Automatic Reverse-Mode Differentiation for Long Short Term Memory (LSTM)

TAs: Tao Lin (tao.lin@cs.cmu.edu),
Rose (rosecatherinek@cs.cmu.edu)

Out Nov 7, 2017
Due Nov 14, 2017 via Autolab

Prerequisite: Please ensure you have first read Prof Cohen's notes on Automatic Reverse Mode Differentiation[1]. You can also refer to the sample code here[2].

Guidelines for Answers: Please answer to the point. Please state any additional assumptions you make while answering the questions. You need to submit a tar file containing source files and a **pdf** version of report separately to autolab. Please make sure you write the report legibly for grading.

Rules for Student Collaboration: The purpose of student collaboration in solving assignments is to facilitate learning, not to circumvent it. Studying the material in groups is strongly encouraged. It is allowed to seek help from other students in understanding the material needed to solve a homework problem, provided no written notes are taken or shared during group discussions. The actual solutions must be written and implemented by each student alone, and the student should be ready to reproduce their solution upon request. You may ask clarifying questions on Piazza. However, under no circumstances should you reveal any part of the answer publicly on Piazza or any other public website. Any incidents of plagiarism or collaboration without full disclosure will be handled severely.

Rules for External Help: Some of the homework assignments used in this class may have been used in prior versions of this class, or in classes at other institutions. Avoiding the use of heavily tested assignments detracts

[1] http://www.cs.cmu.edu/~wcohen/10-605/notes/autodiff.pdf
[2] http://www.cs.cmu.edu/~wcohen/10-605/code/sample-use-of-xman.py

from the main purpose of these assignments, which is to reinforce the material and stimulate thinking. Because some of these assignments may have been used before, solutions to them may be available online or from other people. It is explicitly forbidden to use any such sources or to consult people who have solved these problems before. You must solve the homework assignments completely on your own. We will mostly rely on your wisdom and honor to follow this rule. However, if a violation is detected, it will be dealt with harshly.

- Did you receive any help whatsoever from anyone in solving this assignment? Yes/No
- If you answered yes, give full details:_____
  (e.g. "Jane explained to me what is asked in Question 3.4")
- Did you give any help whatsoever to anyone in solving this assignment? Yes/No
- If you answered yes, give full details:_____
  (e.g. "I pointed Joe to section 2.3 to help him with Question 2")

1 Overview

In this assignment we will use the automatic differentiation system introduced in HW4 to implement a Long Short Term Memory (LSTM) network architecture for character level entity classification, using `python` and `numpy`.

Character level entity classification refers to determining the type of an entity given the characters which appear in its name as features. For example, given the name "`Antonio_Veciana`" you might guess that it is a `Person`, and given the name "`Anomis_esocampta`" you might guess that it is a `Species`. As in HW4, we will be classifying the following 5 `DBPedia` categories - `Person`, `Place`, `Organisation`, `Work`, `Species`.

2 Long Short Term Memory

The MLP that you implemented in HW4 is a powerful model – with enough hidden units it can approximate any function, but it is not the most appropriate model when the input is a sequence. For sequences, the input size of the MLP and consequently size of $W^{(1)}$, will increase linearly with the size of the length of the sequence and might get prohibitively large. Instead, we would like to have a model which can loop over the input sequence, and starting from an initial state iteratively updates its output based on the input at that time step. LSTMs are one example of such a model [3].

Lets say we have a sequence of inputs $x_1, x_2, \ldots, x_M \in \mathbb{R}^{d_{in}}$, an initial *cell state* $c_0 = \mathbf{0} \in \mathbb{R}^{d_{hid}}$ and an initial *output* $h_0 = \mathbf{0} \in \mathbb{R}^{d_{hid}}$. At time t the LSTM does the following updates:

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

[3] An introduction to LSTMs can be found at http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Here $W_* \in \mathbb{R}^{d_{in} \times d_{hid}}$, $U_* \in \mathbb{R}^{d_{hid} \times d_{hid}}$, $b_* \in \mathbb{R}^{d_{hid}}$ are parameters, $\odot$ is an element-wise product and σ is the sigmoid function (applied element-wise):

$$\sigma(x)_i = \frac{1}{1 + e^{-x_i}} \tag{1}$$

Note: The above equations are shown for single inputs x for clarity. In your implementation you must use minibatches X of N examples at a time, the same way as we did for the MLP. This amounts to replacing x_t^T with matrices X_t of size $N \times d_{in}$, and h_t^T, c_t^T with matrices H_t, C_t of size $N \times d_{hid}$.

Now we are ready to implement LSTM architecture. We will replace the first layer of MLP of HW4 with an LSTM layer. Let $\text{LSTM}(X_1, X_2, \ldots, X_M)$ be a function which loops over the sequence $X_1, \ldots, X_M$, performs the updates described above, and returns the final output H_M. Then, given inputs $X_1, \ldots, X_M$, and their associated targets T, the output P and loss are computed as follows:

$$\begin{aligned} H_M &= \text{LSTM}(X_1, X_2, \ldots, X_M) \\ O^{(2)} &= \text{relu}(H_M W^{(2)} + b^{(2)}) \\ P &= \text{softMax}(O^{(2)}) \\ \text{loss} &= \text{mean}(\text{crossEnt}(P, T)) \end{aligned}$$

2.1 Learning

As before, parameters of the above network can be trained using minibatch SGD. Once the loss function is defined we can take its derivative wrt to any parameter w_{ij} and update it as follows:

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k-1)} - \lambda \frac{d\text{loss}}{dw_{ij}} \tag{2}$$

λ is the learning rate. In this assignment, you are not required to modify the learning rate as the training proceeds.

3 Autodiff Implementation

In the starter code, the following files are given -

- `lstm.py` – you need to implement the lstm here

- `functions.py` – you need to add additional function definitions and their gradients here. Unit tests for new function definitions and their gradients are included to help you debug your implementation faster. You can run `python functions.py` to check your implementation.

Reuse the following files from the starter code of HW4:

- `xman.py`
- `utils.py`
- `autograd.py`

The steps needed to implement the LSTM model are similar to how you implemented MLP. Follow the instructions in Sections 3.1, 3.2 and 3.3 of HW4 to write your model.

4 Data

For this assignment, we will use the same dataset as HW4 (`tiny` for debugging, and `smaller` for reporting results). And the task is also the same – you need to predict the category label of a DBPedia entity based on its title.

As in HW4, we will encode entities for input to the networks by converting characters to a one-hot representation. A string of characters will be encoded to a matrix whose each row is a V-dimensional vector, where V is the total number of characters in the dataset. We will fix the maximum length of an entity to M, longer entities will be truncated to this length, and shorter ones will be padded with white-space. We will use the `DataPreprocessor` of `utils.py` to iterate over the data as before:

```
for (idxs,e,l) in mb_train:
    # idxs - ids of examples in minibatch
    # e - entities in one-hot format
    # l - corresponding output labels also in one-hot format
```

`idxs` has shape N, `e` has shape $N \times M \times V$ and `l` has shape $N \times C$ where N is the batch size. Make sure that this makes sense to you. For input to the LSTM, we will create a sequence of inputs $X_1, \ldots, X_M$ from `e`. Each of these would be a $N \times V$ matrix holding the batch inputs from time-step 1 through M. Since we are using the *final* state of the LSTM for classification,

we must feed inputs to the network in reverse order so that the useful characters appear at end.

As before, you should evaluate the loss function on the validation dataset (*.valid files) after every epoch and store the parameters of the best model in a separate dictionary. Then after training is completed, use these best parameters to make predictions on the test set. Remember that you should not do backpropagation on the validation dataset. For reporting the results, use the test (*.test) files.

As before, you can get the data from

`https://drive.google.com/open?id=0B58rr945j04BcGctanY1UHVFZVE`

For the evaluation on Autolab we will run your code on a separate train, validation and testing dataset `autolab` with a specific set of params as command line arguments.

5 Deliverables

You need to write the code for building, training and evaluating a LSTM in `lstm.py`.

You need to write your function definitions and their derivatives in `functions.py`. Make sure that you call the final predicted labels as `outputs` and the loss function as `loss` in the `XMan` object. You should also remember to initialize the default parameters and inputs. Otherwise you'd get zero score on our gradient checking code.
You should run your code using the following defaults:

```
python lstm.py --max_len 10 --num_hid 50
               --batch_size 64 --dataset small
               --epochs 25 --init_lr 0.5
               --output_file output
               --train_loss_file train_loss
```

In autolab we will call the main function in `lstm.py` and pass a dictionary of hyperparameter values. We will check your code for gradient correctness, training loss trend, mean Loss on the test output, implementation of function definitions and derivates, and speed compared to the benchmark code written by us.

The training loss you obtain after each iteration on the `batch_size`d data (not epoch) should be stored in the file specified by `--train_loss_file` in numpy format using `np.save()`. Your final output probabilities should be stored in the file specified by –output-file in numpy format using np.save() in the same row order as the input test file. Tar the following files for submission.

- lstm.py (This file would contain the LSTM class and the main class with the default params)
- functions.py (This file would contain the function definitions and their gradients)
- Any other helper files

Tar the files directly using the command below. Do NOT put the above files in a folder and then tar the folder. You do not need to upload the saved temporary files.

```
tar -cvf hw5.tar lstm.py functions.py
```

Also, please do not forget to submit your report `hw5.pdf` via the HW5: Report link in Autolab.

6 Questions

1. Plot the average training time per epoch for LSTM for batch size $N = \{16, 32, 64\}$, and default values of remaining hyperparameters. What trends do you observe? Is it similar to the trend you observed for MLP in HW4? Why or why not?

2. Plot the average training time per epoch for LSTM for maximum input length $M = \{10, 15, 20\}$, and default values for remaining hyperparameters. What trends do you observe? Is it similar to the trend you observed for MLP in HW4? Why or why not?

3. Plot the total number of forward and backward steps in one update of the LSTM for maximum input length $M = \{10, 15, 20\}$, and default values for remaining parameters. Now repeat the plot for batch size $N = \{16, 32, 64\}$. One forward step is defined as computing the output

of a primitive operation, and one backward step is defined as computing the derivative of a primitive function wrt *one* of its inputs. Briefly explain the trends that you observe.

Hint: You can count the forward and backward steps by placing counters in `Autograd.eval` and `Autograd.bprop` respectively.

7 Grading Scheme

- Code Correctness (gradient check)[4] - 20 points
- Code Correctness (training loss check) - 10 points
- Code Correctness (unit tests on function implementation) - 10 points
- Code Accuracy (loss on held-out test set) - 15 points
- Code Speed - 15 points
- Report Questions - 30 points

[4] **Reminder** - You must name the loss register as "`loss`" for our gradient checking to work