PRML Assignment-3 Report

SOURCE FILE

- `source.py` : Code of generating poems with PyTorch
- `lstm_numpy.py` : LSTM model with numpy

PART 1

QUESTION 1

- **It is easily to prove that,** $diag[\mathbf{a}]X = \mathbf{a} \circ X$ **and** $\mathbf{a}^T diag[\mathbf{b}] = \mathbf{a} \circ \mathbf{b}$**.**
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} = diag[\tanh(\mathbf{c}_t)]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} = diag[\mathbf{o}_t \circ (1 - \tanh(\mathbf{c}_t)^2)]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\frac{\partial \mathbf{c}_t}{\partial \mathbf{f}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} diag[\mathbf{c}_{t-1}]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} diag[\tilde{\mathbf{c}}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \tilde{\mathbf{c}}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\frac{\partial \mathbf{c}_t}{\partial \tilde{\mathbf{c}}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} diag[\mathbf{i}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\mathbf{f}_t$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_{t-1}} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t}\frac{\partial \mathbf{f}_t}{\partial \mathbf{h}_{t-1}} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t}\frac{\partial \mathbf{i}_t}{\partial \mathbf{h}_{t-1}} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} diag[\mathbf{o}_t \circ (1 - \mathbf{o}_t)]W_{oh} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} diag[\mathbf{f}_t \circ (1 - \mathbf{f}_t)]W_{fh} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} diag[\mathbf{i}_t \circ (1 - \mathbf{i}_t)]W_{ih}$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{x}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{x}_t} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t}\frac{\partial \mathbf{f}_t}{\partial \mathbf{x}_t} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t}\frac{\partial \mathbf{i}_t}{\partial \mathbf{x}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} diag[\mathbf{o}_t \circ (1 - \mathbf{o}_t)]W_{ox} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} diag[\mathbf{f}_t \circ (1 - \mathbf{f}_t)]W_{fx} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} diag[\mathbf{i}_t \circ (1 - \mathbf{i}_t)]W_{ix}$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_f} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} diag[\mathbf{f}_t \circ (1 - \mathbf{f}_t)] diag[\mathbf{z}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_i} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} diag[\mathbf{i}_t \circ (1 - \mathbf{i}_t)] diag[\mathbf{z}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_c} = \frac{\partial \mathbf{h}_t}{\partial \tilde{\mathbf{c}}_t} diag[1 - \tilde{\mathbf{c}}_t^2] diag[\mathbf{z}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_o} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} diag[\mathbf{o}_t \circ (1 - \mathbf{o}_t)] diag[\mathbf{z}_t]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_f} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{f}_t} diag[\mathbf{f}_t \circ (1 - \mathbf{f}_t)]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_i} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} diag[\mathbf{i}_t \circ (1 - \mathbf{i}_t)]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_c} = \frac{\partial \mathbf{h}_t}{\partial \tilde{\mathbf{c}}_t} diag[1 - \tilde{\mathbf{c}}_t^2]$
- $\frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_o} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} diag[\mathbf{o}_t \circ (1 - \mathbf{o}_t)]$

QUESTION 2

- To update the parameters Wh, Wx and b, we should **calculate the gradient every times**(in this assignment, the times is equal to sentence length), then **add all** of them. To calculate the gradient, we take the **Backpropagation Through Time**(BPTT), and from the questions 1, we can know just need to get the gradient of $\mathbf{h}_t$(called below δ_t).
- We can get the δ_t of last time through directly calculate $\frac{\partial Loss}{\partial \mathbf{h}_t}$ and

$$\delta_{t-1} = \frac{\partial Loss}{\partial \mathbf{h}_{t-1}} = \frac{\partial Loss}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \delta_t \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}.$$

Therefore, we can get the all gradients.

PART 2

QUESTION 1

Why should not just initialize all parameters to zero

- Generally, when initializing all parameters to zero, there is a **symmetry problem**, that is, if all pathways are initialized the **same exact way**, the learning process works the exact same way for all of the pathways, and so our **outputs will always be exactly the same** for each neuron which is no different to just have one neuron in the hidden layer. Because of this, the network will end up learning just one function while the goal of a neural network is to have different neurons compute different function.
- Moreover, sometimes nothing will train at all **depending on what activation function and error function used**. It is proved that for the Relu function, if the weights parameters are 0, the gradients will be 0, and therefore the weights will not change at all.
- Reference (https://www.quora.com/Why-dont-we-initialize-the-weights-of-a-neural-network-to-zero)

Way to initialize parameters

- One way to initialize parameters is called **Xavier Initialization**(random initialization). This method servers as a good starting point for initialization and mitigate the chances of exploding or vanishing gradients. It set the weights **neither too much bigger nor too much less**. So the gradients do not vanish or explode too quickly.
- In `PyTorch`, function `torch.nn.init.xavier_uniform_()` realized the Xavier Initialization, and we can just call this function to initialize the parameters.

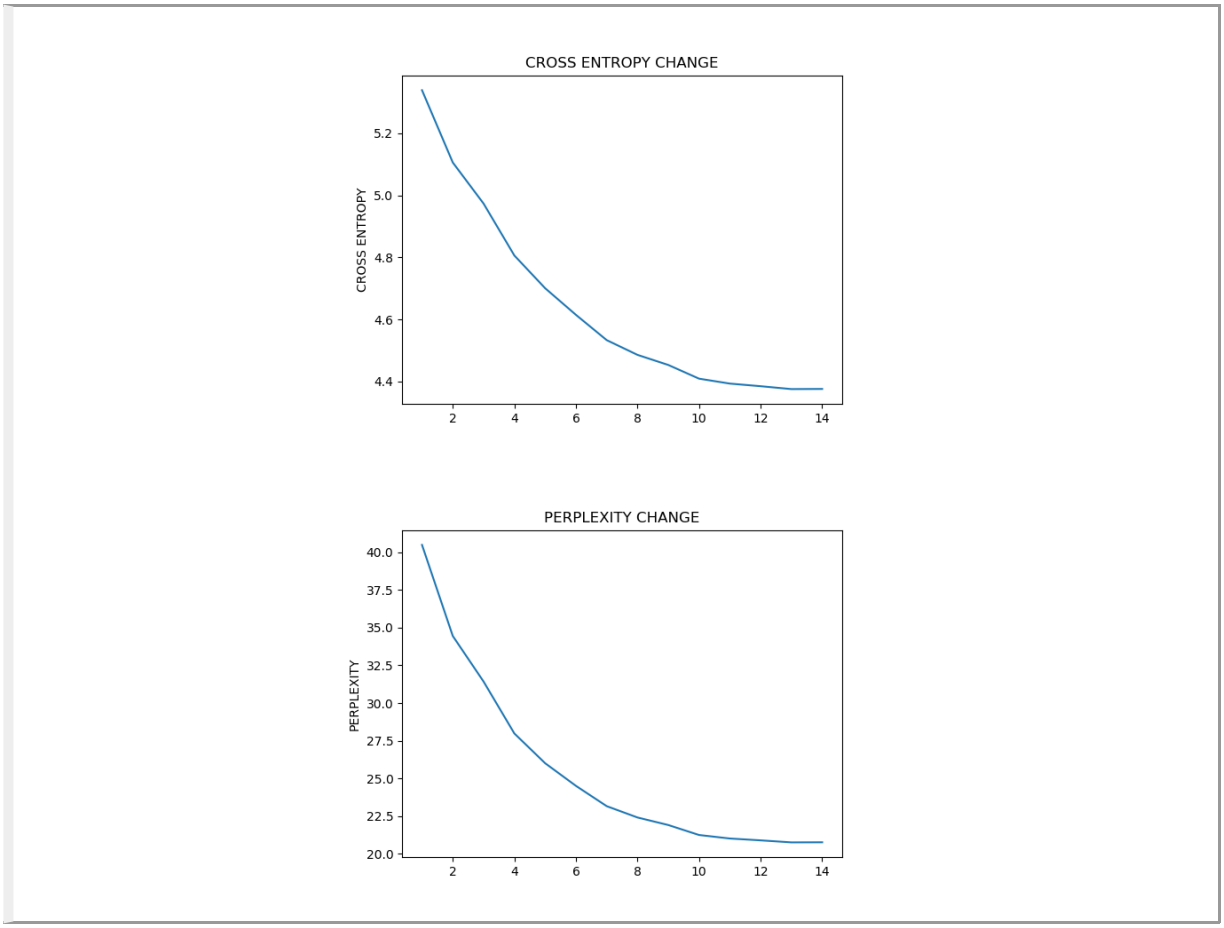
QUESTION 2

Process of generating poems

- Firstly, **preprocess the dataset** downloaded from 全唐诗 (https://github.com/chinese-poetry/chinese-poetry), create a vocabulary dictionary and devide them into training dataset and test dataset(rate about 9:1). When preprocessing the dataset, instead of randomly crop the sequence into batches of short sequence, I choose the sentence length smaller than 50 and the rest part **add tag**. The reason why is 50 is that I found the length of most poems in dataset will no more than 50.
- Secondly, **build a model**, including **embedding layer**, **lstm layer** and **linear layer**. Because this is my first time to use `PyTorch` and I am not much familiar with lstm, I use the lstm layer from `torch.nn.lstm` first. After training successfully, I write my own lstm layer(inherited the `nn.model` and override the `__init__` and `forward` function). The input of the model is a 2-dimension matrix, its shape is (batch size, sentence length), and each poems in batch **does not contain the last word**. On contrary, the poems in target batch does not contain the first word.
- Lastly, train the model by training dataset, and each epoch, I will evaluate it with test dataset, calculate and print the cross entropy and perplexity to judge whether the whole process is wrong. When the perplexity of test dataset begins to decrease, stop training. Then we can generate the poems.
- Finally, the **vocabulary size** is about **3500**(after deleting some low-frequency words), the **batch size** is **256**, the **sentence length** is **50**, the **hidden size** is **256**, the **input size** is **128**.

Result

- The calculation of perplexity, I use Another formula, that is
$$perplexity(W) = 2^{H(W)} = 2^{-\frac{1}{N}logP(w_1w_2...w_N)}$$
, in other words, perplexity is equal to **an exponential form of cross entropy**, and then, we can easily calculate it.
- The change in cross entropy and perplexity of test dataset throughout the process is as follows(in the Adam optimization algorithm).



- The generated poems start with 日、红、山、夜、湖、海、月 are as follows.(the screenshots is in the appendix)

> 日暮銅雀閣，山川上月華。
> 紅萼競燃春苑曙，春風吹雪寒山。
> 山中有餘，何處是鄉。何言嶧陽，不敢寧。
> 夜久星回凜，三秋月照春。
> 湖上春風吹，長安得知名。
> 海岳臨丹壑，仙閣涌南薰。
> 月皎傍風送，山風照安思。

QUESTION 3

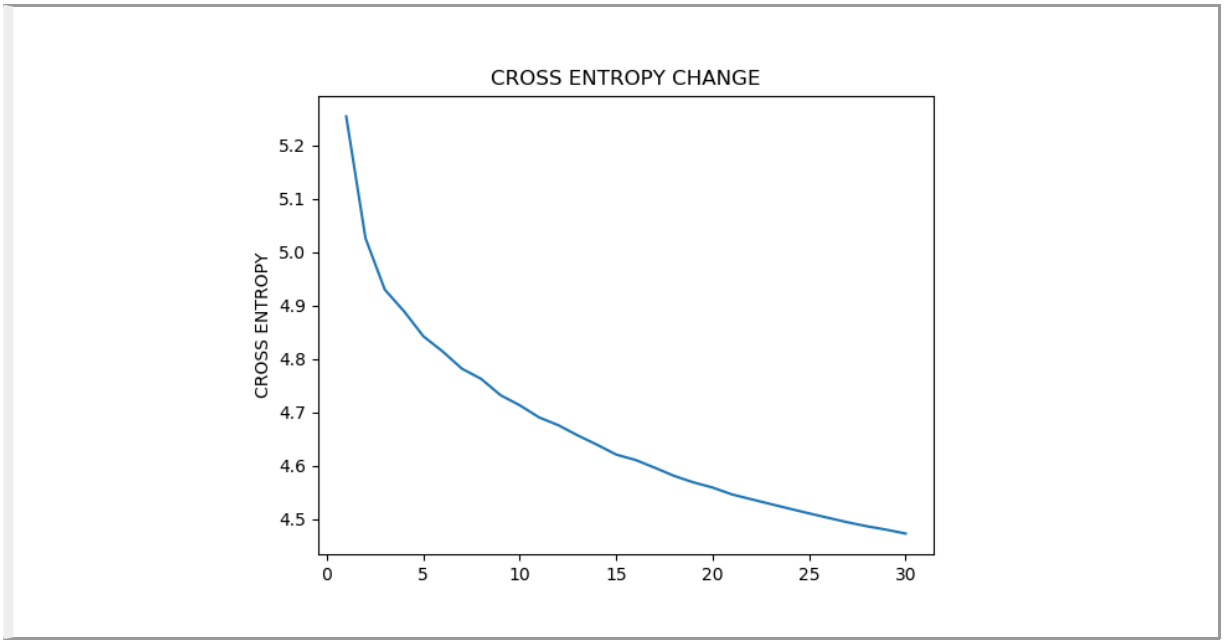
AdaGrad

- For brevity, $\mathbf{g}_t$ is the partial derivative of the loss to the parameter at time step $\mathbf{t}$. Then the SGD update the parameter at each time step $\mathbf{t}$ becomes:
$$\theta_{t+1} = \theta_t - \eta \cdot g_t.$$
And AdaGrad **modifies the general learning rate η at each time step t** for parameter **based on the past gradients** that have been computed for the parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

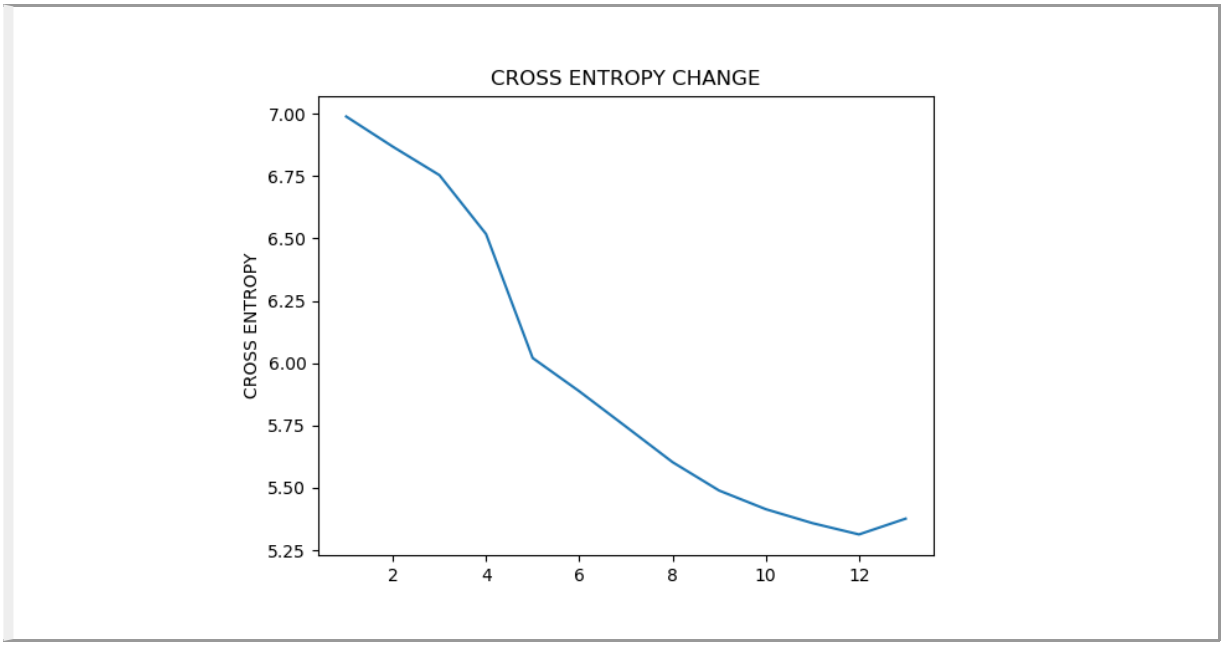
G_t is a diagonal matrix containing the sum of the squares of the past gradients.

- One of the AdaGrad's main benefits is that it **eliminates the need to manually tune the learning rate**. But since every added term in the denominator is positive, the accumulated sum keeps growing during training. This in turn causes the **learning rate to shrink and eventually become inifinitesimally small**.
- From the change in cross entropy of test dataset, we can see that the cross entropy decrease very slowly in the second half of the process. In other word, it almost learned nothing.



Adadelta

- Adadelta is an extension of AdaGrad. Instead of storing previous squared gradients, **the sum of gradients is recursively defined as a decaying average of all past squared gradients**.
- Although the training is stoped as the cross entropy of test dataset began to increase, the value is still large and the result is not so good.



Adam

- In addition to storing an exponentially decaying average of past squared gradients, Adam also **keeps an exponentially decaying average of past gradients**. Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. The result is the above one.
- Reference (http://ruder.io/optimizing-gradient-descent/index.html#adagrad)

APPENDIX