A Tutorial On Backward Propagation Through Time (BPTT) In The Gated Recurrent Unit (GRU) RNN

Minchen Li
Department of Computer Science
The University of British Columbia
minchenl@cs.ubc.ca

Abstract

In this tutorial, we provide a thorough explanation on how BPTT in GRU[1] is conducted. A MATLAB program which implements the entire BPTT for GRU and the psudo-codes describing the algorithms explicitly will be presented. We provide two algorithms for BPTT, a direct but quadratic time algorithm for easy understanding, and an optimized linear time algorithm. This tutorial starts with a specification of the problem followed by a mathematical derivation before the computational solutions.

1 Specification

We want to use a dataset containing n_s sentences each with n_w words to train a GRU language model, and our vocabulary size is n_v. Namely, we have input $x \in R^{n_v \times n_w \times n_s}$ and label $y \in R^{n_v \times n_w \times n_s}$ both representing n_s sentences.

For simplicity, lets look at one sentence at a time. In one sentence, the one-hot vector $x_t \in R^{n_v \times 1}$ represents the t^{th} word. For time step t, the GRU unit computes the output $\hat{y}_t$ using the input x_t and the previous internal state s_{t-1} as follows:

$$
\begin{aligned}
z_t &= \sigma(U_z x_t + W_z s_{t-1} + b_z) \\
r_t &= \sigma(U_r x_t + W_r s_{t-1} + b_r) \\
h_t &= \tanh(U_h x_t + W_h(s_{t-1} \odot r_t) + b_h) \\
s_t &= (1 - z_t) \odot h_t + z_t \odot s_{t-1} \\
\hat{y}_t &= softmax(V s_t + b_V)
\end{aligned}
\tag{1}
$$

Here $\odot$ is the vector element-wise multiplication, $\sigma()$ is the element-wise $sigmoid$ function, and $tanh()$ is the element-wise $hyperbolictangent$ function. The dimensions of the parameters are as follows:

$$
\begin{aligned}
&U_z, U_r, U_h \in R^{n_i \times n_v} \\
&W_z, W_r, W_h \in R^{n_i \times n_i} \\
&b_z, b_r, b_h \in R^{n_i \times 1} \\
&V \in R^{n_v \times n_i}, b_V \in R^{n_v \times 1}
\end{aligned}
$$

where n_i is the internal memory size set by the user.

[1]GRU is an improved version of traditional RNN (Recurrent Neural Network, see WildML.com for an introduction. This link also provides an introduction to GRU and some general discussion on BPTT and beyond.)

Then for step t, we can calculate the cross entropy loss L_t as:

$$L_t = sumOfAllElements\Big(- y_t \odot log(\hat{y}_t)\Big) \tag{2}$$

Here log is also an element-wise function.

To train the GRU, we want to know the values of all parameters that minimize the total loss $L = \sum_{t=1}^{n_w} L_t$:

$$\underset{\Theta}{\operatorname{argmin}} L$$

where $\Theta = \{U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V\}$. This is a non-convex problem with huge input data. So people usually use Stochastic Gradient Descent[2] method to solve this problem, which means we need to calculate $\partial L/\partial U_z$, $\partial L/\partial U_r$, $\partial L/\partial U_h$, $\partial L/\partial W_z$, $\partial L/\partial W_r$, $\partial L/\partial W_h$, $\partial L/\partial b_z$, $\partial L/\partial b_r$, $\partial L/\partial b_h$, $\partial L/\partial V$, $\partial L/\partial b_V$ given a batch of sentences. (Note that in each step, these parameters stays the same.) In this tutorial we consider using only one sentence at a time to make it concise.

2 Derivation

The best way to calculate gradients using the Chain Rule from output to input is to first draw the expression graph of the entire model in order to figure out the relations between the output, intermediate results, and the input[3]. Here we draw part of the expression graph of GRU in Fig.1.

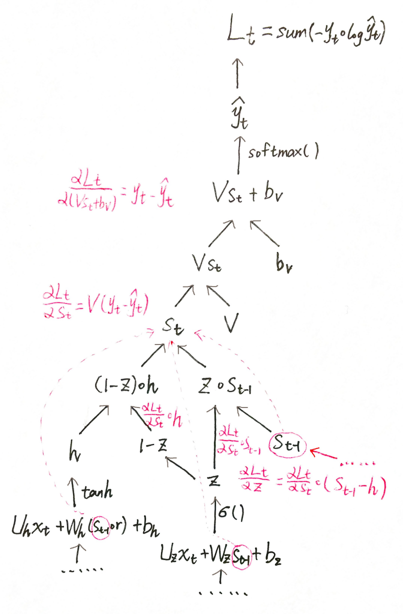

Figure 1: The upper part of expression graph describing the operations of GRU. Note that the sub-graph which s_{t-1} depends on is just like the sub-graph of s_t. This is what the red dashed lines mean.

With this expression graph, the Chain Rule works if you go backwards along the edges (top-down). If a node X has multiple outgoing edges connecting the target node T, you need to sum over the partial derivatives of each of those outgoing edges to derive the gradient $\partial T/\partial X$. We will illustrate the rules in the following paragraphs.

Let's take $\partial L/\partial U_z$ as the example here. Others are just similar. Since $L = \sum_{t=1}^{n_w} L_t$ and the parameters stay the same in each step, we also have $\partial L/\partial U_z = \sum_{t=1}^{n_w}(\partial L_t/\partial U_z)$, so let's calculate each $\partial L_t/\partial U_z$ independently and sum them up.

[2] See the Wikipedia to get some knowledge about Stochastic Gradient Descent.

[3] See colah's blog and Stanford CS231n Course Note for some general introductions.

With the Chain Rule, we have:

$$\frac{\partial L_t}{\partial U_z} = \frac{\partial L_t}{\partial s_t}\frac{\partial s_t}{\partial U_z} \tag{3}$$

The first part is just trivial if you know how to differentiate the cross entropy loss function embedded with the $softmax$ function:

$$\frac{\partial L_t}{\partial s_t} = V(\hat{y}_t - y_t)$$

For $\partial z/\partial U_z$, similarly, some people might just derive: (if they know how to differentiate $sigmoid$ function)

$$\overline{\frac{\partial s_t}{\partial U_z}} = \Big((s_{t-1} - h_t) \odot z_t \odot (1 - z_t)\Big)x_t^T \tag{4}$$

Here there are two expressions $1 - z$ and $z \odot s_{t-1}$ influencing $\partial s_t/\partial z$ as shown in our expression graph. The solution is to derive partial derivatives through each edge and then add them up, which is exactly how we deal with $\partial s_t/\partial s_{t-1}$ as you will see in the following paragraphs. However, Eq.4 only calculates one part of the gradient, so we put a bar on top of it, while you may find this very useful in our following calculations.

Note that s_{t-1} also depends on U_z, so we can not treat it as a constant here. Moreover, this s_{t-1} will also introduce the influence of s_i, where $i = 1, ..., t-2$. So for clearness, we should expand Eq.3 as:

$$\begin{aligned}\frac{\partial L_t}{\partial U_z} &= \frac{\partial L_t}{\partial s_t}\frac{\partial s_t}{\partial U_z} \\ &= \frac{\partial L_t}{\partial s_t}\sum_{i=1}^{t}\Big(\frac{\partial s_t}{\partial s_i}\overline{\frac{\partial s_i}{\partial U_z}}\Big) \\ &= \frac{\partial L_t}{\partial s_t}\sum_{i=1}^{t}\Big(\Big(\prod_{j=i}^{t-1}\frac{\partial s_{j+1}}{\partial s_j}\Big)\overline{\frac{\partial s_i}{\partial U_z}}\Big)\end{aligned} \tag{5}$$

where $\overline{\partial s_i}/\partial U_z$ is the gradient of s_i with respect to U_z while taking s_{i-1} as a constant, of which a similar example has been shown in Eq.4 for step t.

The derivation of $\partial s_t/\partial s_{t-1}$ is similar to the derivation of $\partial s_t/\partial z$ as has been discussed above. Since there are four outgoing edges from s_{t-1} to s_t directly and indirectly through z_t, r_t, and h_t in the expression graph, we need to sum all the four partial derivatives together:

$$\begin{aligned}\frac{\partial s_t}{\partial s_{t-1}} &= \frac{\partial s_t}{\partial h_t}\frac{\partial h_t}{\partial s_{t-1}} + \frac{\partial s_t}{\partial z_t}\frac{\partial z_t}{\partial s_{t-1}} + \overline{\frac{\partial s_t}{\partial s_{t-1}}} \\ &= \frac{\partial s_t}{\partial h_t}\Big(\frac{\partial h_t}{\partial r_t}\frac{\partial r_t}{\partial s_{t-1}} + \overline{\frac{\partial h_t}{\partial s_{t-1}}}\Big) + \frac{\partial s_t}{\partial z_t}\frac{\partial z_t}{\partial s_{t-1}} + \overline{\frac{\partial s_t}{\partial s_{t-1}}}\end{aligned} \tag{6}$$

where $\overline{\partial s_t}/\partial s_{t-1}$ is the gradient of s_t with respect to s_{t-1} while taking h_t and z_t as constants. Similarly, $\overline{\partial h_t}/\partial s_{t-1}$ is the gradient of h_t with respect to s_{t-1} while taking r_t as a constant.

Plugging the intermediate results in the above formula, we get:

$$\begin{aligned}\frac{\partial s_t}{\partial s_{t-1}} =&(1 - z_t)\Big(W_r^T((W_h^T(1 - h \odot h)) \odot s_{t-1} \odot r \odot (1 - r)) + ((W_h^T(1 - h \odot h)) \odot r_t\Big) + \\ &W_z^T\Big((s_{t-1} - h_t) \odot z_t \odot (1 - z_t)\Big) + z\end{aligned}$$

Till now, we have covered all the components needed to calculate $\partial L_t/\partial U_z$. The gradient of L_t with respect to other parameters are just similar. In the next chapter, we will provide a more machinery view of the calculation - the psudo-code describing the algorithm to calculate the gradients. In the last chapter of this tutorial, we will provide the pure machine representation - a MATLAB program which implements the calculation and verification of BPTT. If you just want to understand the idea behind BPTT and decide to use fully supported auto-differentiation packages (like Theano[4]) to build your own GRU, you can stop here. If you need to implement the exact chain rule like us or just curious about what will happen next, get ready to proceed!

[4] Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.

3 Algorithm

Here we also only take $\partial L/\partial U_z$ as the example. We will provide the calculation of all the gradients in the next chapter.

We present two algorithms, one direct algorithm as derived previously calculating $\partial L_t/\partial U_z$ and sum them up while taking $O(n_w^2)$ time, and the other $O(n_w)$ time algorithm which we will see later.

Algorithm 1 A direct but $O(n_w^2)$ time algorithm to calculate $\partial L/\partial U_z$ (and beyond)

Input: The training data $X, Y \in R^{n_v \times n_w}$ composed of the one-hot column vectors $x_t, y_t \in R^{n_v \times 1}$, $t = 1, 2, ..., n_w$ representing the words in the sentence.
Input: A vector $s_0 \in R^{n_i \times 1}$ representing the initial internal state of the model (usually set to 0).
Input: The parameters $\Theta = \{U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V\}$ of the model.
Output: The total loss gradient $\partial L/\partial U_z$.

1: %forward propagate to calculate the internal states $S \in R^{n_i \times n_w}$, the predictions $\hat{Y} \in R^{n_v \times n_w}$, the losses $L_{mtr} \in R^{n_w \times 1}$, and the intermediate results $Z, R, C \in R^{n_i \times n_w}$ of each step:
2: $[S, \hat{Y}, L_{mtr}, Z, R, C] = forward(X, Y, \Theta, s_0)$ % forward() can be implemented easily according to Eq.1 and Eq.2
3: $dU_z = zeros(n_i, n_v)$ % initialize a variable dU_z
4: $\partial L_{mtr}/\partial S = V^T(\hat{Y} - Y)$ % calculate $\partial L_t/\partial s_t$ for $t = 1, 2, ..., n_w$ with one matrix operation
5: **for** $t \leftarrow 1$ `to` n_w % calculate each $\partial L_t/\partial U_z$ and accumulate
6:     **for** $j \leftarrow t$ `to` 1 % calculate each $(\partial L_t/\partial s_j)(\overline{\partial s_j}/\partial U_z)$ and accumulate
7:         $\partial L_t/\partial z_j = \partial L_t/\partial s_j \odot (s_{j-1} - h_j)$ % $\partial s_j/\partial z_j$ is $(s_{j-1} - h_j)$, $\partial L_t/\partial s_j$ is calculated in the last inner loop iteration or in Line 4
8:         $\partial L_t/\partial(U_z x_j + W_z s_{j-1} + b_z) = \partial L_t/\partial z_j \odot z_j \odot (1 - z_j)$ % $\partial\sigma(x)/\partial x = \sigma(x) \odot (1 - \sigma(x))$
9:         $dU_z += \Big(\partial L_t/\partial(U_z x_j + W_z s_{j-1} + b_z)\Big) x_j^T$ % accumulate
10:         calculate $\partial L_t/\partial s_{j-1}$ using $\partial L_t/\partial s_j$ and Eq.6 % for the next inner loop iteration
11:     **end**
12: **end**
13: return dU_z % $\partial L/\partial U_z$

The above direct algorithm actually follows Eq.5 to calculate $\partial L_t/\partial U_z$ and then add them up to form $\partial L/\partial U_z$:

$$
\begin{aligned}
\frac{\partial L}{\partial U_z} &= \sum_{t=1}^{n_w} \frac{\partial L_t}{\partial U_z} \\
&= \sum_{t=1}^{n_w} \Big(\frac{\partial L_t}{\partial s_t} \sum_{i=1}^{t} \Big(\frac{\partial s_t}{\partial s_i} \frac{\overline{\partial s_i}}{\partial U_z} \Big) \Big) \\
&= \sum_{t=1}^{n_w} \Big(\frac{\partial L_t}{\partial s_t} \sum_{i=1}^{t} \Big(\Big(\prod_{j=i}^{t-1} \frac{\partial s_{j+1}}{\partial s_j} \Big) \frac{\overline{\partial s_i}}{\partial U_z} \Big) \Big)
\end{aligned}
$$

If we just expand $\partial L_t/\partial U_z$ to the second line of the above equation and do some reordering, we can get:

$$
\begin{aligned}
\frac{\partial L}{\partial U_z} &= \sum_{t=1}^{n_w} \Big(\frac{\partial L_t}{\partial s_t} \sum_{i=1}^{t} \Big(\frac{\partial s_t}{\partial s_i} \frac{\overline{\partial s_i}}{\partial U_z} \Big) \Big) \\
&= \sum_{t=1}^{n_w} \Big(\sum_{i=1}^{t} \Big(\frac{\partial L_t}{\partial s_t} \frac{\partial s_t}{\partial s_i} \frac{\overline{\partial s_i}}{\partial U_z} \Big) \Big) \\
&= \sum_{t=1}^{n_w} \Big(\sum_{i=1}^{t} \Big(\frac{\partial L_t}{\partial s_i} \frac{\overline{\partial s_i}}{\partial U_z} \Big) \Big)
\end{aligned}
$$

Right now the inner summation keeps the subscript of ∂L_t and iterate over ∂s_i. If we further expand the inner summation and then sort them to iterate over ∂L_i, we get:

$$\frac{\partial L}{\partial U_z} = \sum_{t=1}^{n_w}\Big(\Big(\sum_{i=t}^{n_w}\frac{\partial L_i}{\partial s_t}\Big)\overline{\frac{\partial s_t}{\partial U_z}}\Big) \tag{7}$$

For the inner summation of Eq.7, we have:

$$\begin{aligned}\sum_{i=t}^{n_w}\Big(\frac{\partial L_i}{\partial s_t}\Big) &= \Big(\sum_{i=t+1}^{n_w}\Big(\frac{\partial L_i}{\partial s_{t+1}}\frac{\partial s_{t+1}}{\partial s_t}\Big)\Big) + \frac{\partial L_t}{\partial s_t} \\ &= \Big(\sum_{i=t+1}^{n_w}\frac{\partial L_i}{\partial s_{t+1}}\Big)\frac{\partial s_{t+1}}{\partial s_t} + \frac{\partial L_t}{\partial s_t}\end{aligned} \tag{8}$$

This just gives us an updating formula to calculate this inner summation for each step t incrementally rather than executing another for loop, thus making it possible for us to implement an $O(n_w)$ time algorithm!

Algorithm 2 An optimized $O(n_w)$ time algorithm to calculate $\partial L/\partial U_z$ (and beyond)

Input: The training data $X, Y \in R^{n_v \times n_w}$ composed of the one-hot column vectors $x_t, y_t \in R^{n_v \times 1}$, $t = 1, 2, ..., n_w$ representing the words in the sentence.
Input: A vector $s_0 \in R^{n_i \times 1}$ representing the initial internal state of the model (usually set to 0).
Input: The parameters $\Theta = \{U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V\}$ of the model.
Output: The total loss gradient $\partial L/\partial U_z$.

1: %forward propagate to calculate the internal states $S \in R^{n_i \times n_w}$, the predictions $\hat{Y} \in R^{n_v \times n_w}$, the losses $L_{mtr} \in R^{n_w \times 1}$, and the intermediate results $Z, R, C \in R^{n_i \times n_w}$ of each step:
2: $[S, \hat{Y}, L_{mtr}, Z, R, C] = forward(X, Y, \Theta, s_0)$ % forward() can be implemented easily according to Eq.1 and Eq.2
3: $dU_z = zeros(n_i, n_v)$ % initialize a variable dU_z
4: $\partial L_{mtr}/\partial S = V^T(\hat{Y} - Y)$ % calculate $\partial L_t/\partial s_t$ for $t = 1, 2, ..., n_w$ with one matrix operation
5: **for** $t \leftarrow n_w$ `to` 1 % calculate each $\Big(\sum_{i=t}^{n_w}\Big(\frac{\partial L_i}{\partial s_t}\Big)\Big)\overline{\frac{\partial s_t}{\partial U_z}}$ and accumulate
6: $\quad\sum_{i=t}^{n_w}(\partial L_i/\partial z_t) = \Big(\sum_{i=t}^{n_w}(\partial L_i/\partial s_t)\Big) \odot (s_{t-1} - h_t)$ % $\partial s_t/\partial z_t$ is $(s_{t-1} - h_t)$, $\sum_{i=t}^{n_w}(\partial L_i/\partial s_t)$ is calculated in the last iteration or in Line 4. (when $t = n_w$, $\sum_{i=t}^{n_w}(\partial L_i/\partial s_t) = \partial L_t/\partial s_t$)
7: $\quad\sum_{i=t}^{n_w}(\partial L_i/\partial(U_z x_t + W_z s_{t-1} + b_z)) = \Big(\sum_{i=t}^{n_w}(\partial L_i/\partial z_t)\Big) \odot z_t \odot (1 - z_t)$ % $\partial\sigma(x)/\partial x = \sigma(x) \odot (1 - \sigma(x))$
8: $\quad dU_z += \Big(\sum_{i=t}^{n_w}(\partial L_i/\partial(U_z x_t + W_z s_{j-t} + b_z))\Big)x_t^T$ % accumulate
9: $\quad$calculate $\sum_{i=t-1}^{n_w}(\partial L_i/\partial s_{t-1})$ using Eq.6 and Eq.8 % for the next iteration
10: **end**
11: return dU_z % $\partial L/\partial U_z$

4 Implementation

Here we provide the MATLAB program which calculates the gradients with respect to all the parameters of GRU using our two proposed algorithms. It also checks the gradients with the numerical results. We will divide our code into two parts, the first part presented below contains the core functions implementing the BPTT of GRU we just derived, the second part is composed of some functions that are less important to the topic of this tutorial.

Core Functions

```
% This program tests the BPTT process we manually developed for GRU.
% We calculate the gradients of GRU parameters with chain rule, and then
% compare them to the numerical gradients to check whether our chain rule
% derivation is correct.

% Here, we provided 2 versions of BPTT, backward_direct() and backward().
% The former one is the direct idea to calculate gradient within each step
% and add them up (O(sentence_size^2) time). The latter one is optimized to
% calculate the contribution of each step to the overall gradient, which is
% only O(sentence_size) time.

% This is very helpful for people who wants to implement GRU in Caffe since
% Caffe didn't support auto-differentiation. This is also very helpful for
% the people who wants to know the details about Backpropagation Through
% Time algorithm in the Reccurent Neural Networks (such as GRU and LSTM)
% and also get a sense on how auto-differentiation is possible.

% NOTE: We didn't involve SGD training here. With SGD training, this
% program would become a complete implementation of GRU which can be
% trained with sequence data. However, since this is only a CPU serial
% Matlab version of GRU, applying it on large datasets will be dramatically
% slow.

% by Minchen Li, at The University of British Columbia. 2016-04-21

function testBPTT_GRU
    % set GRU and data scale
    vocabulary_size = 64;
    iMem_size = 4;
    sentence_size = 20; % number of words in a sentence
                        %(including start and end symbol)
                        % since we will only use one sentence for training,
                        % this is also the total steps during training.

    [x y] = getTrainingData(vocabulary_size, sentence_size);

    % initialize parameters:
    % multiplier for input x_t of intermediate variables
    U_z = rand(iMem_size, vocabulary_size);
    U_r = rand(iMem_size, vocabulary_size);
    U_c = rand(iMem_size, vocabulary_size);
    % multiplier for pervious s of intermediate variables
    W_z = rand(iMem_size, iMem_size);
    W_r = rand(iMem_size, iMem_size);
    W_c = rand(iMem_size, iMem_size);
    % bias terms of intermediate variables
    b_z = rand(iMem_size, 1);
```

```
    b_r = rand(iMem_size, 1);
    b_c = rand(iMem_size, 1);
    % decoder for generating output
    V = rand(vocabulary_size, iMem_size);
    b_V = rand(vocabulary_size, 1); % bias of decoder
    % previous s of step 1
    s_0 = rand(iMem_size, 1);

    % calculate and check gradient
    tic
    [dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0] = ...
        backward_direct(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
    toc
    tic
    checkGradient_GRU(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0, ...
        dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0);
    toc

    tic
    [dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0] = ...
        backward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
    toc
    tic
    checkGradient_GRU(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0, ...
        dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0);
    toc
end

% Forward propagate calculate s, y_hat, loss and intermediate variables for each step
function [s, y_hat, L, z, r, c] = forward(x, y, ...
    U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0)
    % count sizes
    [vocabulary_size, sentence_size] = size(x);
    iMem_size = size(V, 2);

    % initialize results
    s = zeros(iMem_size, sentence_size);
    y_hat = zeros(vocabulary_size, sentence_size);
    L = zeros(sentence_size, 1);
    z = zeros(iMem_size, sentence_size);
    r = zeros(iMem_size, sentence_size);
    c = zeros(iMem_size, sentence_size);

    % calculate result for step 1 since s_0 is not in s
    z(:,1) = sigmoid(U_z*x(:,1) + W_z*s_0 + b_z);
    r(:,1) = sigmoid(U_r*x(:,1) + W_r*s_0 + b_r);
    c(:,1) = tanh(U_c*x(:,1) + W_c*(s_0.*r(:,1)) + b_c);
    s(:,1) = (1-z(:,1)).*c(:,1) + z(:,1).*s_0;
    y_hat(:,1) = softmax(V*s(:,1) + b_V);
    L(1) = sum(-y(:,1).*log(y_hat(:,1)));
    % calculate results for step 2 - sentence_size similarly
    for wordI = 2:sentence_size
        z(:,wordI) = sigmoid(U_z*x(:,wordI) + W_z*s(:,wordI-1) + b_z);
        r(:,wordI) = sigmoid(U_r*x(:,wordI) + W_r*s(:,wordI-1) + b_r);
        c(:,wordI) = tanh(U_c*x(:,wordI) + W_c*(s(:,wordI-1).*r(:,wordI)) + b_c);
```

```
        s(:,wordI) = (1-z(:,wordI)).*c(:,wordI) + z(:,wordI).*s(:,wordI-1);
        y_hat(:,wordI) = softmax(V*s(:,wordI) + b_V);
        L(wordI) = sum(-y(:,wordI).*log(y_hat(:,wordI)));
    end
end

% Backward propagate to calculate gradient using chain rule
% (O(sentence_size) time)
function [dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0] = ...
    backward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0)
    % forward propagate to get the intermediate and output results
    [s, y_hat, L, z, r, c] = forward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, ...
        b_z, b_r, b_c, V, b_V, s_0);
    % count sentence size
    [~, sentence_size] = size(x);

    % calculate gradient using chain rule
    delta_y = y_hat - y;
    db_V = sum(delta_y, 2);

    dV = zeros(size(V));
    for wordI = 1:sentence_size
        dV = dV + delta_y(:,wordI)*s(:,wordI)';
    end

    ds_0 = zeros(size(s_0));
    dU_c = zeros(size(U_c));
    dU_r = zeros(size(U_r));
    dU_z = zeros(size(U_z));
    dW_c = zeros(size(W_c));
    dW_r = zeros(size(W_r));
    dW_z = zeros(size(W_z));
    db_z = zeros(size(b_z));
    db_r = zeros(size(b_r));
    db_c = zeros(size(b_c));
    ds_single = V'*delta_y;
    % calculate the derivative contribution of each step and add them up
    ds_cur = zeros(size(ds_single,1), 1);
    for wordJ = sentence_size:-1:2
        ds_cur = ds_cur + ds_single(:,wordJ);
        ds_cur_bk = ds_cur;

        dtanhInput = (ds_cur.*(1-z(:,wordJ)).*(1-c(:,wordJ).*c(:,wordJ)));
        db_c = db_c + dtanhInput;
        dU_c = dU_c + dtanhInput*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_c = dW_c + dtanhInput*(s(:,wordJ-1).*r(:,wordJ))';
        dsr = W_c'*dtanhInput;
        ds_cur = dsr.*r(:,wordJ);
        dsigInput_r = dsr.*s(:,wordJ-1).*r(:,wordJ).*(1-r(:,wordJ));
        db_r = db_r + dsigInput_r;
        dU_r = dU_r + dsigInput_r*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_r = dW_r + dsigInput_r*s(:,wordJ-1)';
        ds_cur = ds_cur + W_r'*dsigInput_r;

        ds_cur = ds_cur + ds_cur_bk.*z(:,wordJ);
        dz = ds_cur_bk.*(s(:,wordJ-1)-c(:,wordJ));
        dsigInput_z = dz.*z(:,wordJ).*(1-z(:,wordJ));
        db_z = db_z + dsigInput_z;
```

```
        dU_z = dU_z + dsigInput_z*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_z = dW_z + dsigInput_z*s(:,wordJ-1)';
        ds_cur = ds_cur + W_z'*dsigInput_z;
    end

    % s_1
    ds_cur = ds_cur + ds_single(:,1);

    dtanhInput = (ds_cur.*(1-z(:,1)).*(1-c(:,1).*c(:,1)));
    db_c = db_c + dtanhInput;
    dU_c = dU_c + dtanhInput*x(:,1)'; %could be accelerated by avoiding add 0
    dW_c = dW_c + dtanhInput*(s_0.*r(:,1))';
    dsr = W_c'*dtanhInput;
    ds_0 = ds_0 + dsr.*r(:,1);
    dsigInput_r = dsr.*s_0.*r(:,1).*(1-r(:,1));
    db_r = db_r + dsigInput_r;
    dU_r = dU_r + dsigInput_r*x(:,1)'; %could be accelerated by avoiding add 0
    dW_r = dW_r + dsigInput_r*s_0';
    ds_0 = ds_0 + W_r'*dsigInput_r;

    ds_0 = ds_0 + ds_cur.*z(:,1);
    dz = ds_cur.*(s_0-c(:,1));
    dsigInput_z = dz.*z(:,1).*(1-z(:,1));
    db_z = db_z + dsigInput_z;
    dU_z = dU_z + dsigInput_z*x(:,1)'; %could be accelerated by avoiding add 0
    dW_z = dW_z + dsigInput_z*s_0';
    ds_0 = ds_0 + W_z'*dsigInput_z;
end

% A more direct view of backward propagate to calculate gradient using
% chain rule. (O(sentence_size^2) time)
% Instead of calculating how much contribution of derivative each step has,
% here we calculate the gradient within every step.
function [dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0] = ...
    backward_direct(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0)
    % forward propagate to get the intermediate and output results
    [s, y_hat, L, z, r, c] = forward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, ...
        b_z, b_r, b_c, V, b_V, s_0);
    % count sentence size
    [~, sentence_size] = size(x);

    % calculate gradient using chain rule
    delta_y = y_hat - y;
    db_V = sum(delta_y, 2);

    dV = zeros(size(V));
    for wordI = 1:sentence_size
        dV = dV + delta_y(:,wordI)*s(:,wordI)';
    end

    ds_0 = zeros(size(s_0));
    dU_c = zeros(size(U_c));
    dU_r = zeros(size(U_r));
    dU_z = zeros(size(U_z));
    dW_c = zeros(size(W_c));
    dW_r = zeros(size(W_r));
    dW_z = zeros(size(W_z));
```

```matlab
    db_z = zeros(size(b_z));
    db_r = zeros(size(b_r));
    db_c = zeros(size(b_c));
    ds_single = V'*delta_y;
    % calculate the derivatives in each step and add them up
    for wordI = 1:sentence_size
        ds_cur = ds_single(:,wordI);
        % since in each step t, the derivatives depends on s_0 - s_t,
        % we need to trace back from t ot 0 each time
        for wordJ = wordI:-1:2
            ds_cur_bk = ds_cur;

            dtanhInput = (ds_cur.*(1-z(:,wordJ)).*(1-c(:,wordJ).*c(:,wordJ)));
            db_c = db_c + dtanhInput;
            dU_c = dU_c + dtanhInput*x(:,wordJ)'; %could be accelerated by avoiding add 0
            dW_c = dW_c + dtanhInput*(s(:,wordJ-1).*r(:,wordJ))';
            dsr = W_c'*dtanhInput;
            ds_cur = dsr.*r(:,wordJ);
            dsigInput_r = dsr.*s(:,wordJ-1).*r(:,wordJ).*(1-r(:,wordJ));
            db_r = db_r + dsigInput_r;
            dU_r = dU_r + dsigInput_r*x(:,wordJ)'; %could be accelerated by avoiding add 0
            dW_r = dW_r + dsigInput_r*s(:,wordJ-1)';
            ds_cur = ds_cur + W_r'*dsigInput_r;

            ds_cur = ds_cur + ds_cur_bk.*z(:,wordJ);
            dz = ds_cur_bk.*(s(:,wordJ-1)-c(:,wordJ));
            dsigInput_z = dz.*z(:,wordJ).*(1-z(:,wordJ));
            db_z = db_z + dsigInput_z;
            dU_z = dU_z + dsigInput_z*x(:,wordJ)'; %could be accelerated by avoiding add 0
            dW_z = dW_z + dsigInput_z*s(:,wordJ-1)';
            ds_cur = ds_cur + W_z'*dsigInput_z;
        end

        % s_1
        dtanhInput = (ds_cur.*(1-z(:,1)).*(1-c(:,1).*c(:,1)));
        db_c = db_c + dtanhInput;
        dU_c = dU_c + dtanhInput*x(:,1)'; %could be accelerated by avoiding add 0
        dW_c = dW_c + dtanhInput*(s_0.*r(:,1))';
        dsr = W_c'*dtanhInput;
        ds_0 = ds_0 + dsr.*r(:,1);
        dsigInput_r = dsr.*s_0.*r(:,1).*(1-r(:,1));
        db_r = db_r + dsigInput_r;
        dU_r = dU_r + dsigInput_r*x(:,1)'; %could be accelerated by avoiding add 0
        dW_r = dW_r + dsigInput_r*s_0';
        ds_0 = ds_0 + W_r'*dsigInput_r;

        ds_0 = ds_0 + ds_cur.*z(:,1);
        dz = ds_cur.*(s_0-c(:,1));
        dsigInput_z = dz.*z(:,1).*(1-z(:,1));
        db_z = db_z + dsigInput_z;
        dU_z = dU_z + dsigInput_z*x(:,1)'; %could be accelerated by avoiding add 0
        dW_z = dW_z + dsigInput_z*s_0';
        ds_0 = ds_0 + W_z'*dsigInput_z;
    end
end

% Sigmoid function for neural network
function val = sigmoid(x)
```

```
    val = sigmf(x,[1 0]);
end
```

testBPTT_GRU.m

Less Important Functions

```
% Fake a training data set: generate only one sentence for training.
%!!! Only for testing. Needs to be changed to read in training data from files.
function [x_t, y_t] = getTrainingData(vocabulary_size, sentence_size)
    assert(vocabulary_size > 2); % for start and end of sentence symbol
    assert(sentence_size > 0);

    % define start and end of sentence in the vocabulary
    SENTENCE_START = zeros(vocabulary_size, 1);
    SENTENCE_START(1) = 1;
    SENTENCE_END = zeros(vocabulary_size, 1);
    SENTENCE_END(2) = 1;

    % generate sentence:
    x_t = zeros(vocabulary_size, sentence_size-1); % leave one slot for SENTENCE_START
    for wordI = 1:sentence_size-1
        % generate a random word excludes start and end symbol
        x_t(randi(vocabulary_size-2,1,1)+2, wordI) = 1;
    end
    y_t = [x_t, SENTENCE_END];    % training output
    x_t = [SENTENCE_START, x_t]; % training input
end

% Use numerical differentiation to approximate the gradient of each
% parameter and calculate the difference between these numerical results
% and our results calculated by applying chain rule.
function checkGradient_GRU(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0, ...
    dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0)
    % Here we use the centre difference formula:
    %   df(x)/dx = (f(x+h)-f(x-h)) / (2h)
    % It is a second order accurate method with error bounded by O(h^2)

    h = 1e-5;
    % NOTE: h couldn't be too large or too small since large h will
    % introduce bigger truncation error and small h will introduce bigger
    % roundoff error.

    dV_numerical = zeros(size(dV));
    % Calculate partial derivative element by element
    for rowI = 1:size(dV_numerical,1)
        for colI = 1:size(dV_numerical,2)
            V_plus = V;
            V_plus(rowI,colI) = V_plus(rowI,colI) + h;
            V_minus = V;
            V_minus(rowI,colI) = V_minus(rowI,colI) - h;
            [~, ~, L_plus] = forward(x, y, ...
                U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V_plus, b_V, s_0);
            [~, ~, L_minus] = forward(x, y, ...
                U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V_minus, b_V, s_0);
            dV_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
        end
    end
```

```
display(sum(sum(abs(dV_numerical-dV)./(abs(dV_numerical)+h))), ...
    'dV relative error'); % prevent dividing by 0 by adding h

dU_c_numerical = zeros(size(dU_c));
for rowI = 1:size(dU_c_numerical,1)
    for colI = 1:size(dU_c_numerical,2)
        U_c_plus = U_c;
        U_c_plus(rowI,colI) = U_c_plus(rowI,colI) + h;
        U_c_minus = U_c;
        U_c_minus(rowI,colI) = U_c_minus(rowI,colI) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c_plus, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c_minus, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
        dU_c_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
end
display(sum(sum(abs(dU_c_numerical-dU_c)./(abs(dU_c_numerical)+h))), ...
    'dU_c relative error');

dW_c_numerical = zeros(size(dW_c));
for rowI = 1:size(dW_c_numerical,1)
    for colI = 1:size(dW_c_numerical,2)
        W_c_plus = W_c;
        W_c_plus(rowI,colI) = W_c_plus(rowI,colI) + h;
        W_c_minus = W_c;
        W_c_minus(rowI,colI) = W_c_minus(rowI,colI) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c_plus, b_z, b_r, b_c, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c_minus, b_z, b_r, b_c, V, b_V, s_0);
        dW_c_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
end
display(sum(sum(abs(dW_c_numerical-dW_c)./(abs(dW_c_numerical)+h))), ...
    'dW_c relative error');

dU_r_numerical = zeros(size(dU_r));
for rowI = 1:size(dU_r_numerical,1)
    for colI = 1:size(dU_r_numerical,2)
        U_r_plus = U_r;
        U_r_plus(rowI,colI) = U_r_plus(rowI,colI) + h;
        U_r_minus = U_r;
        U_r_minus(rowI,colI) = U_r_minus(rowI,colI) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r_plus, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r_minus, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
        dU_r_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
end
display(sum(sum(abs(dU_r_numerical-dU_r)./(abs(dU_r_numerical)+h))), ...
    'dU_r relative error');
```

```
    dW_r_numerical = zeros(size(dW_r));
    for rowI = 1:size(dW_r_numerical,1)
        for colI = 1:size(dW_r_numerical,2)
            W_r_plus = W_r;
            W_r_plus(rowI,colI) = W_r_plus(rowI,colI) + h;
            W_r_minus = W_r;
            W_r_minus(rowI,colI) = W_r_minus(rowI,colI) - h;
            [~, ~, L_plus] = forward(x, y, ...
                U_z, U_r, U_c, W_z, W_r_plus, W_c, b_z, b_r, b_c, V, b_V, s_0);
            [~, ~, L_minus] = forward(x, y, ...
                U_z, U_r, U_c, W_z, W_r_minus, W_c, b_z, b_r, b_c, V, b_V, s_0);
            dW_r_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
        end
    end
    display(sum(sum(abs(dW_r_numerical-dW_r)./(abs(dW_r_numerical)+h))), ...
        'dW_r relative error');

    dU_z_numerical = zeros(size(dU_z));
    for rowI = 1:size(dU_z_numerical,1)
        for colI = 1:size(dU_z_numerical,2)
            U_z_plus = U_z;
            U_z_plus(rowI,colI) = U_z_plus(rowI,colI) + h;
            U_z_minus = U_z;
            U_z_minus(rowI,colI) = U_z_minus(rowI,colI) - h;
            [~, ~, L_plus] = forward(x, y, ...
                U_z_plus, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
            [~, ~, L_minus] = forward(x, y, ...
                U_z_minus, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
            dU_z_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
        end
    end
    display(sum(sum(abs(dU_z_numerical-dU_z)./(abs(dU_z_numerical)+h))), ...
        'dU_z relative error');

    dW_z_numerical = zeros(size(dW_z));
    for rowI = 1:size(dW_z_numerical,1)
        for colI = 1:size(dW_z_numerical,2)
            W_z_plus = W_z;
            W_z_plus(rowI,colI) = W_z_plus(rowI,colI) + h;
            W_z_minus = W_z;
            W_z_minus(rowI,colI) = W_z_minus(rowI,colI) - h;
            [~, ~, L_plus] = forward(x, y, ...
                U_z, U_r, U_c, W_z_plus, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
            [~, ~, L_minus] = forward(x, y, ...
                U_z, U_r, U_c, W_z_minus, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0);
            dW_z_numerical(rowI,colI) = (sum(L_plus) - sum(L_minus)) / 2 / h;
        end
    end
    display(sum(sum(abs(dW_z_numerical-dW_z)./(abs(dW_z_numerical)+h))), ...
        'dW_z relative error');

    db_z_numerical = zeros(size(db_z));
```

```
    for i = 1:length(db_z_numerical)
        b_z_plus = b_z;
        b_z_plus(i) = b_z_plus(i) + h;
        b_z_minus = b_z;
        b_z_minus(i) = b_z_minus(i) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z_plus, b_r, b_c, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z_minus, b_r, b_c, V, b_V, s_0);
        db_z_numerical(i) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
    display(sum(abs(db_z_numerical-db_z)./(abs(db_z_numerical)+h)), ...
        'db_z relative error');

    db_r_numerical = zeros(size(db_r));
    for i = 1:length(db_r_numerical)
        b_r_plus = b_r;
        b_r_plus(i) = b_r_plus(i) + h;
        b_r_minus = b_r;
        b_r_minus(i) = b_r_minus(i) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r_plus, b_c, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r_minus, b_c, V, b_V, s_0);
        db_r_numerical(i) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
    display(sum(abs(db_r_numerical-db_r)./(abs(db_r_numerical)+h)), ...
        'db_r relative error');

    db_c_numerical = zeros(size(db_c));
    for i = 1:length(db_c_numerical)
        b_c_plus = b_c;
        b_c_plus(i) = b_c_plus(i) + h;
        b_c_minus = b_c;
        b_c_minus(i) = b_c_minus(i) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c_plus, V, b_V, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c_minus, V, b_V, s_0);
        db_c_numerical(i) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
    display(sum(abs(db_c_numerical-db_c)./(abs(db_c_numerical)+h)), ...
        'db_c relative error');

    db_V_numerical = zeros(size(db_V));
    for i = 1:length(db_V_numerical)
        b_V_plus = b_V;
        b_V_plus(i) = b_V_plus(i) + h;
        b_V_minus = b_V;
        b_V_minus(i) = b_V_minus(i) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V_plus, s_0);
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V_minus, s_0);
        db_V_numerical(i) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
    display(sum(abs(db_V_numerical-db_V)./(abs(db_V_numerical)+h)), ...
```

```
        'db_V relative error');

    ds_0_numerical = zeros(size(ds_0));
    for i = 1:length(ds_0_numerical)
        s_0_plus = s_0;
        s_0_plus(i) = s_0_plus(i) + h;
        s_0_minus = s_0;
        s_0_minus(i) = s_0_minus(i) - h;
        [~, ~, L_plus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0_plus
    );
        [~, ~, L_minus] = forward(x, y, ...
            U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V,
    s_0_minus);
        ds_0_numerical(i) = (sum(L_plus) - sum(L_minus)) / 2 / h;
    end
    display(sum(abs(ds_0_numerical-ds_0)./(abs(ds_0_numerical)+h)), ...
        'ds_0 relative error');
end
```

testBPTT_GRU.m