CS5242_Assignment_3

November 2, 2017

Updates from V1

Change the descriptions and corresponding codes in section 5, set the construction of brnn as optional. If you want to try brnn, please implement the imcomplete functions in "code_base/classifiers/brnn.py".

1 Task Description

ASSIGNMENT DEADLINE: 24 NOV 2017 (FRI) 11.59PM

In this assignment, you will need to implement a bi-directional recurrent neural network and use it to train a model that can generate sentiment of a sentence. The dataset we use comes from a Kaggle competition for sentiment analysis, see the full dataset on https://www.kaggle.com/c/si650winter11/data. We only sample 1000 sentences from the original training set as our dataset. train.csv and test.csv are generated with the ratio of 4:1 from the 1000 samples.

We have provided APIs for loading dataset, building word dictionary and convert sentence into one-hot vector, you can just call the functions from "code_base/data_utils.py" to do these things. You may notice that the word dictionary may contain strange words and symbols. This happens since we adopt python "nltk" package and a simple corpus to help tokenize the sentences. More elegant ways might be adopted for tokenization, but it is not the main focus of this assignment.

For submission, the submission format will need to be in output text form (similar to the previous assignment). For each question, we will provide the input arguments and you have to provide a text file containing the corresponding output. We will check your output with our standard solution, the difference should be within the indicated range.

This iPython notebook serves to: - explain the questions - explain the function APIs - providing helper functions to piece functions together and check your code - providing helper functions to load and save arrays as csv file for submission. Hence, we strongly encourage you to use Python for this assignment as you will only need to code the relevant parts and it will reduce your workload significantly. For non-Python users, some of the cells here are for illustration purpose, you do not have to replicate the demos.

The input file will be in the **input_files** folder, and your output files should go into **output_files** folder. Output files are csv files where each file contain one type of result, e.g. one output of a layer or gradients of one type of parameter. Similar to previous assignments, use np.float32 if you are using Python and use **at least 16 significant figures** for your outputs. For Python users, if you use the accompanying printing functions, it should be ok.

2 RNN Step Forward/Backward

2.1 Step Forward

Open the file "code_base/rnn_layers.py". This file implements the forward and backward passes for different types of layers that are commonly used in recurrent neural networks.

First implement the function rnn_step_forward which implements the forward pass for a single time step of a vanilla recurrent neural network. Notice that there is a **tanh activation function** inside an RNN layer. After doing so run the following to check your implementation. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import rnn_step_forward
from code_base.layer_utils import rel_error

N, D, H = 3, 10, 4
x = np.linspace(-0.4, 0.7, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.2, 0.5, num=N*H).reshape(N, H)
Wx = np.linspace(-0.1, 0.9, num=D*H).reshape(D, H)
Wh = np.linspace(-0.3, 0.7, num=H*H).reshape(H, H)
b = np.linspace(-0.2, 0.4, num=H)
next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
expected_next_h = np.asarray([
  [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
  [ 0.66854692, 0.79562378, 0.87755553, 0.92795967],
  [ 0.97934501, 0.99144213, 0.99646691, 0.99854353]])
print('next_h error: ', rel_error(expected_next_h, next_h))
```

For submission: Submit the corresponding output from your step forward for the given input arguments. Load the files **Wx.csv**, **Wh.csv**, **b.csv**, **x.csv** and **prev_h.csv**, they contain the input arguments for the **Wx**, **Wh**, **b**, **x** and **prev_h** respectively and are flattened to a 1D array in C-style, row-major order (see numpy.ravel for details: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ravel.html).
Assume N=3, D=874, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your rnn_step_forward function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file rnn_step_forward_out.csv which contains the flattened output of rnn_step_forward.
The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In [ ]:
```
from code_base.rnn_layers import rnn_step_forward
```

```
import numpy as np

x_shape = (3, 874)
Wx_shape = (874, 128)
h_shape = (3, 128)
Wh_shape = (128, 128)
b_shape = (128,)
x = np.loadtxt('./input_files/x.csv', delimiter=',')
x = x.reshape(x_shape)
Wx = np.loadtxt('./input_files/Wx.csv', delimiter=',')
Wx = Wx.reshape(Wx_shape)
prev_h = np.loadtxt('./input_files/prev_h.csv', delimiter=',')
prev_h = prev_h.reshape(h_shape)
Wh = np.loadtxt('./input_files/Wh.csv', delimiter=',')
Wh = Wh.reshape(Wh_shape)
b = np.loadtxt('./input_files/b.csv', delimiter=',')
out, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
np.savetxt('./output_files/rnn_step_forward_out.csv', out.ravel(), delimiter=',')
```

2.2 Step Backward

In the file “code_base/rnn_layers.py” , implement the rnn_step_backward function. After doing so run the following to numerically gradient check your implementation. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import rnn_step_forward, rnn_step_backward
from code_base.gradient_check import *
import numpy as np

N, D, H = 4, 5, 6
x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)
out, cache = rnn_step_forward(x, h, Wx, Wh, b)
dnext_h = np.random.randn(*out.shape)
fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
fh = lambda prev_h: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b)[0]
fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0]
dx_num = eval_numerical_gradient_array(fx, x, dnext_h)
dprev_h_num = eval_numerical_gradient_array(fh, h, dnext_h)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
db_num = eval_numerical_gradient_array(fb, b, dnext_h)
dx, dprev_h, dWx, dWh, db = rnn_step_backward(dnext_h, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dprev_h error: ', rel_error(dprev_h_num, dprev_h))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

For submission: Submit the corresponding output from your step backward for the given input arguments. Load the files **Wx.csv**, **Wh.csv**, **b.csv**, **x.csv**, **prev_h.csv** and **dho.csv**, they contain the input arguments for the **Wx**, **Wh**, **b**, **x**, **prev_h** and **dhout** respectively and are flattened to a 1D array in C-style, row-major order.
Assume N=3, D=874, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your rnn_step_backward function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file rnn_step_backward_out_xx.csv (xx means the parameter name) which contains the flattened output of rnn_step_backward. The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In[ ]:
```
from code_base.rnn_layers import rnn_step_forward, rnn_step_backward
import numpy as np

x_shape = (3, 874)
Wx_shape = (874, 128)
h_shape = (3, 128)
Wh_shape = (128, 128)
b_shape = (128,)
x = np.loadtxt('./input_files/x.csv', delimiter=',')
x = x.reshape(x_shape)
Wx = np.loadtxt('./input_files/Wx.csv', delimiter=',')
Wx = Wx.reshape(Wx_shape)
prev_h = np.loadtxt('./input_files/prev_h.csv', delimiter=',')
prev_h = prev_h.reshape(h_shape)
Wh = np.loadtxt('./input_files/Wh.csv', delimiter=',')
Wh = Wh.reshape(Wh_shape)
b = np.loadtxt('./input_files/b.csv', delimiter=',')
out, cache = rnn_step_forward(x, prev_h, Wx, Wh, b)
dhout = np.loadtxt('./input_files/dhout.csv', delimiter=',')
dx, dh, dWx, dWh, db = rnn_step_backward(dhout, cache)
np.savetxt('./output_files/rnn_step_backward_out_dx.csv', dx.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_step_backward_out_dh.csv', dh.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_step_backward_out_dwx.csv', dWx.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_step_backward_out_dwh.csv', dWh.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_step_backward_out_db.csv', db.ravel(), delimiter=',')
```

3 RNN Forward/Backward

3.1 Forward

Now that you have implemented the forward and backward passes for a single time step of a vanilla RNN, you will combine these pieces to implement a RNN that process an entire sequence of data.
In the file "code_base/rnn_layers.py", implement the function rnn_forward. This should be implemented using the rnn_step_forward function that you defined above.

After doing so run the following to check your implementation. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import rnn_forward
from code_base.gradient_check import *
import numpy as np

N, T, D, H = 2, 3, 4, 5
x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)
b = np.linspace(-0.7, 0.1, num=H)
h, _ = rnn_forward(x, h0, Wx, Wh, b)
expected_h = np.asarray([
  [[-0.42070749, -0.27279261, -0.11074945, 0.05740409, 0.22236251],
  [-0.39525808, -0.22554661, -0.0409454, 0.14649412, 0.32397316],
  [-0.42305111, -0.24223728, -0.04287027, 0.15997045, 0.35014525],],
  [[-0.55857474, -0.39065825, -0.19198182, 0.02378408, 0.23735671],
  [-0.27150199, -0.07088804, 0.13562939, 0.33099728, 0.50158768],
  [-0.51014825, -0.30524429, -0.06755202, 0.17806392, 0.40333043]]])
print('h error: ', rel_error(expected_h, h))
```

For submission: Submit the corresponding output from your forward for the given input arguments. Load the files **Wx.csv**, **Wh.csv**, **b.csv**, **x_all.csv** and **prev_h.csv**, they contain the input arguments for the **Wx**, **Wh**, **b**, **x**(contains xi at each timestep, where i is in the range of [0, T]) and **prev_h** (the initial hidden state) respectively and are flattened to a 1D array in C-style, row-major order.
Assume N=3, T=4, D=874, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your rnn_forward function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file rnn_forward_out.csv which contains the flattened output of rnn_forward. The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In[ ]:
```
from code_base.rnn_layers import rnn_forward
import numpy as np

x_all_shape = (3, 5, 874)
Wx_shape = (874, 128)
h_shape = (3, 128)
Wh_shape = (128, 128)
b_shape = (128,)
x_all = np.loadtxt('./input_files/x_all.csv', delimiter=',')
x_all = x_all.reshape(x_all_shape)
Wx = np.loadtxt('./input_files/Wx.csv', delimiter=',')
Wx = Wx.reshape(Wx_shape)
prev_h = np.loadtxt('./input_files/prev_h.csv', delimiter=',')
prev_h = prev_h.reshape(h_shape)
```

```
Wh = np.loadtxt('./input_files/Wh.csv', delimiter=',')
Wh = Wh.reshape(Wh_shape)
b = np.loadtxt('./input_files/b.csv', delimiter=',')
out, _ = rnn_forward(x_all, prev_h, Wx, Wh, b)
np.savetxt('./output_files/rnn_forward_out.csv', out.ravel(), delimiter=',')
```

3.2 Backward

In the file "code_base/rnn_layers.py", implement the backward pass for a vanilla RNN in the function rnn_backward. This should run back-propagation over the entire sequence, calling into the rnn_step_backward function that you defined above. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import rnn_forward, rnn_backward
from code_base.gradient_check import *
from code_base.layer_utils import *
import numpy as np

N, D, T, H = 2, 3, 10, 5
x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)
out, cache = rnn_forward(x, h0, Wx, Wh, b)
dout = np.random.randn(*out.shape)
dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)
fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]
dx_num = eval_numerical_gradient_array(fx, x, dout)
dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)
print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

For submission: Submit the corresponding output from your backward for the given input arguments. Load the files **Wx.csv**, **Wh.csv**, **b.csv**, **x_all.csv**, **prev_h.csv** and **dho_all.csv**, they contain the input arguments for the **Wx**, **Wh**, **b**, **x**(contains xi at each timestep, where i is in the range of [0, T]), **prev_h** and **dhout**(at all timesteps) respectively and are flattened to a 1D array in C-style, row-major order.

Assume N=3, T=4, D=874, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your **rnn_backward** function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file **rnn_backward_out_xx.csv** (xx means the parameter name) which contains the flattened output of **rnn_backward**. The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In[ ]:
```
from code_base.rnn_layers import rnn_forward, rnn_backward
import numpy as np

x_all_shape = (3, 5, 874)
Wx_shape = (874, 128)
h_shape = (3, 128)
Wh_shape = (128, 128)
b_shape = (128,)
dh_all_shape = (3, 5, 128)
x_all = np.loadtxt('./input_files/x_all.csv', delimiter=',')
x_all = x_all.reshape(x_all_shape)
Wx = np.loadtxt('./input_files/Wx.csv', delimiter=',')
Wx = Wx.reshape(Wx_shape)
h0 = np.loadtxt('./input_files/prev_h.csv', delimiter=',')
h0 = h0.reshape(h_shape)
Wh = np.loadtxt('./input_files/Wh.csv', delimiter=',')
Wh = Wh.reshape(Wh_shape)
b = np.loadtxt('./input_files/b.csv', delimiter=',')
out, cache = rnn_forward(x_all, h0, Wx, Wh, b)
dhout = np.loadtxt('./input_files/dho_all.csv', delimiter=',')
dhout = dhout.reshape(dh_all_shape)
dx_all, dh0, dWx, dWh, db = rnn_backward(dhout, cache)
np.savetxt('./output_files/rnn_backward_out_dx.csv', dx_all.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_backward_out_dh0.csv', dh0.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_backward_out_dwx.csv', dWx.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_backward_out_dwh.csv', dWh.ravel(), delimiter=',')
np.savetxt('./output_files/rnn_backward_out_db.csv', db.ravel(), delimiter=',')
```

4 (Optional) Temporal Bi-directional Concatenation Layer

At every time step, we use a concatenation function as the output of the bi-directional RNN, which concatenates the RNN hidden vector of the normal sentence input and that of the reversed sentence input (see Lecture Note 8 for more details). This question is optional and is with 0 mark.

4.1 Forward

In the file "code_base/rnn_layers.py", implement the forward pass for the concatenation in the function **bidirectional_rnn_concatenate_forward**. For 0-padding positions, w.r.t. value 0 in the mask, the concatenation result will be a 0 array. **For both**

normal RNN (train on normal sentence sequences) and reversed RNN (train on reversed sentence sequences), paddings are at the end of the sentence sequence. For example, [s1, s2, s3, 0] and [s3, s2, s1, 0] as one sample fed to two RNNs respectively. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import bidirectional_rnn_concatenate_forward
from code_base.layer_utils import *
import numpy as np

N, T, H = 2, 4, 3
h = np.linspace(-0.5, 0, num=N*T*H).reshape(N, T, H)
hr = np.linspace(0, 0.5, num=N*T*H).reshape(N, T, H)
mask = np.ones((N,T))
mask[0][3] = 0 # length of s1 is 3
mask[1][2] = mask[1][3] = 0 # length of s2 is 2
ho, _ = bidirectional_rnn_concatenate_forward(h, hr, mask)
expected_ho = np.array([[
  [-0.5, -0.47826087, -0.45652174, 0.13043478, 0.15217391, 0.17391304],
  [-0.43478261, -0.41304348, -0.39130435, 0.06521739, 0.08695652, 0.10869565],
  [-0.36956522, -0.34782609, -0.32608696, 0., 0.02173913, 0.04347826],
  [0., 0., 0., 0., 0., 0.]],
  [[-0.23913043, -0.2173913 , -0.19565217, 0.32608696, 0.34782609, 0.36956522],
  [-0.17391304, -0.15217391, -0.13043478, 0.26086957, 0.2826087, 0.30434783],
  [0., 0., 0., 0., 0., 0.],
  [0., 0., 0., 0., 0., 0.]]])
print('ho error: ', rel_error(expected_ho, ho, mask))
```

For submission: Submit the corresponding output from your backward for the given input arguments. Load the files **h_all.csv**, **h_all_r.csv** and **mask.csv**, they contain the input arguments for the **h**, **hr** and **mask** respectively and are flattened to a 1D array in C-style, row-major order.
Assume N=3, T=4, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your bidirectional_rnn_concatenate_forward function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file bidirectional_rnn_concatenate_forward_out.csv which contains the flattened output of bidirectional_rnn_concatenate_forward. The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In[ ]:
```
from code_base.rnn_layers import bidirectional_rnn_concatenate_forward
import numpy as np

h_shape = (3, 5, 128)
mask_shape = (3, 5)
h = np.loadtxt('./input_files/h_all.csv', delimiter=',')
h = h.reshape(h_shape)
hr = np.loadtxt('./input_files/h_all_r.csv', delimiter=',')
hr = hr.reshape(h_shape)
mask = np.loadtxt('./input_files/mask.csv', delimiter=',')
mask = mask.reshape(mask_shape)
```

```
hout, _ = bidirectional_rnn_concatenate_forward(h, hr, mask)
np.savetxt('./output_files/bidirectional_rnn_concatenate_forward_out.csv',
hout.ravel(), delimiter=',')
```

4.2 Backward

In the file "code_base/rnn_layers.py" , implement the backward pass for a bi-directional RNN in the function bidirectional_rnn_concatenate_backward. **Notice that mask might be useful in the backward pass**. You should see errors less than 1e-7.

In [ ]:
```
from code_base.rnn_layers import bidirectional_rnn_concatenate_forward,
bidirectional_rnn_concatenate_backward
from code_base.layer_utils import *
import numpy as np

N, T, H = 2, 4, 3
h = np.linspace(-0.5, 0, num=N*T*H).reshape(N, T, H)
hr = np.linspace(0, 0.5, num=N*T*H).reshape(N, T, H)
mask = np.ones((N,T))
mask[0][3] = 0 # length of s1 is 3
mask[1][2] = mask[1][3] = 0 # length of s2 is 2
ho, cache = bidirectional_rnn_concatenate_forward(h, hr, mask)
dho = np.linspace(0., 0.5, num=N*T*2*H).reshape(N, T, 2*H)
dh, dhr = bidirectional_rnn_concatenate_backward(dho, cache)
expected_dh = np.array([
 [[ 0., 0.0106383,  0.0212766 ],
  [ 0.06382979, 0.07446809, 0.08510638],
  [ 0.12765957, 0.13829787, 0.14893617],
  [ 0., 0., 0.]],
 [[ 0.25531915, 0.26595745, 0.27659574],
  [ 0.31914894, 0.32978723, 0.34042553],
  [ 0., 0., 0.],
  [ 0., 0., 0.]]])
expected_dhr = np.array([
 [[ 0.15957447,  0.17021277, 0.18085106],
  [ 0.09574468,  0.10638298, 0.11702128],
  [ 0.03191489,  0.04255319, 0.05319149],
  [ 0.,  0., 0.]],
 [[ 0.35106383,  0.36170213, 0.37234043],
  [ 0.28723404,  0.29787234, 0.30851064],
  [ 0., 0., 0.],
  [ 0., 0., 0.]]])
print('dh error: ', rel_error(expected_dh, dh, mask))
print('dhr error: ', rel_error(expected_dhr, dhr, mask))
```

For submission: Submit the corresponding output from your backward for the given input arguments. Load the files **h_all.csv**, **h_all_r.csv**, **mask.csv** and **dho.csv**, they contain the input arguments for the **h**, **hr**, **masks** and **dho** respectively and are flattened to a 1D array in C-style, row-major order.

Assume N=3, T=4, H=128. For Python users, you can use the code below to load and reshape the arrays to feed into your bidirectional_rnn_backward function. Code is also provided to flatten the array and save your output to a csv file. For users of other programming languages, you have to submit the output file bidirectional_rnn_backward_out_xx.csv (xx means the parameter name) which contains the flattened output of bidirectional_rnn_backward. The array must be fattened in row-major order or else our automated scripts will mark your outputs as incorrect.

In[ ]:
```
from code_base.rnn_layers import bidirectional_rnn_concatenate_forward,
bidirectional_rnn_concatenate_backward
import numpy as np

h_shape = (3, 5, 128)
mask_shape = (3, 5)
h = np.loadtxt('./input_files/h_all.csv', delimiter=',')
h = h.reshape(h_shape)
hr = np.loadtxt('./input_files/h_all_r.csv', delimiter=',')
hr = hr.reshape(h_shape)
mask = np.loadtxt('./input_files/mask.csv', delimiter=',')
mask = mask.reshape(mask_shape)
hout, cache = bidirectional_rnn_concatenate_forward(h, hr, mask)
dhout = np.loadtxt('./input_files/dhc_all.csv', delimiter=',')
dhout = dhout.reshape(3, 5, 256)
dh, dhr = bidirectional_rnn_concatenate_backward(dhout, cache)
np.savetxt('./output_files/bidirectional_rnn_concatenate_backward_out_h.csv',
dh.ravel(), delimiter=',')
np.savetxt('./output_files/bidirectional_rnn_concatenate_backward_out_hr.csv',
dhr.ravel(), delimiter=',')
```

5 RNN for Sentiment Analysis

Now that you have implemented the necessary layers, you can combine them to build a sentiment analysis model. Open the file "code_base/classifiers/rnn.py" and look at the SentimentAnalysisRNN class. **Implement the forward and backward pass of the model in the loss function** of the file "code_base/classifiers/rnn.py". If you complete bidirectional_rnn_concatenate_forward and bidirectional_rnn_concatenate_backward functions, you can try constructing a bi-directional rnn for training and inference in file "code_base/classifiers/brnn.py" (this is **optional**). Since their constructions are slightly different, the output loss will be different too.

5.1 Forward Pass

After you implement the loss function, run the following to check your forward pass using the fixed a small test case; you should see error less than 1e-7.

In [ ]:
```
from code_base.classifiers.rnn import *
```

```
# If you do brnn, please import from code_base.classifiers.brnn instead
import numpy as np

N, H, A, O = 2, 6, 5, 2
word_to_idx = { 'awesome': 0, 'reading':1, 'pretty': 2, 'dog': 3, 'movie': 4,
            'liked': 5, 'most': 6, 'admired': 7, 'bad': 8, 'fucking': 9}
V = len(word_to_idx)
T = 4
model = SentimentAnalysisRNN(word_to_idx,
   hidden_dim=H,
   fc_dim=A,
   output_dim=O,
   cell_type='rnn',
   dtype=np.float64)
# Set all model parameters to fixed values
for k, v in model.params.items():
   model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)
labels = np.array([1, 0], dtype=np.int32)
wordvecs = np.zeros((N, T, V))
wordvecs[0, 0, 0] = wordvecs[0, 1, 5] = wordvecs[0, 2, 2] = wordvecs[0, 3, 7] = 1
wordvecs[1, 0, 4] = wordvecs[1, 1, 8] = wordvecs[1, 2, 5] = 1
mask = np.ones((N, T))
mask[1, 3] = 0
print(wordvecs.shape, labels.shape, mask.shape)
loss, grads = model.loss(wordvecs, labels, mask)
expected_loss = 2.99619226823

# For brnn, the expected_loss should be 2.9577205234
print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

5.2 Backward Pass

Run the following cell to perform numeric gradient checking on the SentimentAnalysisRNN class. You should see errors around 5e-6 or less.

In [ ]:
```
from code_base.classifiers.rnn import *
# If you do brnn, please import from code_base.classifiers.brnn instead
from code_base.gradient_check import *
from code_base.layer_utils import rel_error
import numpy as np

N, T, H, A, O = 2, 4, 6, 5, 2
word_to_idx = {'awesome': 0, 'reading':1, 'pretty': 2, 'dog': 3, 'movie': 4,
            'liked': 5, 'most': 6, 'admired': 7, 'bad': 8, 'fucking': 9}
V = len(word_to_idx)
labels = np.array([1, 0], dtype=np.int32)
wordvecs = np.zeros((N, T, V))
wordvecs[0, 0, 0] = wordvecs[0, 1, 5] = wordvecs[0, 2, 2] = wordvecs[0, 3, 7] = 1
wordvecs[1, 0, 4] = wordvecs[1, 1, 8] = wordvecs[1, 2, 5] = 1
mask = np.ones((N, T))
```

```
mask[1, 3] = 0
model = SentimentAnalysisRNN(word_to_idx,
    hidden_dim=H,
    fc_dim=A,
    output_dim=O,
    cell_type='rnn',
    dtype=np.float64,
)
loss, grads = model.loss(wordvecs, labels, mask)
for param_name in sorted(grads):
    f = lambda _: model.loss(wordvecs, labels, mask)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],
verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

5.3 Training/Inference on Small Data

Similar to the Solver class that we used to train image classification models on the previous assignment, on this assignment we use a SentimentAnalysisSolver class to train sentiment analysis models. Open the file "code_base/sentiment_analysis_solver.py" and read through the SentimentAnalysisSolver class; it should look very familiar. In this part, you will need to **implement the forward pass of the model in the inference function** of the file "code_base/classifiers/rnn.py". Once you have done, you will need to train your model on a small sample of 100 training examples for a few iterations. After that, you will need to use the trained model to generate the probability distribution over the sentiment class. The following codes are given for the purpose.

In[ ]:

```
from code_base.sentiment_analysis_solver import *
from code_base.classifiers.rnn import *
# If you do brnn, please import from code_base.classifiers.brnn instead
from code_base.data_utils import load_data, load_dictionary, sample_minibatch
import matplotlib.pyplot as plt
import numpy as np

small_data = load_data('code_base/datasets/train.csv', sample=True)
small_rnn_model = SentimentAnalysisRNN(
    cell_type='rnn',
    word_to_idx=load_dictionary('code_base/datasets/dictionary.csv')
)
small_rnn_solver = SentimentAnalysisSolver(small_rnn_model,
    small_data,
    update_rule='sgd',
    num_epochs=100,
    batch_size=100,
    optim_config={
        'learning_rate': 1e-3,
    },
    lr_decay=1.0,
    verbose=True,
```

```
    print_every=10,
)
small_rnn_solver.train()

# we will use the same batch of training data for inference
# this is just to let you know the procedure of inference
preds = small_rnn_solver.test(split='train')
np.savetxt('./output_files/rnn_prediction_prob.csv', preds.ravel(), delimiter=',')
# If you do brnn, please save the result to './output_files/brnn_prediction_prob.csv'

# Plot the training losses
plt.plot(small_rnn_solver.loss_history)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training loss history')
plt.show()
```

For submission: the above script will generate the figure of training loss and save the inference results to files. You will need to save the figure of training loss to the output_files folder for submission.

Important note: you don' t need to tune any hyper-parameters for training the model since all of the hyper-parameters and even the parameters are initialized with the fixed values. The main purpose of this section is to test if you can correctly construct the training/inference procedure of the network, rather than testing how good performance you can achieve for the desired task.

6 Final submission instructions

Please submit the following:

1) Your code files in the folder code_base
2) Output files to the functions in the folder output_files
3) A short report (No more than 2 pages) in pdf titled report.pdf, explaining the logic (expressed using mathematical expressions) behind coding each function, and the output loss values from 5.3.