partial h_t}{\partial W_f} = \frac{\partial h_t}{\partial f_t} * \frac{\partial f_t}{\partial W_f} = o_t * (1 - tanh^2(C_t)) * C_{t-1} * f_t * (1 - f_t) * z$
- $\frac{\partial h_t}{\partial b_f} = \frac{\partial h_t}{\partial f_t} * \frac{\partial f_t}{\partial W_f} = o_t * (1 - tanh^2(C_t)) * C_{t-1} * f_t * (1 - f_t)$
- $\frac{\partial h_t}{\partial h_{t-1}}$
  $= \frac{\partial h_t}{\partial f_t} * \frac{\partial f_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial i_t} * \frac{\partial i_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial o_t} * \frac{\partial o_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial \bar{C}_t} * \frac{\partial \bar{C}_t}{\partial h_{t-1}}$
  $= o_t * (1 - tanh^2(C_t)) * C_{t-1} * \frac{\partial f_t}{\partial h_{t-1}} + o_t * (1 - tanh^2(C_t)) * \bar{C}_t * \frac{\partial i_t}{\partial h_{t-1}} + tanh(C_t) * \frac{\partial o_t}{\partial h_{t-1}} + o_t * (1 - tanh^2(C_t)) * i_t * \frac{\partial \bar{C}_t}{\partial h_{t-1}}$
  $= o_t * (1 - tanh^2(C_t)) * (C_{t-1} * f_t * (1 - f_t) * W_h f + \bar{C}_t * i_t * (1 - i_t) * W_h i + i_t * (1 - \bar{C}^2) * W_h C) + tanh(C_t) * o_t * (1 - o_t) * W_h o$
- $\frac{\partial h_t}{\partial x_t}$
  $= \frac{\partial h_t}{\partial f_t} * \frac{\partial f_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial i_t} * \frac{\partial i_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial o_t} * \frac{\partial o_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial \bar{C}_t} * \frac{\partial \bar{C}_t}{\partial h_{t-1}}$
  $= o_t * (1 - tanh^2(C_t)) * C_{t-1} * \frac{\partial f_t}{\partial x_t} + o_t * (1 - tanh^2(C_t)) * \bar{C}_t * \frac{\partial i_t}{\partial x_t} + tanh(C_t) * \frac{\partial o_t}{\partial x_t} +$

$$o_t * (1 - tanh^2(C_t)) * i_t * \frac{\partial \bar{C}_t}{\partial x_t}$$
$$= o_t * (1 - tanh^2(C_t)) * (C_{t-1} * f_t * (1 - f_t) + \bar{C}_t * i_t * (1 - i_t) + i_t * (1 - \bar{C}_t{}^2)) +$$
$$tanh(C_t) * o_t * (1 - o_t)$$

*indicaton: 1. W_h* in the derivation means the submatrix in the W_* which is multiplied by the part made up of h_{t-1} in the z.

2. Note that this derivaion only considers a simple step, so the accumulation of some differentiation such as $\frac{\partial h_t}{\partial W_f}$ is unnessasary.

Task 2

When we are dealing with a sentence, we will get a sequence of inputs, which means the calculation in the lstm cell will iterate and propagate through time. So if we want to calculate a differentiation, we cannot just consider one step. Now we consider the final output h_t (linked with linear transformation). We will get a sequence of $h, h1h2...ht$, whic is the result of each steps. For each h, we can get some differentiation of parameters by the methods mentioned in Task 1. For the parameters of the LSTM, W_* and b_*, the differentiation should be accumulate to get a final result. For instance, For the W_f we have:

$$\frac{\partial h_t}{\partial W_f}_{final} = \sum_{i=1}^{N} \frac{\partial h_i}{\partial W_f}$$

In order to calculate every step in BPTT, we need to record the intermediate variable like f_t in every step. It's worth noticing that C_i remember the cellular state at all times, so its differentiation is also need to be accumulated.
At last, we can get all differentiation of parameters of the LSTM, and then we can use them to update those parameters.

Part 2

Task 1 Initialization

Dataset Because the small dataset had not been uploaded yet when I began to deal with the dataset, I use the "Full Collection of Tang Poems" directly as the dataset and I am too lazy to use the small dataset again. In order to accelerate the training speed, I only use a part of the whole dataset. Then split it to 80% and 20% as train dataset and dev dataset. For each poems, I extract its content without punctuation as a sequence, and then turn the traditional characters to simplified characters. Also I control the length of each sequence to be 80. It means for the sequence with a length lower than 80 I will pad it to 80 with the blank, and for the sequence with a length beyond 80 I will just intercept the first 80 characters of it.

Embedding I use *Vocabulary()* in fastNLP to build the Vocabulary, and then use the Vocabulary to transform the poems sequence to sequences of integer. For the embedding, I use *nn.embedding* in the initialization of my model. Then every time calling forward, the embedding layer will transform every integer in the input sequence to a vector.

initialize weights Why we should not just initialize the weights to 0? There could be many reasons. I think one of them may be that it will make the model equivalent to a linear model. It's obviously that if all the weight is 0, the derivative with respect to loss function will be the same to every element in W_*, and this will also happen in the subsequent iterations[1]. As a result, the hidden units will be symmetric just like a linear model.

Then how to initialize the weights? By looking for some information in the Internet, I know it will be efficiently to use a special heuristic to initialize the weights. If the initialization is totally randomly, you may face a problem that when you pass some value to the activation function like sigmoid function and the value is too big or too small, the gradient descent will be very slow because the sigmoid function will became very smooth when the absolute value of x is too big. So we should give the random a range of values. I use a common heuristic which has a upper bond:

$$\sqrt{\frac{2}{size^{[l-1]} + size^{[l]}}}$$

the $size^{[l]}$ here stands for size of layer l. In this way, we can keep all of the elements of the parameter metrics are stored in the gradient range.

There are also many efficient methods of initialization such as "Standard initialization" and "He initialization" and whatever you change in the heuristic you can get a method.I won't go into too much detail here, and just remenber the point is to keep the value of parameter in the gradient range.

Task 2

After the initialization, I build a model with pytorch and also build another model with numpy. See the code for implementation details. It is worth mentioning that I use Logsoftmax instead of the softmax in the linear layer of lstm, which seems to make the training process quick and stable for it turns the division to a subtraction. And for the numpy version, because it is hard to implement the bp of embedding, I just use a static embedding. Some parameters are set as follows: $learningrate = 1e-2, embeddingsize = 64, hiddensize = 128, batchsize = 32$(I want to train the model faster because there are still much projects left, so I choose not very big parameters, and I think it still work well).

[1]It worth noticing that the bias can be initialized to be zero without any troubles

The loss is change from 6.4 to 4.6, and when the model stop training, the perplexity is about 8.4. The loss of the two models are as followed:

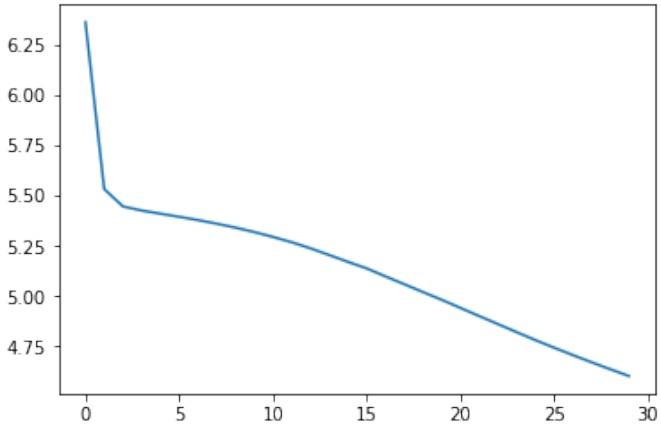

Figure 2.1 loss of the pytorch version

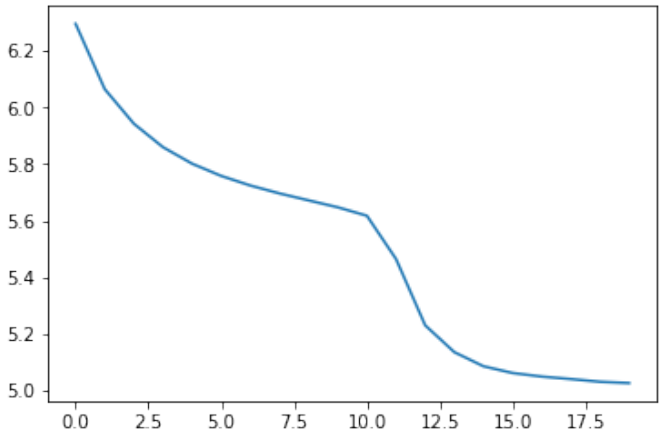

Figure 2.1 loss of the numpy version

It seems that there are some difference between the two figures, because their methods of optimization are different and for the numpy version, I use a smaller dataset to train it. But the result is, both of them have significant gradient descent which means they works well.

The poems generated are as followed(use the pytorch version to generate, and because I remove the punctuation in inputs, the punctuation in the results is added by me rather than generated.):

日： 日出郭乱北，胡衣随燕国。虽见画楼中，白首万卷春。

红： 红旗日暮云雨满，眼前山川客无事。不见衰老翁不见，白头吟青枫树中。

山： 山风动落叶，高堂不可得。老夫不见衰，人不得有人。

夜： 夜年行客有，无穷人生老。不知不知不，敢见汝去无。穷途前后人，老夫不得一。人生不知己，得老病无限。

湖： 湖峡中有无，穷不可知时。危病不敢知，名家老夫名。

海： 海化神女漂，故园衰颜食。敢讬青亭午，宁辞新世路。

月： 月江山城江水悠，柳叶白露满水清。光无复未相对君，老不见乱无穷家。

Task 3

During training, I use Adam and SGD with momentum to optimize. And these two methods and most of the methods mentioned in the instructions need to keep the parameters about the old updating direction. Take momentum as a example, it make the update follow the formula:

$$\Delta x_t = \rho \Delta x_{t-1} - \eta g_t$$

in which the ρ stands for that to what extent should the original direction of update be retained.

In this way, the model can keep the original updating direction, and at the same time do some adjustment according to the current gradient. As a result, the model will avoid totally random gradient descent and its stability can be improved.

I use the Adam algorithm which I think is the one of the best optimizing algorithms. Adam use the estimation of the first moment of gradient and include the Biasing correction is made to estimate the first and second moments initialized from the origin. Its implementation can be represented as:

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{\upsilon_t}} + \epsilon} \hat{m}_t$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\upsilon_t = \beta_2 \upsilon_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - {\beta_1}^2}; \hat{\upsilon}_t = \frac{\upsilon_t}{1 - {\beta_2}^2}$$

For the numpy version, I just write the CrossEntropyLoss and use the MBGD to train the model, it also works well but the training speed will be slower.

Clarification

- In order to accelerate the train, I use about 5000 poems in "Full Collection of Tang Poems" instead of the whole Collection as dataset.
- In order to generate better poems, I will pick up the top-x frequent targets and choose one according to the distribution of the frequency as the next character, so the poems generated will not be constant.