COMP4660/8420 Lab 5

Deep Learning

Part 1: Recurrent Neural Networks and LSTMs

1. What type of tasks do recurrent neural networks excel at?
   Any sequential time series data, such as language modelling and speech recognition etc

2. For an input size of n with h hidden nodes in one hidden layer and the output coming directly from the RNN/LSTM (meaning that there is no output layer for RNN and no additional fully connected layer after the RNN/LSTM), how many weights (including biases) would there be in the RNN and LSTM respectively?

   RNN:

[input to hidden]	n * h
[hidden to hidden]	h * h
[biases]	h
[total]	$nh + h^2 + h$

   LSTM:

   $$
   \begin{aligned}
   i_t &= \text{sigmoid}(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
   f_t &= \text{sigmoid}(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
   g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hc}h_{(t-1)} + b_{hg}) \\
   o_t &= \text{sigmoid}(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
   c_t &= f_t * c_{(t-1)} + i_t * g_t \\
   h_t &= o_t * \tanh(c_t)
   \end{aligned}
   $$

   Each gate and the memory cell receives input from the input layer and the output of the previous time step and has its own biases:

[W_{ii}, W_{if}, W_{ig}, W_{io} dimensions]	n * h
[W_{hi}, W_{hf}, W_{hg}, W_{ho} dimensions]	h * h
[b_i, b_f, bg, b_o biases dimensions]	h
[total]	$4(nh + h^2 + h)$

3. What two problems do traditional recurrent neural networks suffer from?
   - vanishing or exploding gradient during backpropagation,
   - long term dependencies

4. How is 'memory' stored within a LSTM?
   Within a persistent cell state with four gates:
   - a 'forget' gate which controls what should be removed;
   - an 'input gate' which determines what should be added;

- a 'write' gate (g) which determines what data should be updated.
- an 'output' gate which determines what should be output

Part 2: Recurrent Neural Networks and LSTMs in Python/PyTorch

2.1 Recurrent Neural Network

This task will help you understand and build a basic recurrent neural network with Python Numpy.

Download "task2_min_char_rnn.py" and 'textData.txt' from Wattle and put them under the same directory. This script demonstrates how to train a RNN on single characters using the Numpy package, and is a good tutorial for understanding what is going on behind the scenes, so we encourage you to **read and understand** it.

Hint: Please refer to *Appendix. Using the Chain Rule for backpropagation* if you have trouble understanding how gradients are calculated.

Please note that line 41 (`hs[t] = …`) is unimplemented intentionally. Try to **fill in this line.** If you get stuck your tutor can give you that line. This code is provided to help your understanding of what a basic recurrent neural network does, as you can't see the recurrency in Pytorch. It also provides examples of implementation of some of the basic steps of the forward pass and backward pass of a neural nework along with a Tanh (similar to sigmoid) and softmax activation functions, and cross-entropy/log loss. This is to aid your understanding only and you are not expected to be able to implement a neural network without use of a neural network language for this course.

2.3 LSTM

This task will help you understand and build a basic LSTM with PyTorch.

Download "task3_min_char_lstm.py" and 'textData.txt' from Wattle and put them under the same directory. Similar to task 2.2, this script demonstrates how to train a RNN on single characters but with a different package, PyTorch.

The two functions in class `_charLSTM` (`__init__(self)` and `forward(self, input, hprev, cprev, seq_len)`) are intentionally blank, so try to **implement these two functions** to define a one layer LSTM and its forward pass.

Hint: Please refer to http://pytorch.org/docs/master/nn.html#torch.nn.LSTM if you don't know how to start.

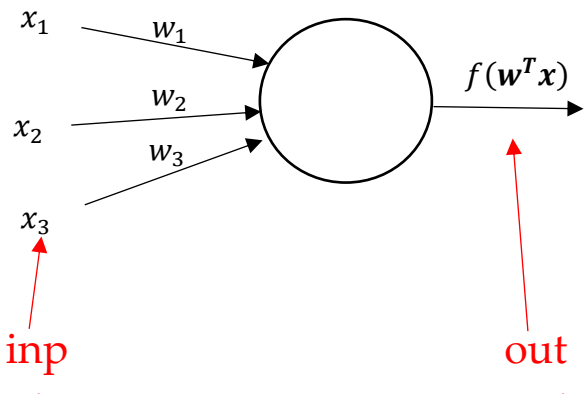
Also line 43 (`criterion = …`) is missing so **try to set a criterion** that equals to the appropriate loss function.

Line 98 (`hprev = …`) and line 99 (`cprev = …`) are not implemented so try to **assign correct values to `hprev` and `cprev`.**

Appendix. Using the Chain Rule for backpropagation

The Chain Rule for differentiation is: $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

This means that once you have calculated the derivative of the loss function with respect to a given output function in a neural network, you can calculate the derivative with respect to an input to that output function by simply multiplying the pre-calculated derivative by the partial derivative with respect to the desired input.



Raw output z = $\sum_i w_i x_i = \boldsymbol{w^T x}$
Final output = $f(\boldsymbol{w^T x})$

For example, if the activation function is $tanh$ in the above diagram and the loss function is the mean squared error (so for each output $L = \frac{1}{2}(t-y)^2$), $\frac{dL}{dy}$ can be calculated using the chain rule as follows:

$$L = \frac{1}{2}(t-y)^2$$

$$\frac{dL}{d(t-y)} = \frac{1}{2} * 2 * (t-y) = (t-y)$$

$$\because \frac{d(t-y)}{dy} = -1$$

$$\therefore \frac{dL}{dy} = \frac{dL}{d(t-y)}\frac{d(t-y)}{dy} = (t-y) * (-1) = y - t$$

Similarly, suppose $yraw$ is the output of the node before it passes through the activation function, and note that the derivative of $tanh(x)$ is $1 - tanh^2(x)$, $\frac{dL}{dw}$ can be calculated using the chain rule as follows:

$$\because y = tanh(yraw)$$

$$\therefore \frac{dy}{dyraw} = 1 - y^2$$

$$\therefore \frac{dL}{dyraw} = \frac{dL}{dy}\frac{dy}{dyraw} = (y-t)(1-y^2)$$

$$\because \frac{dyraw}{dw} = \boldsymbol{x^T} \text{ where } \mathbf{x} \text{ is the tensor (vector) of inputs}$$

$$\therefore \frac{dL}{dw} = \frac{dL}{dy}\frac{dy}{dyraw}\frac{dyraw}{dw} = \frac{dL}{dyraw}\frac{dyraw}{dw} = \frac{dL}{dyraw}\boldsymbol{x}^T$$