Learning Addition Arithmetic with Recurrent Neural Network

Background

In this homework, we try to use recurrent neural network (RNN) to learn how to perform addition arithmetic. At the start, the RNN knows absolutely nothing about addition. However, after seeing enough inputs and targets, it can figure out how to perform addition for itself!

Problem Setup

First we consider two 4-digital numbers, for example 1234 and 5678. We want a RNN can learn to predict the **sum** of them, like

```
a:  1   2   3   4
b:  5   6   7   8
RNN(a, b)
--> 6   9   1   2
```

For human, the standard algorithm for adding multi-digit numbers is to align the numbers vertically and add the columns, starting from the rightmost column. If a column exceeds ten, the extra digit is "*carried*" into the next column. It is a typical **sequence-to-sequence** problem and RNN is suitable for sequence mapping. So we can solve the addition arithmetic with RNN by working sequentially through each pair of digits and outputting the corresponding output digit like this

```
a: 1   2   3   4
b: 5   6   7   8
       |
      RNN
       |
       V
c:     9   1   2
```

RNN implementation with Theano

In `layers.py`, a simple RNN has been implemented to show how to use theano to perform recurrent operation. The core part is to use `theano.scan` to iteratively compute output and hidden states by `step` function, which tells RNN how to forward inputs to outputs one step. To understand more, go to link for details.

Dataset Description

Each training data is a pair of two randomly generated numbers, but in one-hot form for each digits. For example `1234` will be represented in a 4 x 10 size matrix

```
[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  # 1
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  # 2
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  # 3
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]  # 4
```

For a pair of 4-digital number, like `1234` and `5678`, the input is a 4 x 20 size matrix by concatenating two 4 x 10 size matrix horizontally (like aligning two numbers vertically when we do addition). Since we do math from right-most column to the left, the rows of input matrix are further **reversed** in order. To avoid the *carrying* digit at the left-most location, we restrict the sum of two numbers less than 10000. For numbers less than 1000, the left digits are padded with zeros to fixed 4-digital length.

Python Files Description

- `data_preparation.py`: prepare 50000 training data and 10000 testing data.
- `layers.py`: including `Relu`, `Softmax`, `Linear`, `RNN`, `LSTM`
- `loss.py`: cross entropy loss
- `network.py`: sequentially add layers and compiling `train`, `test` functions
- `optimizer.py`: different stochastic gradient descent algorithms to minimize loss
- `run_rnn`, `solve_rnn`: the main program and training protocol
- `utils.py`: some auxiliary functions

Report

We perform the following experiments in this homework:

1) Implement LSTM layer. Notice we only need to implement the basic form of LSTM described in the following equations:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1} + V_i c_{t-1} + b_i) \\
f_t &= \sigma(W_f x_t + U_f h_{t-1} + V_f c_{t-1} + b_f) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + V_o c_{t-1} + b_o) \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where i, f, o are input gate, forget gate and output gate, c is memory cell, $\odot$ is elementwise product operator and σ is sigmoid function.

2) Construct one RNN layer network and one LSTM layer network using `SGDOptimizer`. Compare the difference of two results (you can discuss the difference from the aspects of training time, loss convergence and accuracy in *figures* and *tables*)

3) Implement `RMSpropOptimizer` and compare the results with 2) (from training

time/convergence/testing accuracy). RMSprop is a very effective, but currently unpublished adaptive learning rate method (see Lecture 6 of Geoff Hinton's Coursera class). The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:

```python
# Assume the gradient dx and parameter vector x
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

Submission Guideline

- **report**: well formatted readable summary including your results, discussions and ideas. Source codes should not be included in report writing. Only some essential lines of codes are permitted for explaining complicated thoughts.
- **codes**: organized source code files with README for any extra installation or other procedures. Ensure one can successfully reproduce your results following your instructions. **DO NOT** include model weights/raw data/compiled objects/unrelated stuff over 100MB...

Deadline: Nov. 20

TA contact info: Yulong Wang (王宇龙), wang-yl15@mails.tsinghua.edu.cn