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 [1]: # 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))))
```

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.random.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-3-40b6cc66db36>: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 unders
        # 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_cell
```

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 1.3522993435792944e-07
Difference between your GRU implementation and tf GRU 1.7900158443936265e-06
```

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]:
```
from rnn_param_helper import set_gru_params
from scipy.special import expit as sigmoid

# Assign some value to a parameter of GRU

####################################################################################
# 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.                     #
####################################################################################

wtr_h = np.zeros((hidden_size, hidden_size), dtype=np.float32)
wtr_x = np.zeros((input_size, hidden_size), dtype=np.float32)
biasr = np.full((hidden_size,), np.inf, dtype=np.float32)

wtu_h = np.zeros((hidden_size, hidden_size), dtype=np.float32)
wtu_x = np.zeros((input_size, hidden_size), dtype=np.float32)
biasu = np.full((hidden_size,), -np.inf, dtype=np.float32)

wt_h, wt_x, bias = get_rnn_params(rnn_cell, session)

wtc_h = wt_h
wtc_x = wt_x
biasc = bias

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)

# 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_out
```

```
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 [7]:
```
from rnn_param_helper import set_rnn_params, set_gru_params
```

```python
# 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) - 0

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

# copy the basic RNN and the GRU RNN above here:
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(hidden_size)
gru_cell = tf.nn.rnn_cell.GRUCell(hidden_size)

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 th
# from the larger problem. Basically you just need to copy some code above here.
#########################################################################################

#rnn_outputs, _ = tf.nn.dynamic_rnn(rnn_cell, ...) # please complete this line and lin
#gru_outputs, _ = tf.nn.dynamic_rnn(gru_cell, ...) # please complete this line and lin

rnn_outputs, _ = tf.nn.dynamic_rnn(rnn_cell, input_data, initial_state=initial_state, d
gru_outputs, _ = 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)
```

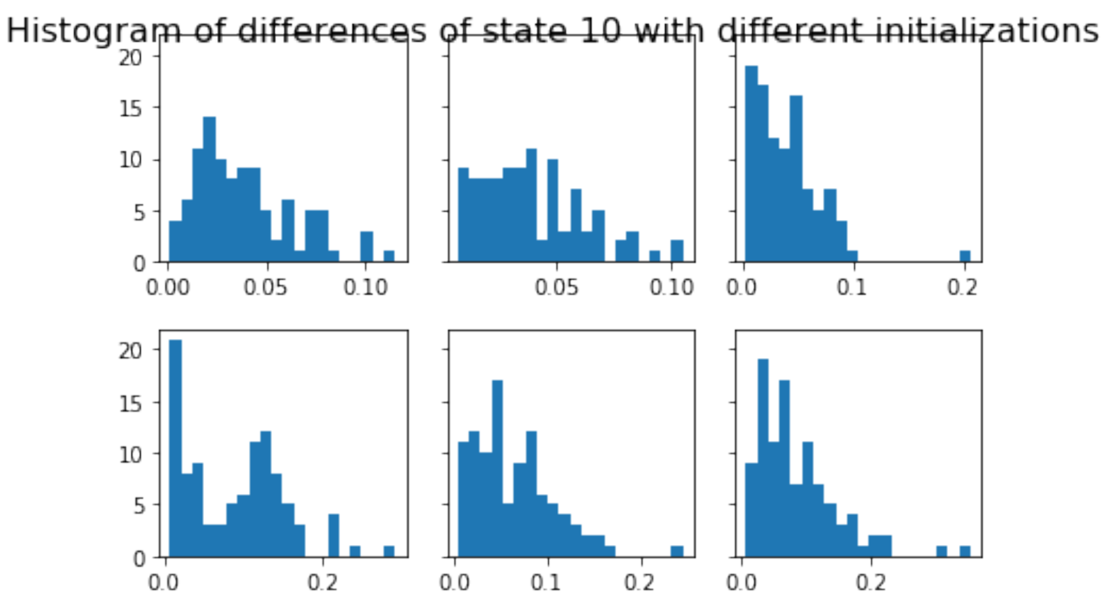
```python
def show_hist_of_hidden_values(session, initial_state, state, title):
    """Set `initial_state` to different values and run the `state` value. Check differe
       values due to different initializations. If the model cannot capture long term d
       initialization does not have much effect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

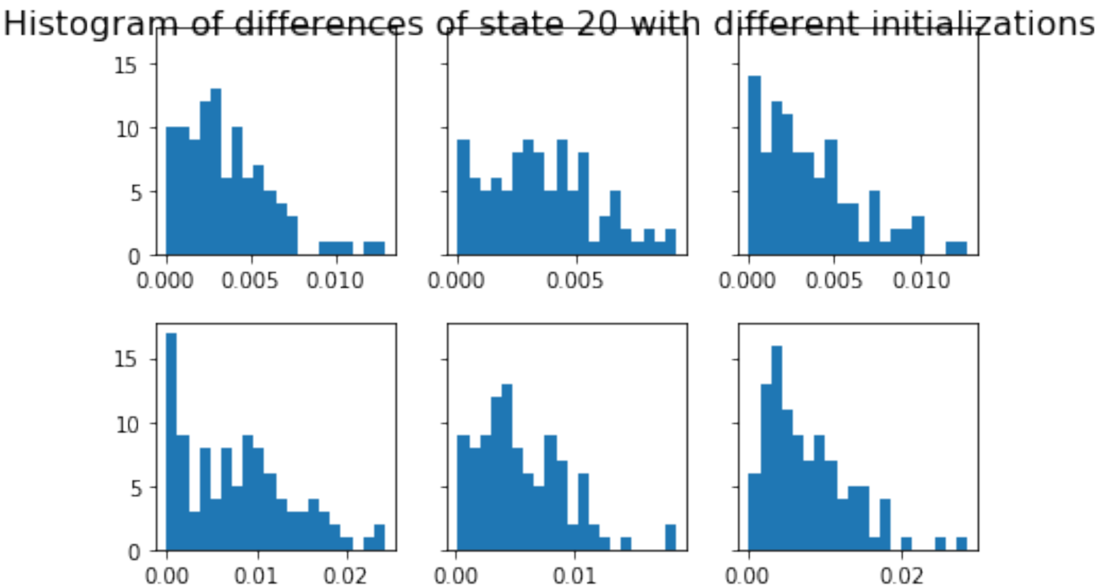
```
    batch_size, hiddens_size = state.get_shape()

    # 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 diffe
    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)
```

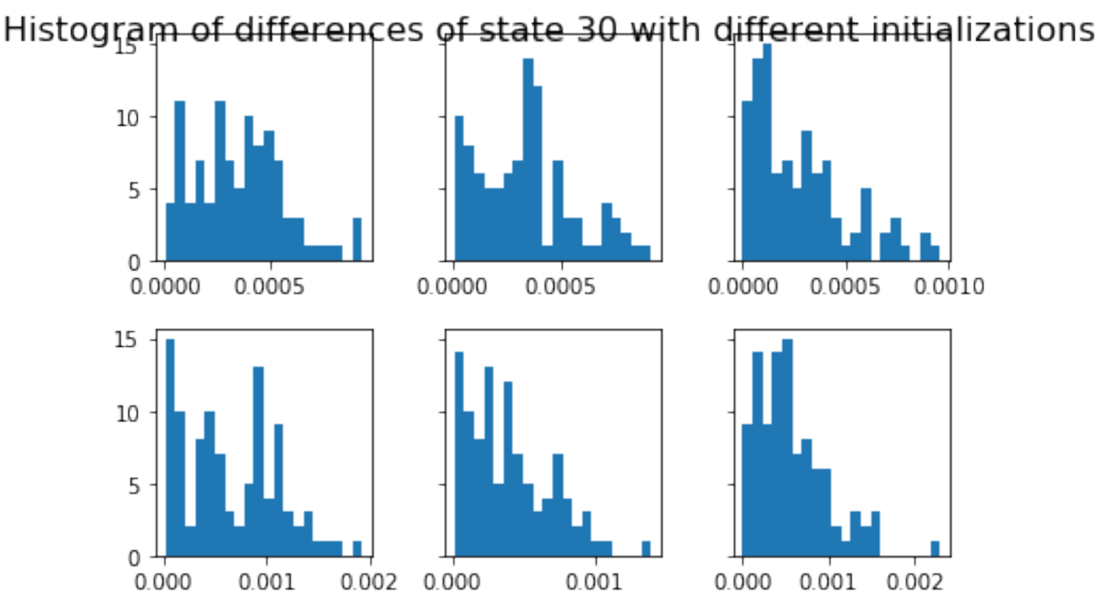
```
# set values for the basic RNN model

# play with the scale, and see if you can find any value that achieves long-term memor
```

```
    scale = np.sqrt(2/hidden_size)
    wt_h = (np.random.rand(hidden_size, hidden_size) - 0.5) * scale *2
    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
```

```
/anaconda3/envs/dnn/lib/python3.6/site-packages/matplotlib/figure.py:2366: UserWarning: This fi
  warnings.warn("This figure includes Axes that are not compatible "
```

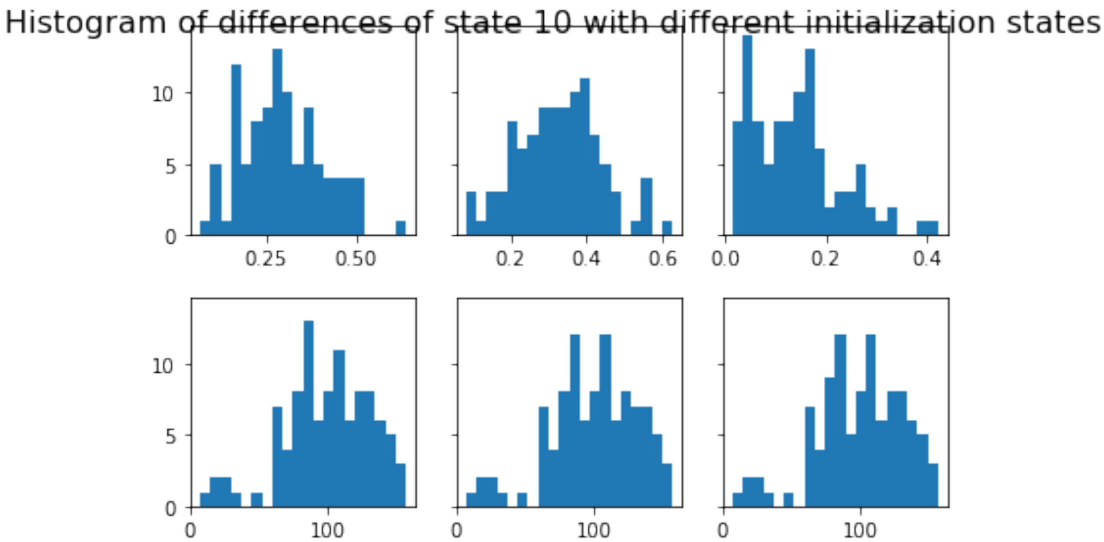


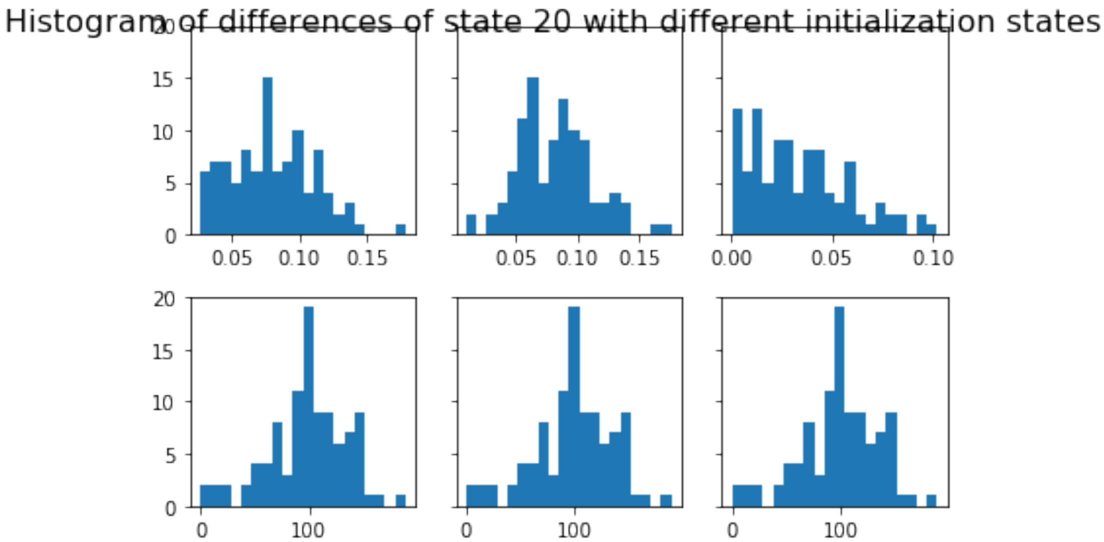


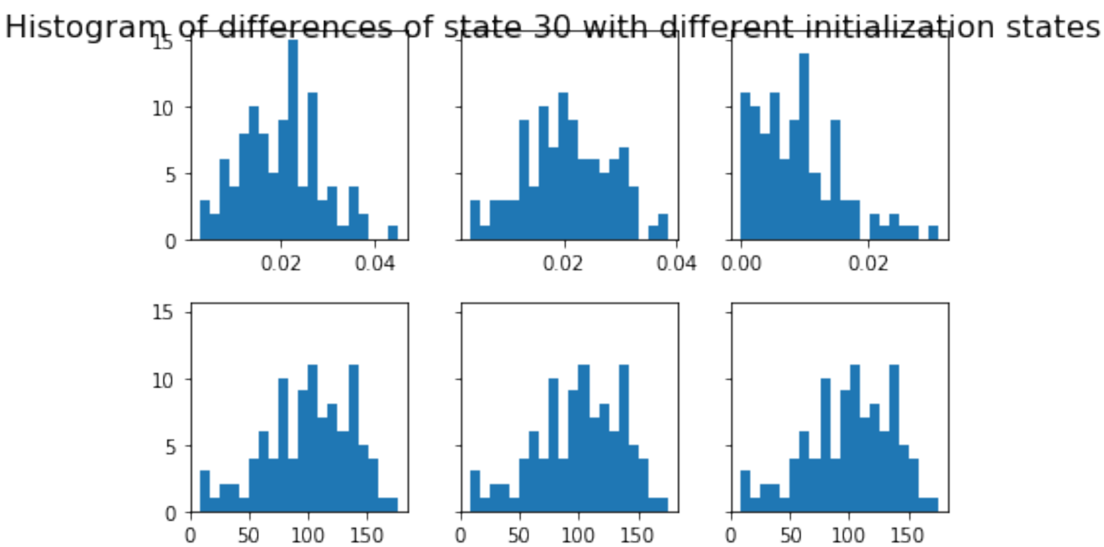


In [8]:
```
# Can you set GRU parameters such that it maintains the initial state in the memory fo

scale_gru = np.sqrt(2/hidden_size)
wtu_h, wtu_x, biasu, wtr_h, wtr_x, biasr, wtc_h, wtc_x, biasc = get_gru_params(gru_cel
wtu_h = wtu_h * scale_gru * 2
wtu_x = wtu_x * scale_gru
biasu = biasu * scale_gru
```

```
wtr_h = wtr_h * scale_gru * 2
wtr_x = wtr_x * scale_gru
biasr = biasr * scale_gru
wtc_h = wtc_h * scale_gru * 2
wtc_x = wtc_x * scale_gru
biasc = biasc * 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 initial

# 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 initial

# 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 initial
```





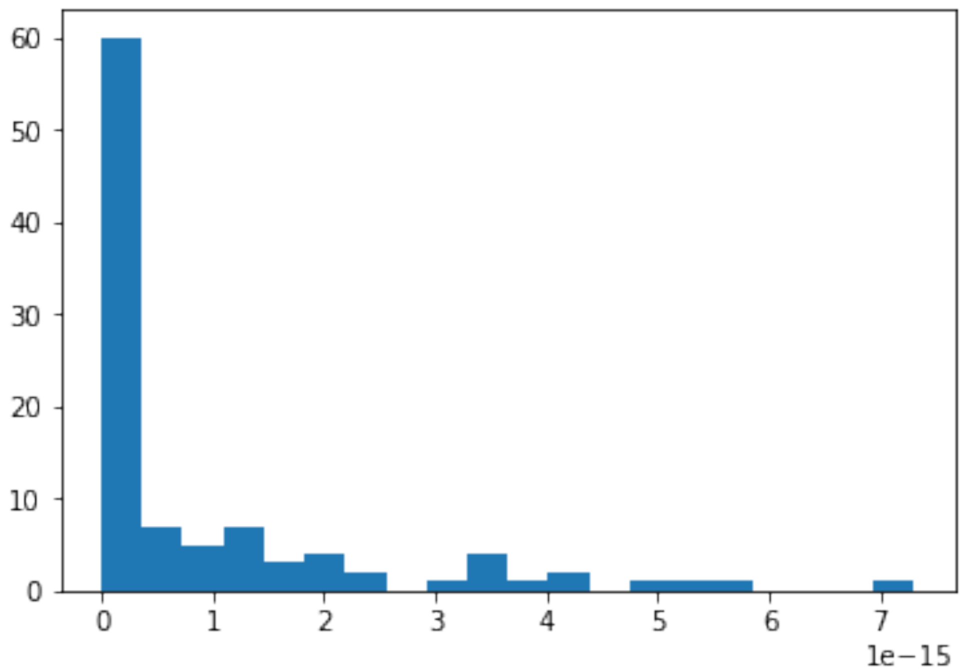


1.2.1 Backpropagation: vanishing gradients and exploding gradients

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

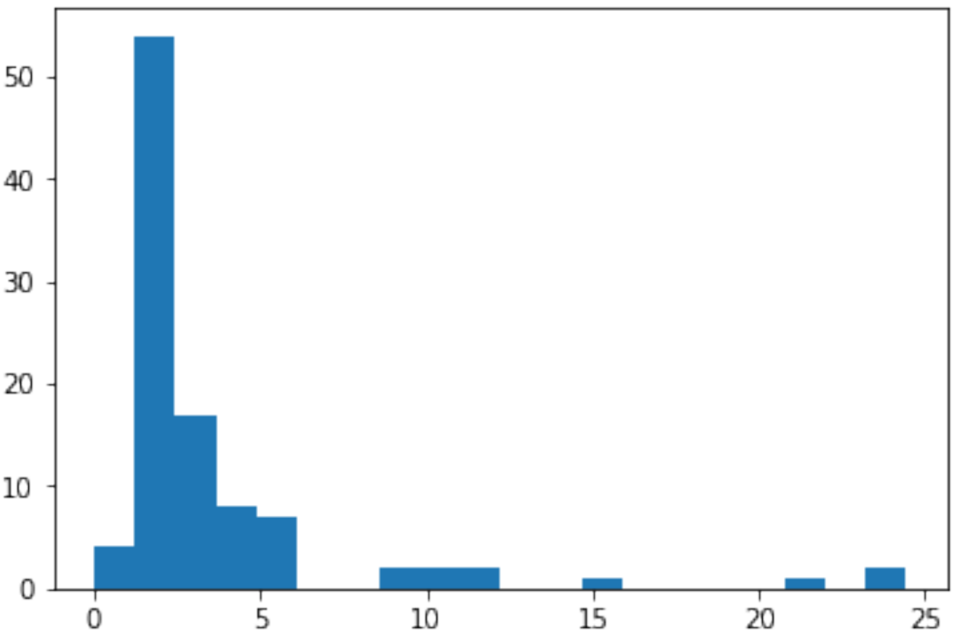
In [9]:
```
# Calculate gradient with respect to the initial state

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

```
scale = np.sqrt(2 / hidden_size)
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 gradie
####################################################################################


# 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)
```




```
In [10]: # Can you set GRU parameters such that the gradient does not vanish?

          wtu_h = wtu_h * scale_gru
          wtu_x = wtu_x * scale_gru
          biasu = biasu * scale_gru
```

```
wtr_h = wtr_h * scale_gru
wtr_x = wtr_x * scale_gru
biasr = biasr * scale_gru
wtc_h = wtc_h * scale_gru
wtc_x = wtc_x * scale_gru
biasc = biasc * 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)

# 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]

###################################################################################
# 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 [ ]:
```

```
Training perplexity: 122.0152816772461
Validation perplexity: 155.42393493652344
Total time: 13.16973066329956
Epoch 5
Training perplexity: 109.44181823730469
Validation perplexity: 149.56298828125
Total time: 13.221745252609253
Epoch 6
Training perplexity: 99.98884582519531
Validation perplexity: 145.85891723632812
Total time: 13.195596694946289
Epoch 7
Training perplexity: 92.5727310180664
Validation perplexity: 143.75823974609375
Total time: 13.353787183761597
Epoch 8
Training perplexity: 86.54630279541016
Validation perplexity: 142.7977294921875
Total time: 13.301990509033203
Epoch 9
Training perplexity: 81.5321273803711
Validation perplexity: 142.63426208496094
Total time: 13.250473976135254
Epoch 10
Training perplexity: 77.28886413574219
Validation perplexity: 143.09457397460938
Total time: 13.290369033813477
Epoch 11
Training perplexity: 73.64620208740234
Validation perplexity: 144.11981201171875
Total time: 13.211983442306519
Epoch 12
Training perplexity: 70.47470092773438
Validation perplexity: 145.56590270996094
=-==-==-==-==-=
Test perplexity: 133.71490478515625
=-==-==-==-==-=


in boston soup stern publicly pile of new york predict <eos>


they have extraordinary profit baseball payment at N other # N level throughout road between the meanwhile presented the <unk> news program <eos>


please recognize virginia with it suggests this quarter is what is adequate <unk> consideration of eating dioxide a weekly <eos>


today hangs two verdict aimed at providing mgm grand who would find close to the <unk> of a further supplier <eos>


the president chancellor of about its three lines stake tendered to a small company base <eos>

Process finished with exit code 0
```