rnn

April 4, 2019

1 Recurrent Neural Network

In this task, we implement RNN cells to understand the computation of RNN. Then we build RNN with different cells for a language modeling task.

In [14]:
```
# As usual, a bit of setup
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 Recurrent Neural Networks

1.1.1 A toy problem

In [2]:
```
## Setup an example. Provide sizes and the input data.

# set sizes
time_steps = 5
```

```
batch_size = 4
input_size = 3
hidden_size = 2

# create input data with shape [batch_size, time_steps, num_features]
np.random.seed(15009)
input_data = np.random.rand(batch_size, time_steps, input_size).astype(np.float32)
```

1.1.2 Implement an RNN and a GRU with tensorflow

In [3]:
```
## Create an RNN model
tf.reset_default_graph()
tf.set_random_seed(15009)

# initialize a state of zero for both RNN and GRU
# 'state' is a tensor of shape [batch_size, hidden_size]
init_state = np.zeros([batch_size, hidden_size])
initial_state = tf.Variable(init_state, dtype=tf.float32)

# create a BasicRNNCell
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(hidden_size)


# 'outputs' is a tensor of shape [batch_size, max_time, hidden_size]
# RNN cell outputs the hidden state directly, so the output at each step is the hidden
# final_state is the last state of the sequence. final_state == outputs[:, -1, :]
rnn_outputs, rnn_final_state = tf.nn.dynamic_rnn(rnn_cell, input_data,
                                   initial_state=initial_state,
                                   dtype=tf.float32)


# create a GRUCell
gru_cell = tf.nn.rnn_cell.GRUCell(hidden_size)

# 'outputs' is a tensor of shape [batch_size, time_steps, hidden_size]
# Same as the basic RNN cell, final_state == outputs[:, -1, :]
gru_outputs, gru_final_state = tf.nn.dynamic_rnn(gru_cell, input_data,
                                   initial_state=initial_state,
                                   dtype=tf.float32)



# initialize variables
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)

# run the RNN model and get outputs and the final state
```

```
tfrnn_outputs, tfrnn_final_state = session.run([rnn_outputs, rnn_final_state])

# run the GRU model and get outputs and the final state
tfgru_outputs, tfgru_final_state = session.run([gru_outputs, gru_final_state])
```

```
WARNING:tensorflow:From <ipython-input-3-f13d312c1fa1>:11: BasicRNNCell.__init__ (from tensorfl
Instructions for updating:
This class is equivalent as tf.keras.layers.SimpleRNNCell, and will be replaced by that in Tens
```

1.1.3 Read out parameters from RNN and GRU cells

In [4]:
```
from rnn_param_helper import get_rnn_params, get_gru_params

##########################################################################################
# 0. Understand TF BasicRNN and GRU parameters (0 points)
# Please read the code and documentation of get_rnn_params and get_gru_params to under
# what are these parameters. You will need to use these parameters in your own impleme
# NO implementation is needed here.
##########################################################################################

wt_h, wt_x, bias = get_rnn_params(rnn_cell, session)
wtu_h, wtu_x, biasu, wtr_h, wtr_x, biasr, wtc_h, wtc_x, biasc = get_gru_params(gru_cel
```

1.1.4 Numpy Implementation

Implement your own RNN model with numpy. Your implementation needs to match the tensorflow calculation.

In [5]:
```
##########################################################################################
# 1. Implementation (9 points)
# Implement your basic RNN and GRU RNN in implementations/rnn.py and compare them agai
# tensorflow functions
##########################################################################################

from implementations.rnn import rnn,gru

# calculation from your own implemenation of a basic RNN
nprnn_outputs, nprnn_final_state = rnn(wt_h, wt_x, bias, init_state, input_data)

print("Difference between your RNN implementation and tf RNN",
                     rel_error(tfrnn_outputs, nprnn_outputs) + rel_error(tfrnn_final_s

# calculation from your own implemenation of a GRU RNN
npgru_outputs, npgru_final_state = gru(wtu_h, wtu_x, biasu, wtr_h, wtr_x, biasr, wtc_h

print("Difference between your GRU implementation and tf GRU",
      rel_error(tfgru_outputs, npgru_outputs) + rel_error(tfgru_final_state, npgru_fina
```

```
Difference between your RNN implementation and tf RNN 2.0376387104263702e-07
Difference between your GRU implementation and tf GRU 6.114071923523539e-07
```

1.1.5 GRU includes RNN as a special case

Can you assign a special set of parameters to GRU such that its outputs is almost the same as RNN?

```
In [6]: # Assign some value to a parameter of GRU

        from rnn_param_helper import *

        ####################################################################################
        # 2. Setting GRU weights (4 points)                                                #
        # Get weights/bias from the basic RNN and set them to some GRU weights/bias        #
        # Then set some other parameter of GRU, then GRU recovers RNN.                     #
        ####################################################################################

        #session.run(gru_cell.weights[0].assign(np.zeros((5,4))))
        #session.run(gru_cell.weights[1].assign(np.ones(4)))
        #session.run(gru_cell.weights[2].assign(rnn_cell.get_weights()[0]))
        #session.run(gru_cell.weights[3].assign(rnn_cell.get_weights()[1]))

        set_gru_params(gru_cell,session,
                       wtu_h = np.zeros((hidden_size,hidden_size)),
                       wtu_x = np.zeros((input_size,hidden_size)),
                       biasu = np.ones(hidden_size)*-np.inf,
                       wtr_h = np.zeros((hidden_size,hidden_size)),
                       wtr_x = np.zeros((input_size,hidden_size)),
                       biasr = np.ones(hidden_size)*np.inf,
                       wtc_h = wt_h,
                       wtc_x = wt_x,
                       biasc = bias,
                      )

        # outputs from the GRU with special parameters.
        updated_outputs = session.run(gru_outputs)

        # they are the same as the calculation from the basic RNN
        print("Difference between RNN and a special GRU", rel_error(tfrnn_outputs, updated_outp
```

```
Difference between RNN and a special GRU 0.0
```

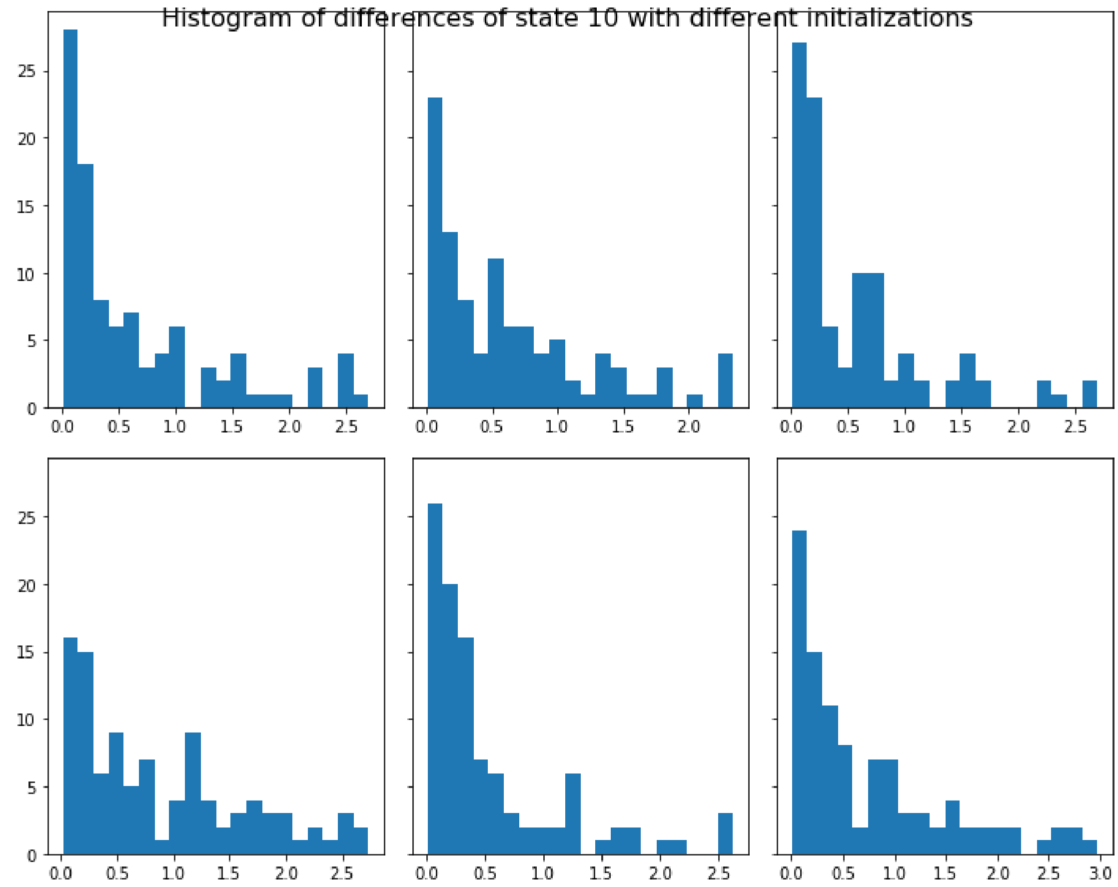
1.2 Long term dependency: forward

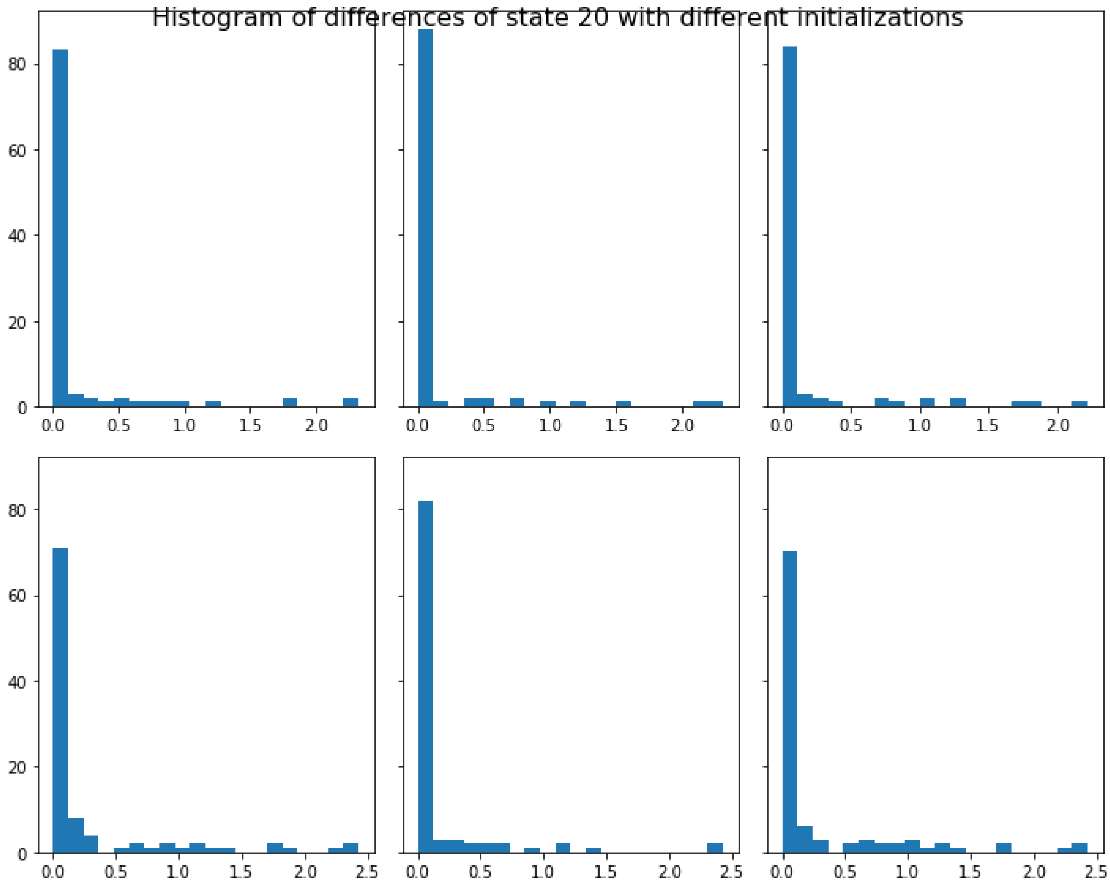
In this experiment, you will see that the basic RNN model is hard to keep long term dependency

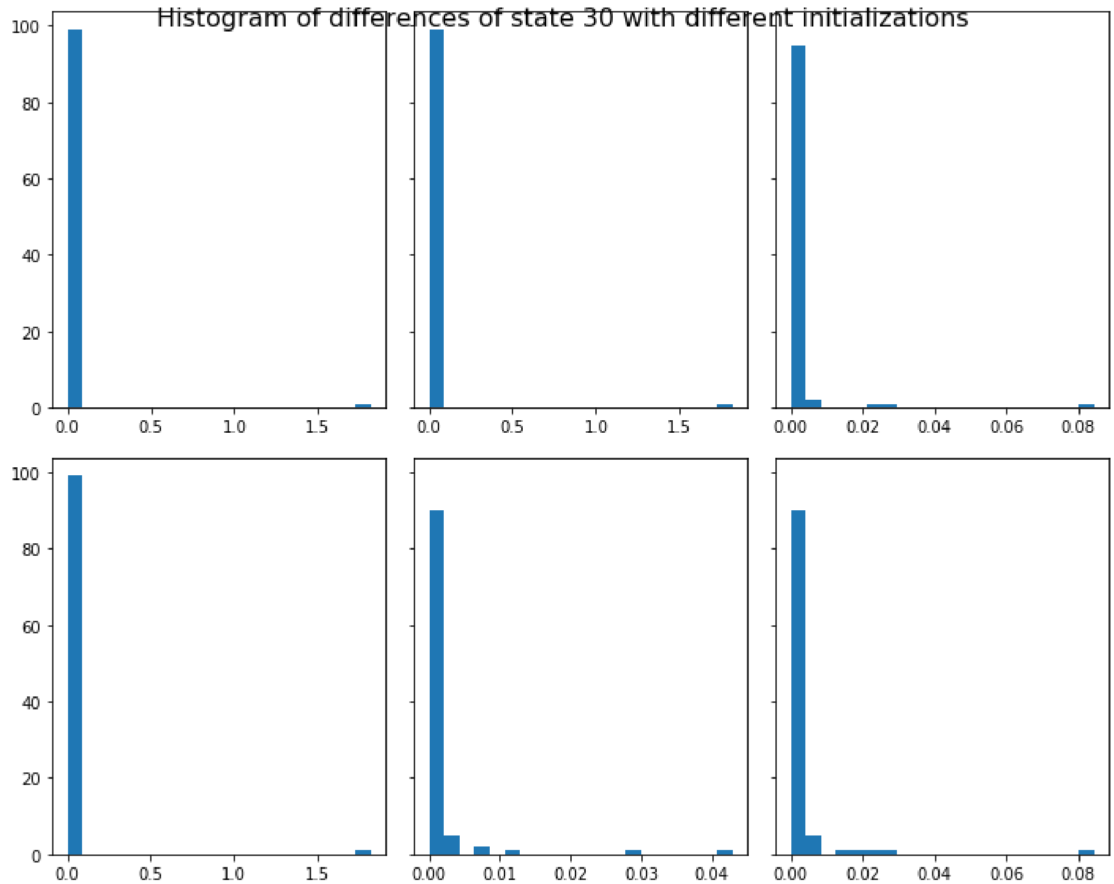
In [16]:
```
from rnn_param_helper import set_rnn_params, set_gru_params

# Create a larger problem

# set sizes
time_steps = 50
batch_size = 100
input_size = 5
hidden_size = 8

# create input data with shape [batch_size, time_steps, num_features]
np.random.seed(15009)
input_data = np.random.rand(batch_size, time_steps, input_size).astype(np.float32) - (

## Create an RNN model with GRU
tf.reset_default_graph()
tf.set_random_seed(15009)


# create a GRUCell
gru_cell = tf.nn.rnn_cell.GRUCell(hidden_size)
# create a BasicRNNCell
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(hidden_size)

# copy the basic RNN and the GRU RNN above here:

initial_state = tf.Variable(np.zeros([batch_size, hidden_size]), dtype=tf.float32)

##############################################################################
# 3. Apply TF RNN functions (2 points)
# Please use the tensorflow function for the basic RNN and the GRU RNN below to get t
# from the larger problem. Basically you just need to copy some code above here.
##############################################################################

rnn_outputs, _ = tf.nn.dynamic_rnn(rnn_cell, input_data, initial_state=initial_state,
gru_outputs, _ = tf.nn.dynamic_rnn(gru_cell, input_data, initial_state=initial_state,

# initialize variables
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)



def show_hist_of_hidden_values(session, initial_state, state, title):
    """Set `initial_state` to different values and run the `state` value. Check diffe
       values due to different initializations. If the model cannot capture long term
       initialization does not have much effect to the value of `state` at a later ti
```

```
    """

    batch_size, hiddens_size = state.get_shape()

    # intialize the model with different initial states and then calculate the final s
    init_zero = np.zeros([batch_size, hidden_size])
    session.run(initial_state.assign(init_zero))
    state_zero_init = session.run(state)

    init_rand1 = np.random.rand(batch_size, hidden_size)
    session.run(initial_state.assign(init_rand1))
    state_rand1_init = session.run(state)

    init_rand2 = np.random.rand(batch_size, hidden_size)
    session.run(initial_state.assign(init_rand2))
    state_rand2_init = session.run(state)

    init_scaleup1 = init_rand1 * 100
    session.run(initial_state.assign(init_scaleup1))
    state_scaleup1_init = session.run(state)

    # plot the difference between the four difference settings

    # For each sequence, calculate the norm of the difference of the states from diff
    norm_diff1 = np.linalg.norm(state_zero_init - state_rand1_init, axis=1)
    norm_diff2 = np.linalg.norm(state_zero_init - state_rand2_init, axis=1)
    norm_diff3 = np.linalg.norm(state_rand1_init - state_rand2_init, axis=1)
    norm_diff4 = np.linalg.norm(state_scaleup1_init - state_zero_init, axis=1)
    norm_diff5 = np.linalg.norm(state_scaleup1_init - state_rand1_init, axis=1)
    norm_diff6 = np.linalg.norm(state_scaleup1_init - state_rand2_init, axis=1)

    # plot the histogram of norms of differences
    n_bins = 20
    fig, axs = plt.subplots(2, 3, sharey=True, tight_layout=True)
    plt.suptitle(title, fontsize=16)

    axs[0, 0].hist(norm_diff1, bins=n_bins)
    axs[0, 1].hist(norm_diff2, bins=n_bins)
    axs[0, 2].hist(norm_diff3, bins=n_bins)
    axs[1, 0].hist(norm_diff4, bins=n_bins)
    axs[1, 1].hist(norm_diff5, bins=n_bins)
    axs[1, 2].hist(norm_diff6, bins=n_bins)




# set values for the basic RNN model
```

```
# play with the scale, and see if you can find any value that achieves long-term memo
scale = 2
wt_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale
wt_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale
bias = (np.random.rand(hidden_size) - 0.5) * scale

set_rnn_params(rnn_cell, session, wt_h, wt_x, bias)

# get the 10th state and check its dependency on the initial state
rnn_state10 = tf.transpose(rnn_outputs, [1, 0, 2])[10]
show_hist_of_hidden_values(session, initial_state, rnn_state10,
                           'Histogram of differences of state 10 with different initia

# get the 20th state and check its dependency on the initial state
rnn_state20 = tf.transpose(rnn_outputs, [1, 0, 2])[20]
show_hist_of_hidden_values(session, initial_state, rnn_state20,
                           'Histogram of differences of state 20 with different initia

# get the 20th state and check its dependency on the initial state
rnn_state30 = tf.transpose(rnn_outputs, [1, 0, 2])[30]
show_hist_of_hidden_values(session, initial_state, rnn_state30,
                           'Histogram of differences of state 30 with different initia
```



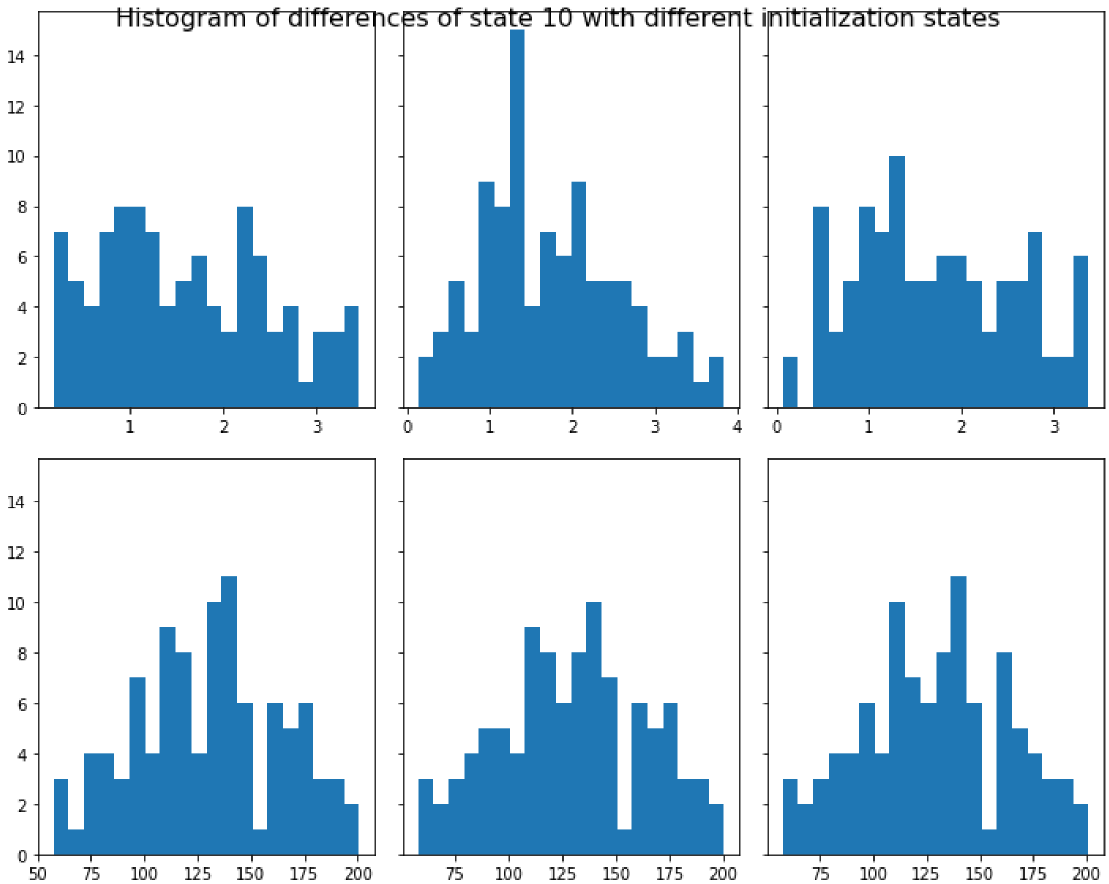

Histogram of differences of state 20 with different initializations
80
60
40
20
0
0.0
0.5
1.0
1.5
2.0
2.5



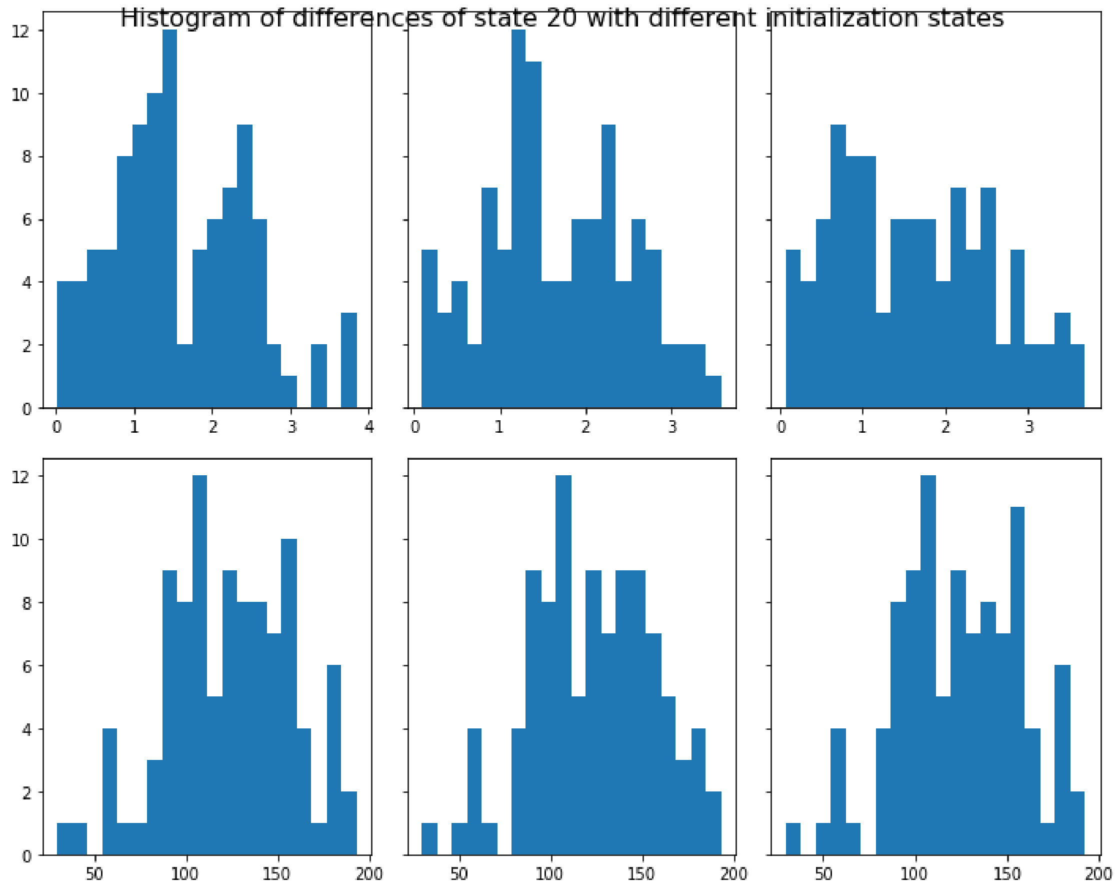
I couldn't find a parameter that kept long term dependency in the vanilla rnn

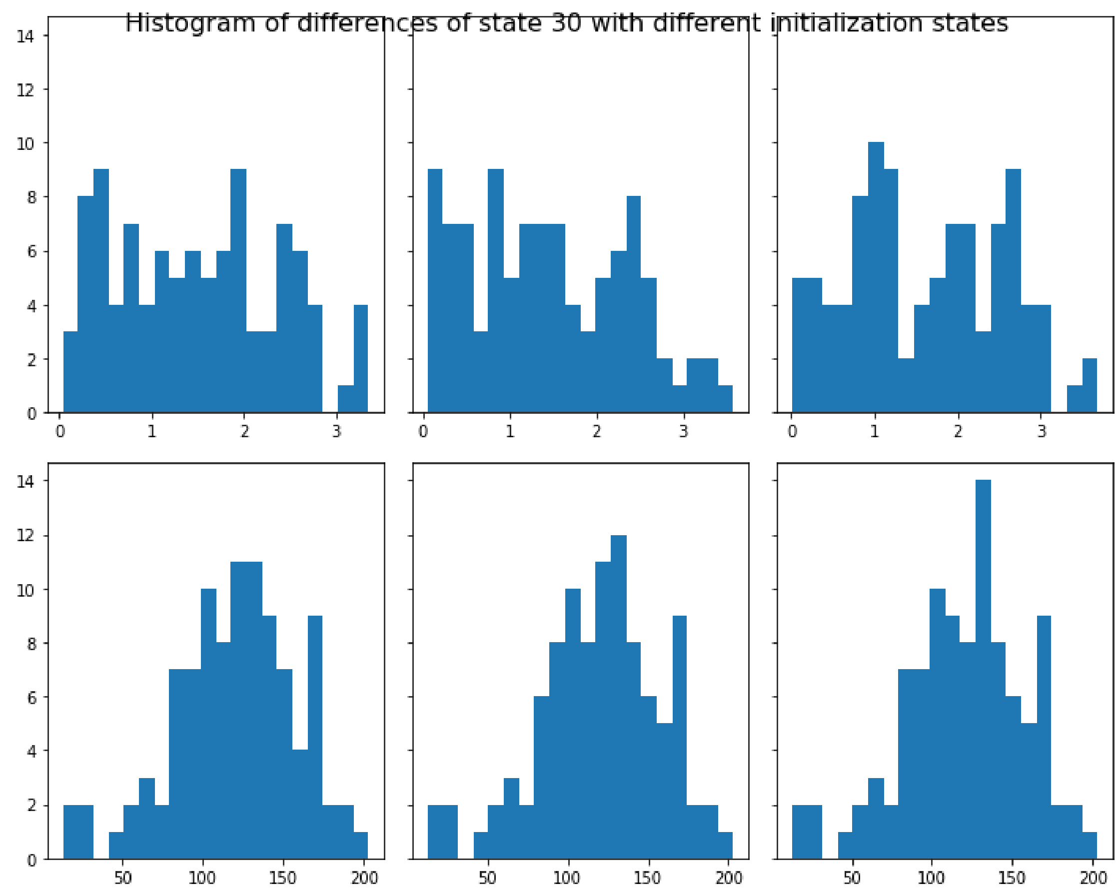
```
In [26]: # Can you set GRU parameters such that it maintains the initial state in the memory f

         scale_gru = 10
         wtu_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
         wtu_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
         biasu = (np.random.rand(hidden_size) - 0.5) * scale_gru
         wtr_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
         wtr_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
         biasr = (np.random.rand(hidden_size) - 0.5) * scale_gru
         wtc_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
         wtc_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
         biasc = (np.random.rand(hidden_size) - 0.5) * scale_gru

         set_gru_params(gru_cell, session,
                        wtu_h = wtu_h,
                        wtu_x = wtu_x,
                       biasu = biasu,
                       wtr_h = wtr_h,
```

```
                wtr_x = wtr_x,
                biasr = biasr,
                wtc_h = wtc_h,
                wtc_x = wtc_x,
                biasc = biasc)
#########################################################################
# 4. Setting GRU parameters (4 points)                                  #
# Set GRU parameters here so that it can capture long term dependency   #
#########################################################################

# get the 10th state
gru_state10 = tf.transpose(gru_outputs, [1, 0, 2])[10]
show_hist_of_hidden_values(session, initial_state, gru_state10,
                           'Histogram of differences of state 10 with different initi

# get the 20th state
gru_state20 = tf.transpose(gru_outputs, [1, 0, 2])[20]
show_hist_of_hidden_values(session, initial_state, gru_state20,
                           'Histogram of differences of state 20 with different initi

# get the 20th state
gru_state30 = tf.transpose(gru_outputs, [1, 0, 2])[30]
show_hist_of_hidden_values(session, initial_state, gru_state30,
                           'Histogram of differences of state 30 with different initi
```



Histogram of differences of state 10 with different initialization states


Histogram of differences of state 20 with different initialization states

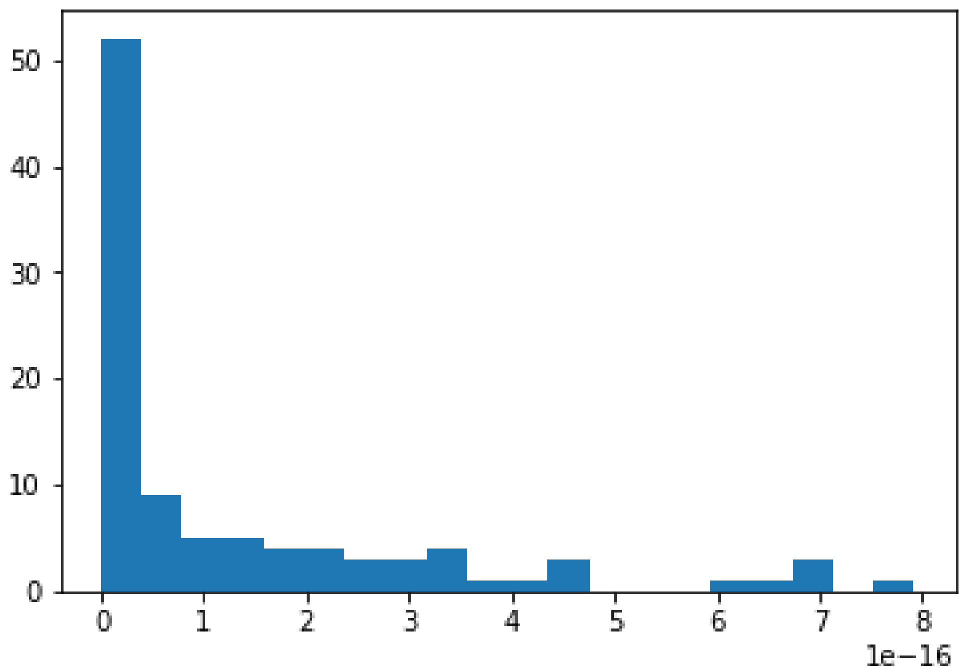


1.2.1 Backpropagation: vanishing gradients and exploding gradients

In the experiment, you will observe vanishing gradients and exploding gradients

In [10]:
```
# Calculate gradient with respect to the initial state

# the gradient with respect to state 30 is [1, 1, ..., 1]. Propagate the gradient bac
rnn_loss30 = tf.reduce_sum(rnn_state30)
rnn_gradh = tf.gradients([rnn_loss30], [initial_state])[0]

scale = .3
wt_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale
wt_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale
bias = (np.random.rand(hidden_size) - 0.5) * scale
set_rnn_params(rnn_cell, session, wt_h, wt_x, bias)


##################################################################################
# 5. Observe vanishing gradients (3 points)
```

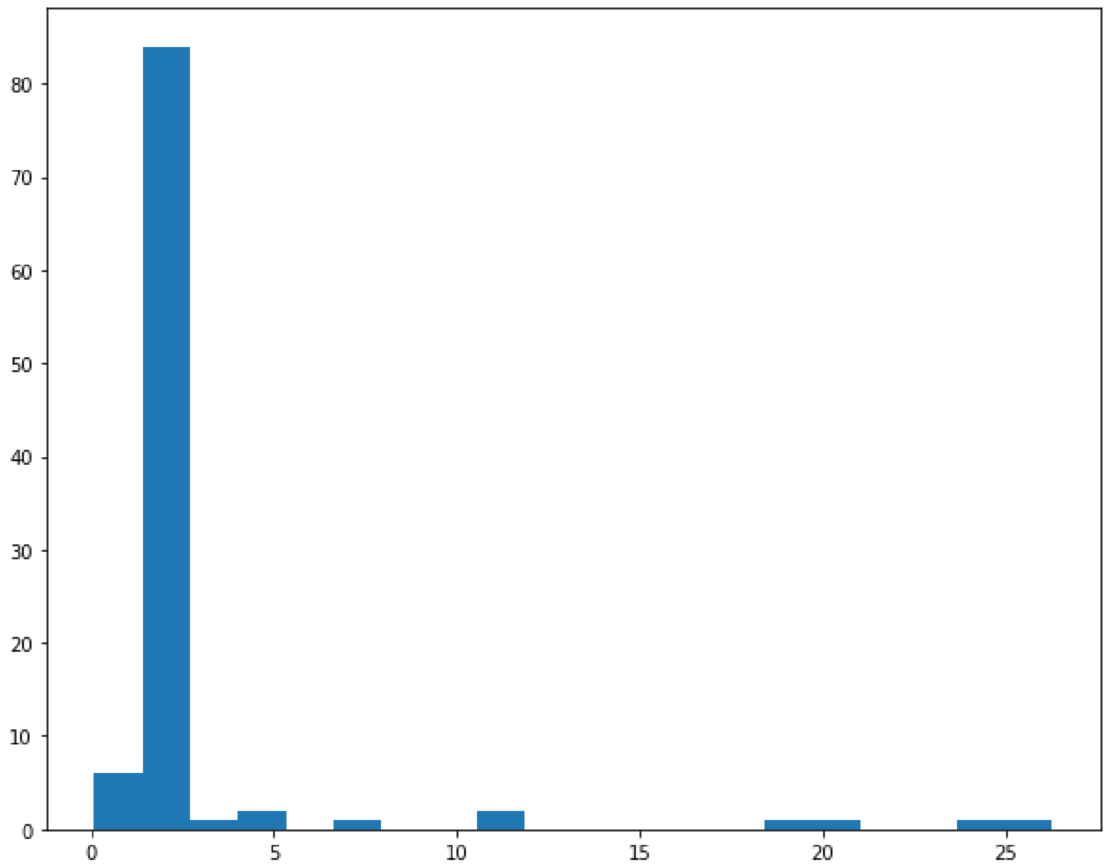
```
# Try a different settings of the parameters and see if you still get vanishing gradi
#####################################################################################


# show the norms of gradients. Most of them are zero.
np_rnn_gradh = session.run(rnn_gradh)
rnn_grad_norm = np.linalg.norm(np_rnn_gradh, axis=1)

n_bins = 20
_ = plt.hist(rnn_grad_norm, bins=n_bins)
```



I tried many different parameter settings and still got vanishing gradients for vanilla RNN

In [28]:
```
# Can you set GRU parameters such that the gradient does not vanish?

# the gradient with respect to state 30 is [1, 1, ..., 1]. Propagate the gradient bac
gru_loss30 = tf.reduce_sum(gru_state30)
gru_gradh = tf.gradients([gru_loss30], [initial_state])[0]

scale_gru = 5
wtu_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
wtu_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
biasu = (np.random.rand(hidden_size) - 0.5) * scale_gru
wtr_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
wtr_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
biasr = (np.random.rand(hidden_size) - 0.5) * scale_gru
```

```python
wtc_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale_gru
wtc_x = (np.random.rand(input_size, hidden_size) - 0.5) * scale_gru
biasc = (np.random.rand(hidden_size) - 0.5) * scale_gru

set_gru_params(gru_cell, session,
               wtu_h = wtu_h,
               wtu_x = wtu_x,
              biasu = biasu,
              wtr_h = wtr_h,
              wtr_x = wtr_x,
              biasr = biasr,
              wtc_h = wtc_h,
              wtc_x = wtc_x,
              biasc = biasc)

#################################################################################
# 6. GRU parameters that don't have vanishing gradients (3 points)              #
# Set GRU parameters so that the gradient of a later state with respect to the  #
# initial state is not near zero.                                               #
#################################################################################

# show norms of gradients
np_gru_gradh = session.run(gru_gradh)
gru_grad_norm = np.linalg.norm(np_gru_gradh, axis=1)

n_bins = 20
_ = plt.hist(gru_grad_norm, bins=n_bins)
```




```
In [ ]:
```

Task 2

April 4, 2019

```
In [114]: # As usual, a bit of setup
          import time
          import numpy as np
          import tensorflow as tf
          import matplotlib.pyplot as plt


          %matplotlib inline
          plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
          plt.rcParams['image.interpolation'] = 'nearest'
          plt.rcParams['image.cmap'] = 'gray'

          # for auto-reloading exroternal modules
          # see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
          %load_ext autoreload
          %autoreload 2

          from q3_RNNLM import *
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1 Running the Final Model

```
In [147]: test_RNNLM()
```

```
929589.0 total words with 10000 uniques
929589.0 total words with 10000 uniques
Epoch 0
Training perplexity: 467.6376037597656
Validation perplexity: 289.06573486328125
Total time: 17.174071311950684
Epoch 1
Training perplexity: 237.09164428710938
Validation perplexity: 212.89657592773438
Total time: 17.509174823760986
```

```
Epoch 2
Training perplexity: 182.98269653320312
Validation perplexity: 181.60520935058594
Total time: 16.92972421646118
Epoch 3
Training perplexity: 154.789794921875
Validation perplexity: 165.90191650390625
Total time: 16.649473905563354
Epoch 4
Training perplexity: 136.9542236328125
Validation perplexity: 155.64813232421875
Total time: 17.014497756958008
Epoch 5
Training perplexity: 124.54450988769531
Validation perplexity: 149.719970703125
Total time: 16.758183002471924
Epoch 6
Training perplexity: 115.26555633544922
Validation perplexity: 145.48056030273438
Total time: 16.539767265319824
Epoch 7
Training perplexity: 107.82228088378906
Validation perplexity: 141.95249938964844
Total time: 16.948673486709595
Epoch 8
Training perplexity: 101.83666229248047
Validation perplexity: 139.7904510498047
Total time: 17.073339700698853
Epoch 9
Training perplexity: 96.89061737060547
Validation perplexity: 138.23367309570312
Total time: 16.914764404296875
Epoch 10
Training perplexity: 92.6684341430664
Validation perplexity: 137.10202026367188
Total time: 17.11223530769348
Epoch 11
Training perplexity: 89.13345336914062
Validation perplexity: 136.18280029296875
Total time: 18.168486833572388
Epoch 12
Training perplexity: 86.03854370117188
Validation perplexity: 136.1054229736328
Total time: 17.760778427124023
Epoch 13
Training perplexity: 83.37432098388672
Validation perplexity: 135.13983154296875
Total time: 17.5199134349823
```

```
Epoch 14
Training perplexity: 81.03960418701172
Validation perplexity: 135.4987335205078
Total time: 17.121501684188843
Epoch 15
Training perplexity: 78.92498779296875
Validation perplexity: 135.07244873046875
Total time: 17.703442096710205
Epoch 16
Training perplexity: 77.07970428466797
Validation perplexity: 135.52450561523438
Total time: 16.21398687362671
Epoch 17
Training perplexity: 75.35388946533203
Validation perplexity: 135.6739959716797
Total time: 16.15951418876648
Epoch 18
Training perplexity: 73.81407165527344
Validation perplexity: 136.0308074951172
INFO:tensorflow:Restoring parameters from ptb_rnnlm.weights
=-==-==-==-==-== 126.20983886718751
Test perplexity: 126.08154296875
=-==-==-==-==-=


in boston which is give the new role in the letter of motorola 's system is <unk> to like <unk


they have had other clothes <eos>


please exist and there is n't appropriate for imports for commissions <eos>


today stem by continued from cold fusion anyone with start with <unk> tv positions <eos>


the president bush was having used the fat climate of gaf a staff cut in percentage of the <unl


in winter days he advises raises an <unk> plant on monday 's and said <eos>


i want to gift payments <eos>


look at risk with offsetting trades close to our <unk> new post as it is the <unk> because of l
```

```
come to settle a basket we 've think that we happen as many they ca n't lose the <unk> because

he said they 'd n't transferred to comprehensive care <eos>
```

In [ ]: