

Recurrent Neural Network using TensorFlow

Youngjae Yu

May 18, 2017

<https://yj-yu.github.io/rnntf>

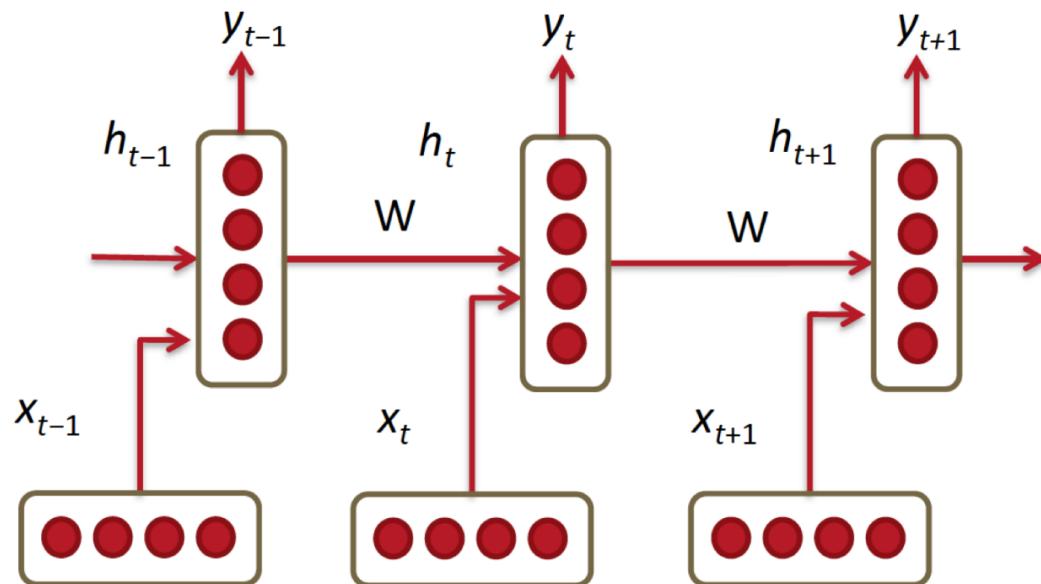


About

- Contents
 - RNN Basics
 - Tensorflow Basics
 - Tensorflow APIs for RNN
 - Practice - RNN
 - Practice - Monitoring
 - Advanced
 - Advanced Practice - char rnn

RNN Basics

Recurrent Neural Networks!

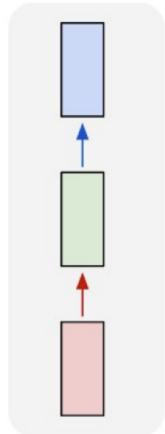


Richard Socher

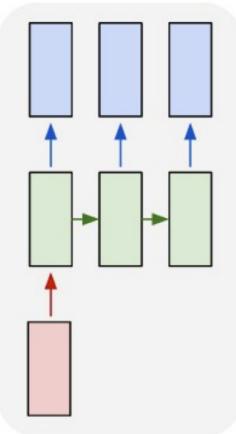
- RNNs tie the weights at **each time**
- Condition the neural network on **all previous words**
- RAM requirement only scales with number of words

RNN Basics

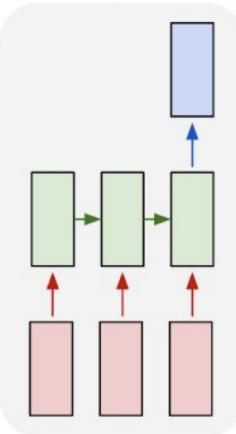
one to one



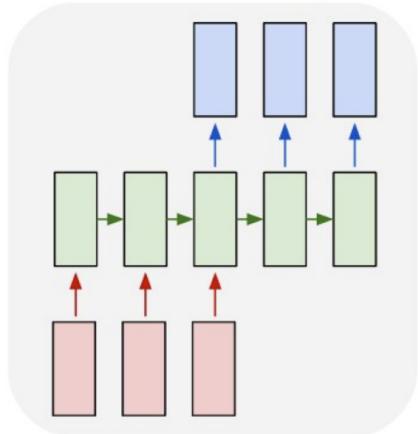
one to many



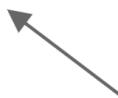
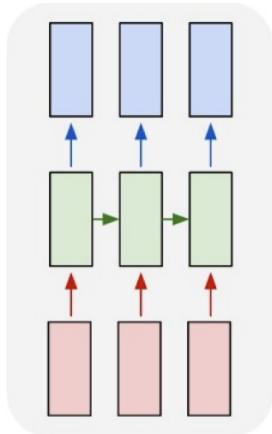
many to one



many to many



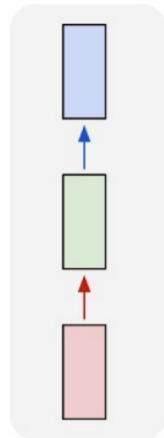
many to many



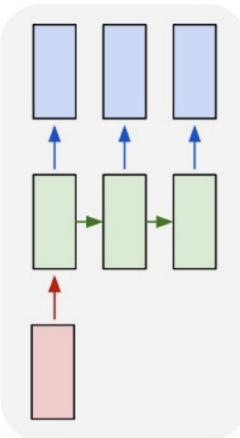
Vanilla Neural Networks

RNN Basics

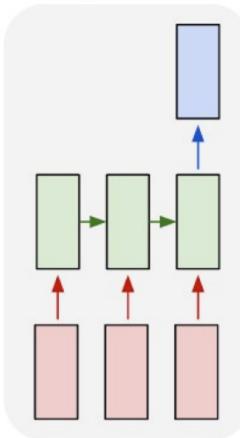
one to one



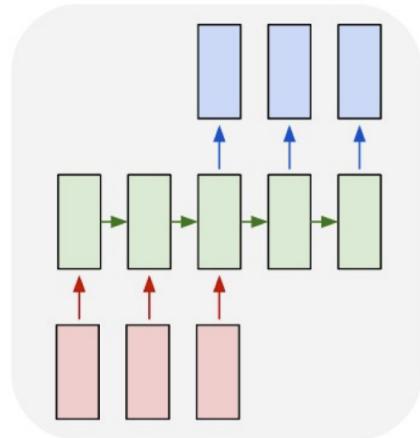
one to many



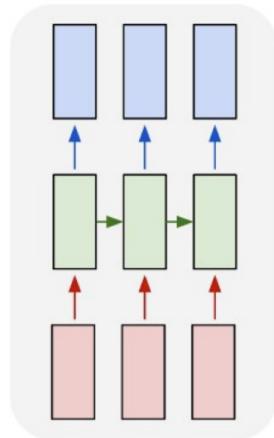
many to one



many to many



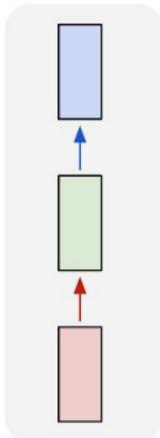
many to many



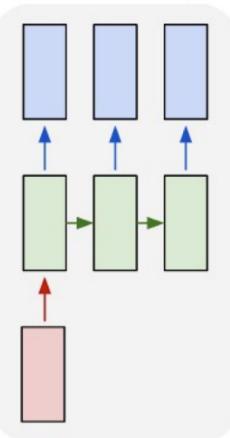
e.g. **Image Captioning**
image -> sequence of words

RNN Basics

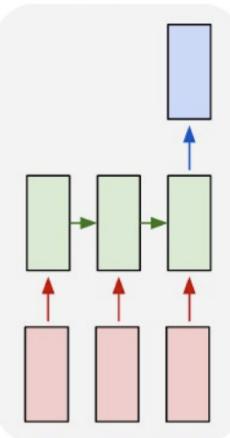
one to one



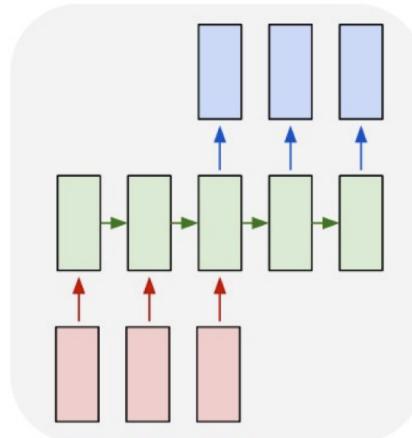
one to many



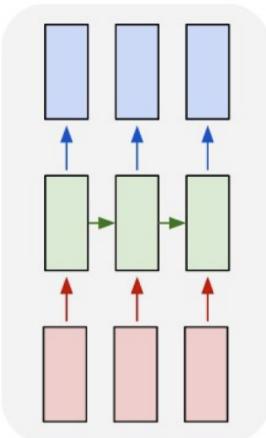
many to one



many to many



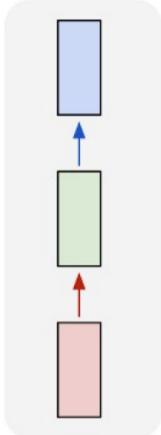
many to many



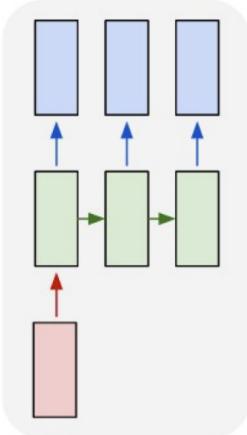
e.g. **Sentiment Classification**
sequence of words → sentiment

RNN Basics

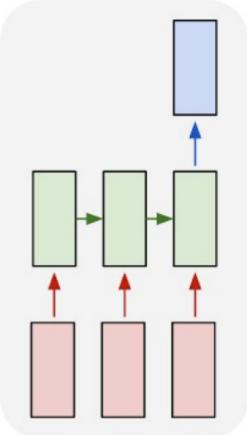
one to one



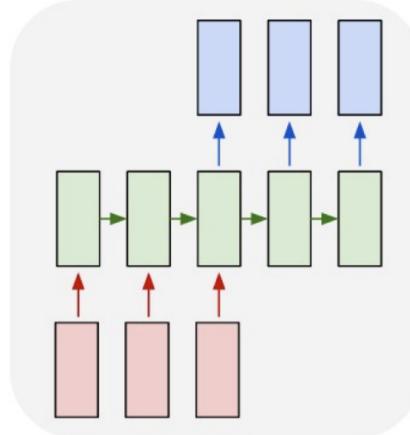
one to many



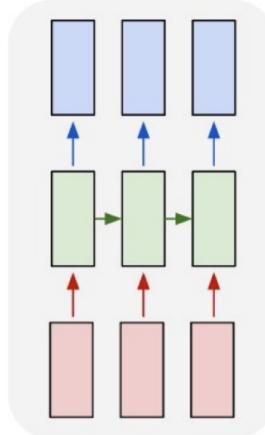
many to one



many to many



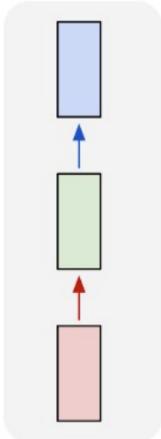
many to many



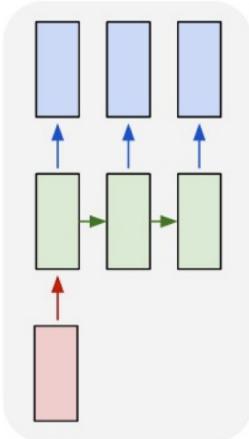
e.g. **Machine Translation**
seq of words -> seq of words

RNN Basics

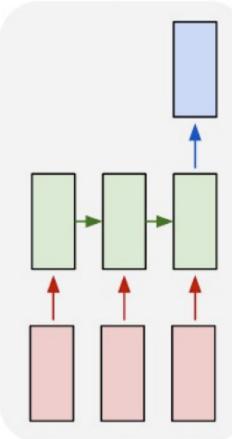
one to one



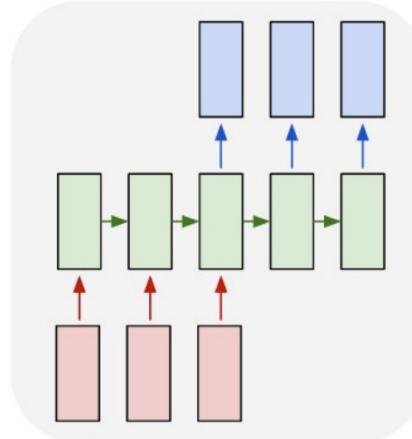
one to many



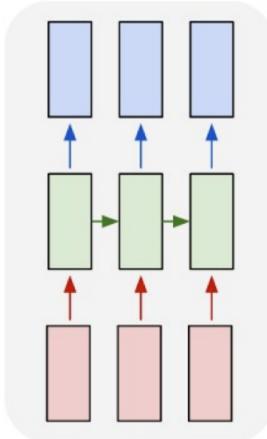
many to one



many to many

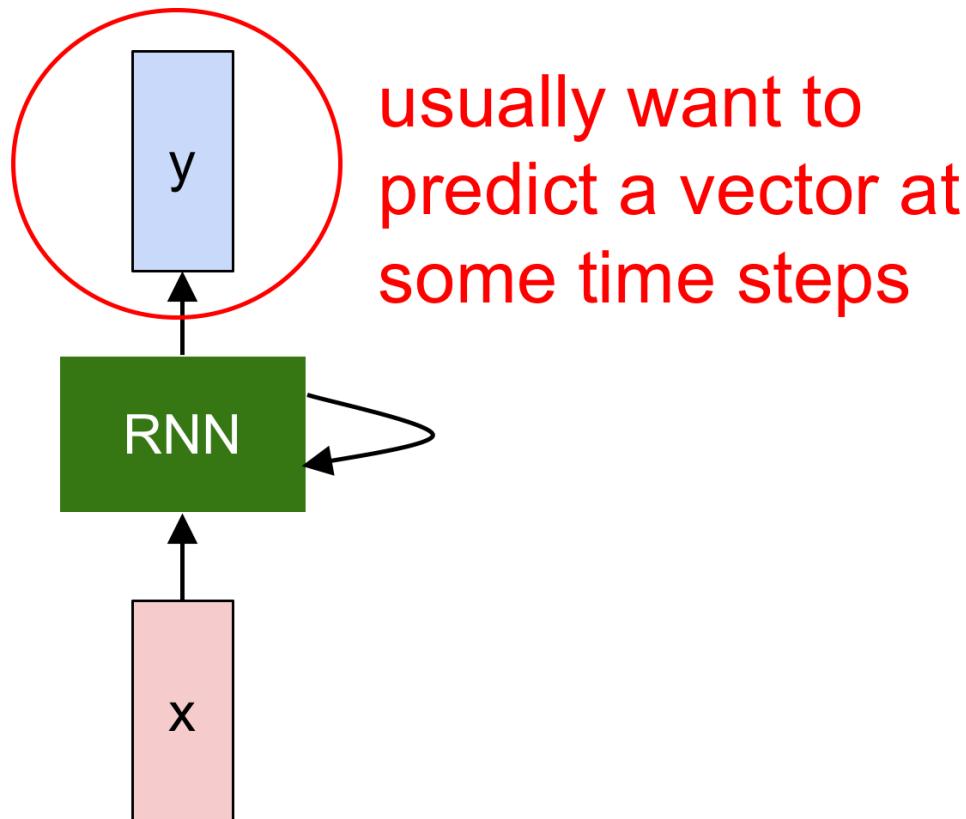


many to many



e.g. **Video classification on frame level**

RNN Basics

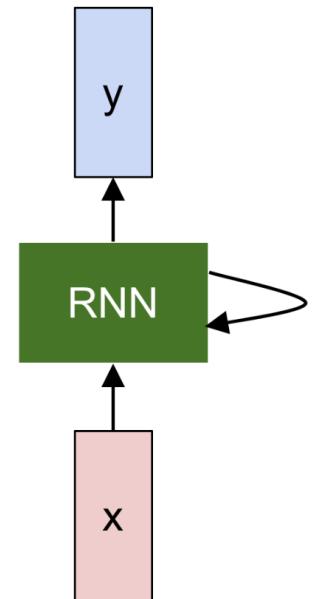


RNN Basics

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state / old state input vector at
 some function some time step
 with parameters W

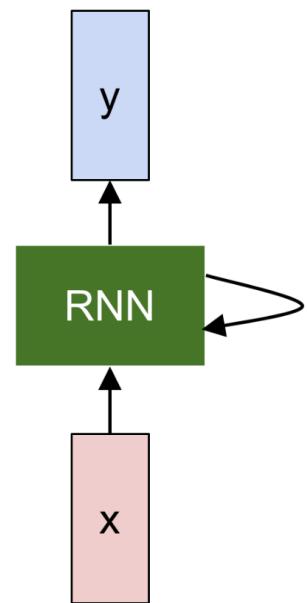


RNN Basics

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

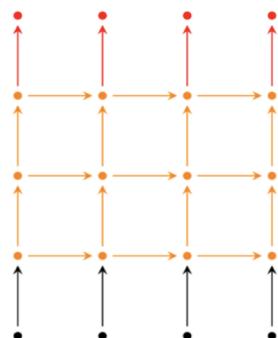
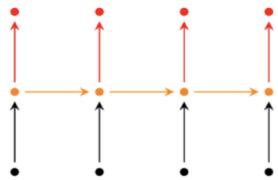
$$h_t = f_W(h_{t-1}, x_t)$$

Notice: the same function and the same set of parameters are used at every time step.



RNN Basics

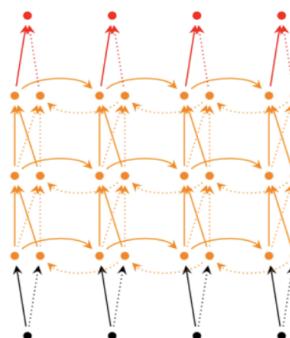
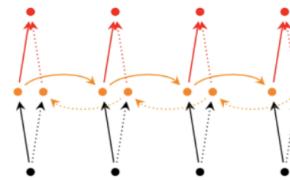
RNN



$$h_t = f(Wx_t + Vh_{t-1} + b) \quad (1)$$

$$y_t = g(Uh_t + c) \quad (2)$$

BRNN



$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b}) \quad (3)$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b}) \quad (4)$$

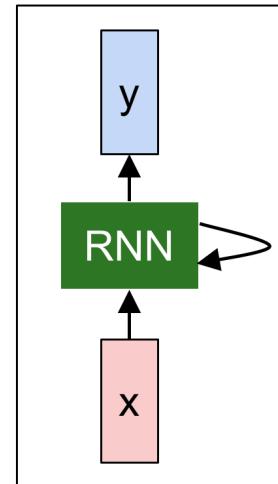
$$y_t = g(U_{\rightarrow}\vec{h}_t + U_{\leftarrow}\overleftarrow{h}_t + c) \quad (5)$$

RNN Basic

**Character-level
language model
example**

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

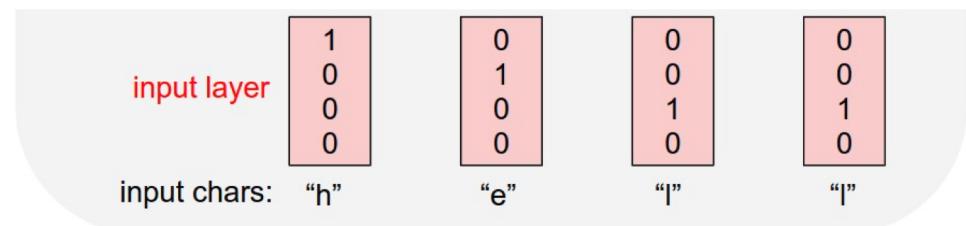


RNN Basic

**Character-level
language model
example**

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”



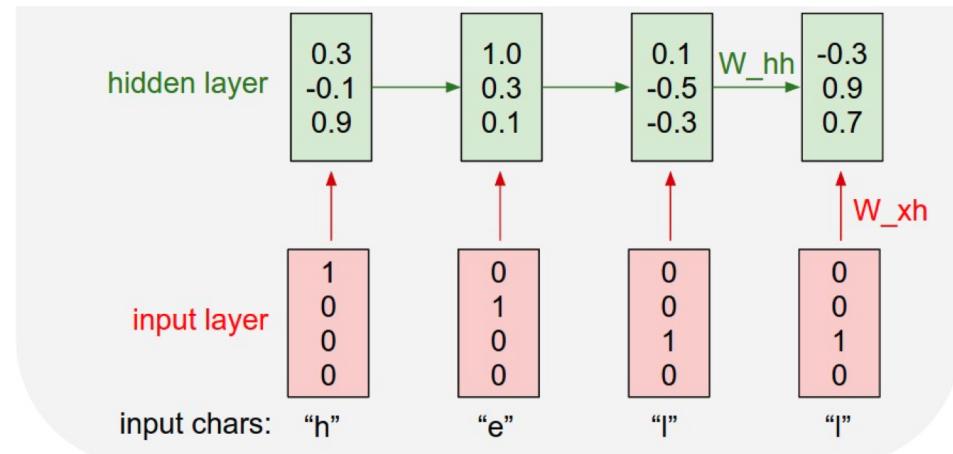
RNN Basic

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

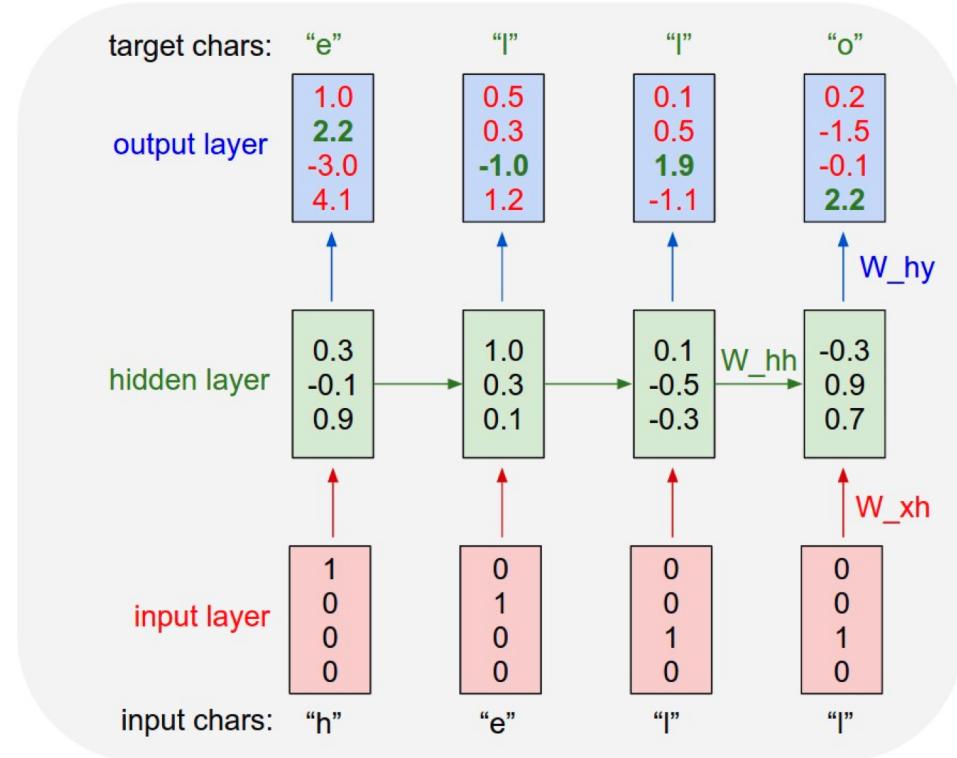


RNN Basic

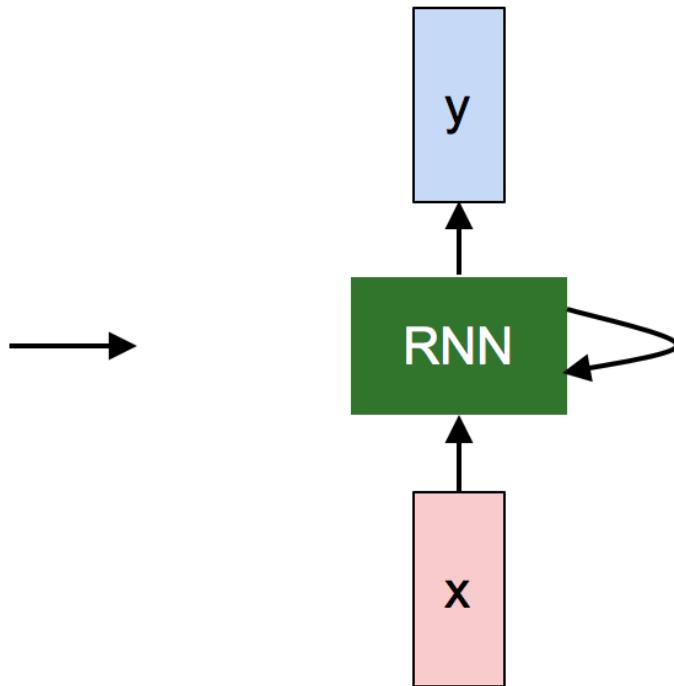
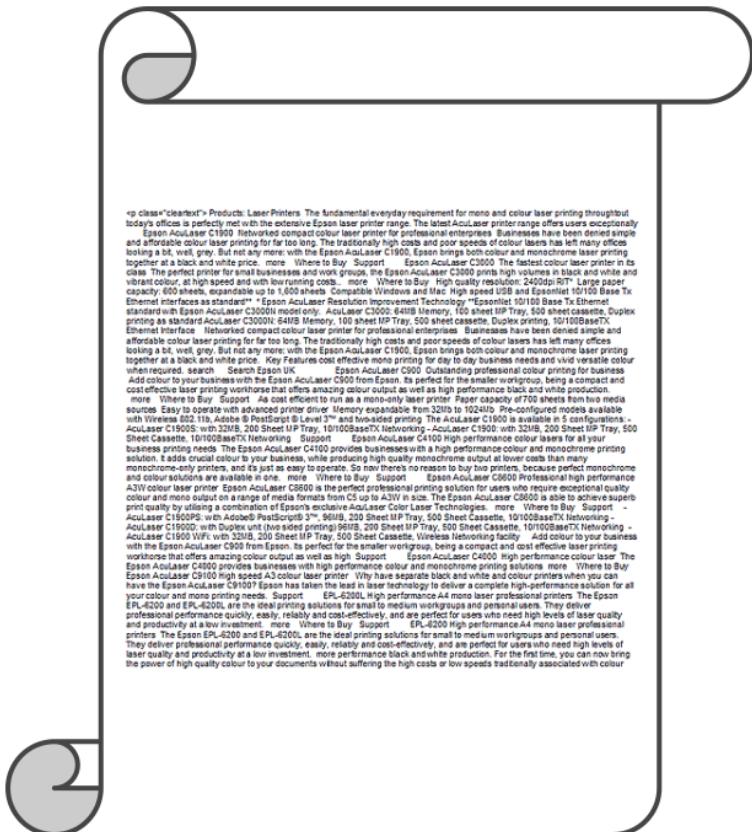
Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



RNN Basic



RNN Basic

at first:

tyntd-iafhatawiaoahrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tkldrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

↓ train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwyl fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

↓ train more

Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.

↓ train more

"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

RNN Python

<https://goo.gl/5npVuw>

RNN Python

[min-char-rnn.py](#) gist

```

# Maximal character-level Vanilla RNN Model. Written by Andrej Karpathy (karpathy)
# B00 License
#
# Import numpy as np

# # data I/O
# data = open('input.txt', 'r').read() # should be simple plain text file
# chars = list(data)
# vocab_size = len(chars)
# data_size = len(data)
# len(chars)

print("data has %d characters, %d unique." % (data_size, len(chars)))
print("data has %d characters, %d unique." % (data_size, vocab_size))

char_to_ix = {c: i for i, c in enumerate(chars)}
ix_to_char = {i: c for i, c in enumerate(chars)}
```



Data I/O

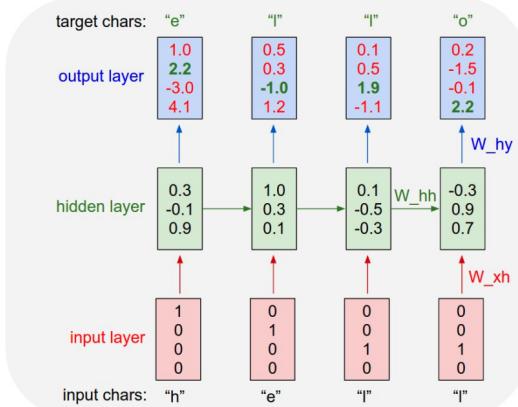
```
1 """
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be simple plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
12 char_to_ix = { ch:i for i,ch in enumerate(chars) }
13 ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

RNN Python

[min-char-rnn.py](#) gist

```
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
```

recall:



RNN Python

min-char-rnn.py gist

```
1 Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
2 BSD License
3
4 import numpy as np
5
6 # data I/O
7 data = open('input.txt', 'r').read() # should be simple plain text file
8 data_size, vocab_size = len(data), len(chars)
9 print 'data has %d characters, %d unique' % (data_size, vocab_size)
10 char_to_ix = {c:i for i,c in enumerate(chars)}
11 ix_to_char = {i:c for i,c in enumerate(chars)}
12
13 # hyperparameters
14 hidden_size = 100 # size of hidden layer of neurons
15 seq_length = 25 # number of steps in unroll the RNN for learning, rate = 1e-1
16
17 # model parameters
18 wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
19 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
20 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
21 bh = np.zeros((hidden_size,)) # hidden bias
22 by = np.zeros((vocab_size,)) # output bias
23
24 # forward pass
25 # inputs, targets are lists like of integers.
26 # inputs[i] is a list of integers.
27 # inputs[i] is not array of initial hidden state
28 # returns the loss, gradients on model parameters, and last hidden state
29 # integers
30 xs, hs, ys, ps = ([], [], [], [])
31
32 def lossFun(inputs, targets, hprev):
33     """ Compute total loss on a sentence by averaging per-step losses """
34
35     loss = 0.0
36     n, p = 0, 0
37
38     mWxh, mwLhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
39     mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
40
41     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
42
43     while True:
44
45         # prepare inputs (we're sweeping from left to right in steps seq_length long)
46         if p+seq_length+1 >= len(data) or n == 0:
47             hprev = np.zeros((hidden_size,1)) # reset RNN memory
48             p = 0 # go from start of data
49
50         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
51         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
52
53         # sample from the model now and then
54         if n % 100 == 0:
55             sample_ix = sample(hprev, inputs[0], 200)
56             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
57             print '----\n%s\n----' % (txt, )
58
59         # forward seq_length characters through the net and fetch gradient
60         loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
61         smooth_loss = smooth_loss * 0.999 + loss * 0.001
62
63         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
64
65         # perform parameter update with Adagrad
66         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
67                                       [dWxh, dWhh, dWhy, dbh, dby],
68                                       [mWxh, mwLhh, mWhy, mbh, mby]):
69             mem += dparam * dparam
70             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
71
72         p += seq_length # move data pointer
73         n += 1 # iteration counter
74
75     # perform parameter update with Adagrad
76     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
77                                   [dWxh, dWhh, dWhy, dbh, dby],
78                                   [mWxh, mwLhh, mWhy, mbh, mby]):
79         mem += dparam * dparam
80         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
81
82     # save parameters
83     np.save('RNN_params.npz', {'Wxh': Wxh, 'Whh': Whh, 'Why': Why,
84                               'bh': bh, 'by': by})
85
86     print 'done'
```

Main loop



```
81     n, p = 0, 0
82     mWxh, mwLhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83     mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85
86     while True:
87
88         # prepare inputs (we're sweeping from left to right in steps seq_length long)
89         if p+seq_length+1 >= len(data) or n == 0:
90             hprev = np.zeros((hidden_size,1)) # reset RNN memory
91             p = 0 # go from start of data
92
93         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
94         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
95
96
97         # sample from the model now and then
98         if n % 100 == 0:
99             sample_ix = sample(hprev, inputs[0], 200)
100            txt = ''.join(ix_to_char[ix] for ix in sample_ix)
101            print '----\n%s\n----' % (txt, )
102
103         # forward seq_length characters through the net and fetch gradient
104         loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
105         smooth_loss = smooth_loss * 0.999 + loss * 0.001
106
107         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
108
109         # perform parameter update with Adagrad
110         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
111                                       [dWxh, dWhh, dWhy, dbh, dby],
112                                       [mWxh, mwLhh, mWhy, mbh, mby]):
113             mem += dparam * dparam
114             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
115
116         p += seq_length # move data pointer
117         n += 1 # iteration counter
118
119
120     # perform parameter update with Adagrad
121     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
122                                   [dWxh, dWhh, dWhy, dbh, dby],
123                                   [mWxh, mwLhh, mWhy, mbh, mby]):
124         mem += dparam * dparam
125         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
126
127     # save parameters
128     np.save('RNN_params.npz', {'Wxh': Wxh, 'Whh': Whh, 'Why': Why,
129                               'bh': bh, 'by': by})
130
131     print 'done'
```

RNN Python

min-char-rnn.py gist

```
1  #include character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
2  #
3  # BSD License
4  #
5  import numpy as np
6
7  # data I/O
8  #
9  # Input file (e.g. text) + 'r' read) + should be simple plain text file
10 chars = list(open(data))
11 data_size, vocab_size = len(chars), len(chars)
12 hidden_size = 256
13 seq_length = 25 # number of steps to unroll the RNN for
14 learning_rate = 1e-1
15
16 # parameters
17 # hidden_size = size of hidden layer of neurons
18 # seq_length = number of steps to unroll the RNN for
19 # learning_rate = 1e-1
20
21 # weights
22 # w_in = random(ranchedim(hidden_size, vocab_size)*0.01 + input to hidden
23 wh = np.random.rand(hidden_size, hidden_size)*0.01 + hidden to hidden
24 bh = np.zeros(hidden_size)*0.01 + hidden to output
25 by = np.zeros(vocab_size, 1) + output bias
26
27 def lossFun(inputs, targets, hprev):
28
29     inputs, targets = both lists of integers.
30     hprev is last array of initial hidden state
31     parameters (w_in, wh, bh, by) are on model parameters, and last hidden state
32     are
33     i.e., h, y, a = (D, D, D)
34     h[1:] = 0 (copy(hprev)
35     loss = 0
36
37     for t in range(len(inputs)):
38
39         w_in = np.random.rand(vocab_size, 1) # encode in 1-of-K representation
40         w_in[inputs[t], 1] = 1
41
42         h[1:] = np.tanh(np.dot(w_in, a[t-1]) + np.dot(wh, h[t-1]) + b[t-1]) # hidden state
43         p[t] = np.exp(a[t]/np.sum(np.exp(a[t]))) # probabilities for next chars
44
45         loss += -np.log(p[targets[t]]) # softmax (cross-entropy loss)
46
47         da = np.zeros_like(a[t])
48         da[targets[t]] = 1
49
50         dwh, dbh, dby = np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(by)
51         dhnext = np.zeros_like(h[0])
52
53         for l in reversed(range(t+1, len(inputs))):
54
55             da = np.tanh(np.dot(dwh, da) + dbh + dby)
56             dh = np.tanh(da) * 1 - h[l-1]**2 # backprop into h
57             da = np.dot(wh.T, dh) # dh = backprop through tanh nonlinearity
58             dwh += np.outer(da, a[l-1])
59             dbh += np.outer(da, 1)
60             dby += np.outer(da, 1)
61
62             dhnext = np.dot(wh, dhnext) + dh
63
64         for l in range(1, t+1):
65
66             da = np.tanh(np.dot(dwh, da) + dbh + dby)
67             dh = np.tanh(da) * 1 - h[l-1]**2 # backprop into h
68             da = np.dot(wh.T, dh) # dh = backprop through tanh nonlinearity
69             dwh += np.outer(da, a[l-1])
70             dbh += np.outer(da, 1)
71             dby += np.outer(da, 1)
72
73             dhnext = np.dot(wh, dhnext) + dh
74
75         np.clip(da, -5, 5, da) # clip to mitigate exploding gradients
76         vgrad = np.concatenate([da, dbh, dby, dwh, dhnext])
77
78         def sgmooth(x):
79
80             sample = a sequence of integers from the model
81             h is memory state, seed_ix is seed letter for first time step
82             x = np.zeros((vocab_size, 1))
83             a = np.zeros((hidden_size, 1))
84             h = np.zeros((hidden_size, 1))
85             i = 0
86
87             for t in range(seq_length):
88
89                 if t == 0:
90                     h = np.tanh(np.dot(w_in, x) + np.dot(wh, h) + b)
91                     y = np.exp(a)/np.sum(np.exp(a))
92                     i = np.argmax(y)
93                     x = np.zeros_like(x)
94                     x[i, 1] = 1
95
96                 else:
97                     h = np.tanh(np.dot(dwh, da) + dbh + dby)
98                     da = np.tanh(da) * 1 - h[t-1]**2
99                     dh = np.tanh(da) * 1 - h[t-1]**2
100                    da = np.dot(wh.T, dh)
101                    dwh += np.outer(da, a[t-1])
102                    dbh += np.outer(da, 1)
103                    dby += np.outer(da, 1)
104
105             return a
106
107
108         h, p = sgmooth(w_in, wh, bh, by, vgrad, seed_ix)
109
110         sample_ix = np.argmax(h)
111         txt += ix_to_char[sample_ix]
112
113         # forward one step through the net and fetch gradient
114         loss, dwh, dbh, dby, dh, da, hprev = lossFun(inputs, targets, hprev)
115         smooth_loss = smooth_loss * 0.999 + loss * 0.001
116
117         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
118
119         # perform parameter update with Adagrad
120         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
121                                     [dWxh, dWhh, dWhy, dbh, dby],
122                                     [mWxh, mWhh, mWhy, mbh, mbby]):
123
124             mem += dparam * dparam
125             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
126
127             p += seq_length # move data pointer
128             n += 1 # iteration counter
```

Main loop

```
81     n, p = 0, 0
82     mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83     mbh, mbby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85
86     while True:
87
88         # prepare inputs (we're sweeping from left to right in steps seq_length long)
89         if p+seq_length+1 >= len(data) or n == 0:
90             hprev = np.zeros((hidden_size,1)) # reset RNN memory
91             p = 0 # go from start of data
92             inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
93             targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
94
95
96         # sample from the model now and then
97         if n % 100 == 0:
98             sample_ix = sample(hprev, inputs[0], 200)
99             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
100            print '----\n%s\n----' % (txt, )
101
102
103         # forward seq_length characters through the net and fetch gradient
104         loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
105         smooth_loss = smooth_loss * 0.999 + loss * 0.001
106
107         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
108
109
110         # perform parameter update with Adagrad
111         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
112                                     [dWxh, dWhh, dWhy, dbh, dby],
113                                     [mWxh, mWhh, mWhy, mbh, mbby]):
114
115             mem += dparam * dparam
116             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
117
118             p += seq_length # move data pointer
119             n += 1 # iteration counter
```



RNN Python

[min-char-rnn.py gist](#)

```
BSD License
Copyright (c) 2012 Andrej Karpathy (@karpathy)
All rights reserved.

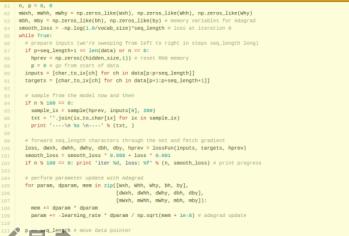
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
data_size, vocab_size = len(data), len(chars)
char_to_ix = {ch:i for i,ch in enumerate(chars)}
ix_to_char = {i:ch for ch in enumerate(chars)}

# hyperparameters
hidden_size = 200 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1
databeta = 0.9

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*np.sqrt(0.01)
Whh = np.random.randn(hidden_size, hidden_size)*np.sqrt(0.01)
Why = np.random.randn(vocab_size, hidden_size)*0.01
bh = np.zeros(hidden_size)
by = np.zeros(vocab_size, 1) # output bias

def lossFun(inputs, targets, hprev):
    """ Compute loss and gradients over a sequence of inputs and targets.
    inputs: targets are both list of integers.
    hprev: is initial hidden state
    returns the loss, gradients on model parameters, and last hidden state
    """
    n, p, m = 0, 0, 0
    x, h, y, dx, dh, db, dy = ([], [], [], [], [], [], [])
    for t in range(len(inputs)):
        x.append(inputs[t])
        y.append(targets[t])
        if t >= 1:
            h = np.copy(hprev)
        if t >= 1:
            x.append(np.zeros((vocab_size, 1))) # encode in 1-of-K representation
        x_t = np.zeros((vocab_size, 1))
        x_t[int(inputs[t])] = 1.0
        h_t = h + np.tanh(np.dot(Wxh, x_t) + np.dot(Whh, h[t-1]) + bh)
        y_t = np.dot(Why, h_t) + by # unnormalized log probabilities for next chars
        y_t -= np.logsumexp(y_t) # softmax (cross-entropy loss)
        loss += -np.log(y_t[targets[t], 0]) # softmax (cross-entropy loss)
        dx_t = np.zeros_like(x_t)
        dh_t = np.zeros_like(h)
        db_t = np.zeros_like(by)
        dy_t = np.zeros_like(y_t)
        dwhy, dbh = np.zeros_like(Why), np.zeros_like(bh), np.zeros_like(by)
        dwhh, dbh = np.zeros_like(Whh), np.zeros_like(bh)
        dwxh, dbh = np.zeros_like(Wxh), np.zeros_like(bh)
        for b in range(len(inputs)-t-1):
            dx[b] = np.copy(dx_t)
            dh[b] = np.copy(dh_t)
            db[b] = np.copy(db_t)
            dy[b] = np.copy(dy_t)
            dwhy += np.outer(dx[b], dy[b]) # backprop into Why
            dbh += np.outer(dx[b], db[b]) # backprop into bh
            dwhh += np.outer(dh[b], dh[b]) # backprop through tanh nonlinearity
            dbh += np.outer(dh[b], db[b])
            dwhh += np.outer(db[b], db[b])
            dwxh += np.outer(dx[b], dh[b])
            dbh += np.outer(db[b], dh[b])
        for d in [dwhy, dbh, dwhh, dwh, dwh, dwh, dbh, db, dy]:
            d *= databeta
        dh[-1] = np.tanh(h[-1]) * dh[-1] # clip to mitigate exploding gradients
    return loss, dx, dh, db, dy, dh[-1], x[-1]
    
```



Main loop

```
81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n%s\n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100 loss, dWxh, dWhh, dWhy, dbh, dyb, hprev = lossFun(inputs, targets, hprev)
101 smooth_loss = smooth_loss * 0.999 + loss * 0.001
102 if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104 # perform parameter update with Adagrad
105 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                               [dWxh, dWhh, dWhy, dbh, dyb],
107                               [mWxh, mWhh, mWhy, mbh, mby]):
108     mem += dparam * dparam
109     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
    
```

RNN Python

[min-char-rnn.py](#) gist

Main loop

```

1 n, p = 0, 0
2 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
3 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
4 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
5 while True:
6     # prepare inputs (we're sweeping from left to right in steps seq_length long)
7     if p+seq_length+1 >= len(data) or n == 0:
8         hprev = np.zeros((hidden_size,1)) # reset RNN memory
9         p = 0 # go from start of data
10    inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
11    targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
12
13    # sample from the model now and then
14    if n % 100 == 0:
15        sample_ix = sample(hprev, inputs[0], 200)
16        txt = ''.join(ix_to_char[ix] for ix in sample_ix)
17        print '----\n% %\n----' % (txt, )
18
19    # forward seq_length characters through the net and fetch gradient
20    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
21    smooth_loss = smooth_loss * 0.999 + loss * 0.001
22    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
23
24    # perform parameter update with Adagrad
25    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
26                                  [dWxh, dWhh, dWhy, dbh, dby],
27                                  [mWxh, mWhh, mWhy, mbh, mby]):
28        mem += dparam * dparam
29        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
30
31    p += seq_length # move data pointer
32    n += 1 # iteration counter

```

RNN Python

[min-char-rnn.py gist](#)

```

1 # Minimal character-level Vanilla Rnn model. Written by Andre Karpathy (karpathy)
2 #
3 # BSD License
4 #
5 # Import numpy as np
6 #
7 # data I/O
8 # data = open('input.txt', 'r').read() # should be simple plain text file
9 # data_size, vocab_size = len(data), len(chars)
10 # hidden_size = 200 # size of hidden layer of neurons
11 # seq_length = 20 # number of steps to unroll the RNN for
12 # learning_rate = 1e-1
13 #
14 # model parameters
15 # wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
16 # bh = np.zeros((hidden_size, 1)) # hidden bias
17 # why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
18 # by = np.zeros((vocab_size, 1)) # output bias
19 #
20 def lossFun(inputs, targets, hprev):
21     """ inputs,targets are both list of integers.
22     hprev is the array of initial hidden state
23     returns the loss, gradients on model parameters, and last hidden state
24     """
25     xs, hs, ys, ps = [], [], [], []
26     loss = 0
27     n = len(inputs)
28     hprev = np.copy(hprev)
29     for t in range(n):
30         x = inputs[t]
31         if t > 0: hprev = np.copy(hprev)
32         xst = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
33         xst[int(x)] = 1.0
34         hprev = np.dot(wh, hprev) + np.dot(bh, hst[t-1]) + xst
35         yst = np.dot(why, hprev) / np.sum(yst) # softmax
36         ps.append(yst)
37         loss += -np.log(yst[targets[t]])
38         y = np.argmax(yst)
39         hs.append(hprev)
40         ys.append(y)
41         hst = np.zeros((hidden_size, 1))
42         hst[int(y)] = 1.0
43         dhy = np.dot(why.T, dy) # dhnext = backprop through softmax nonlinearity
44         dhy -= np.sum(dhy) # unnormalized log probabilities for next chars
45         dhy *= -np.log(yst[1])
46         dhy /= vocab_size # normalize by number of remaining chars after softmax
47         dhy = np.dot(wh.T, dhy) # dhprev = backprop through tanh nonlinearity
48         dhy -= np.sum(dhy) # unnormalized log probabilities for previous chars
49         dhy *= -np.log(yst[0])
50         dhy /= vocab_size # normalize by number of remaining chars after softmax
51         dhy = np.zeros((hidden_size, 1))
52         dhy[0] = 1.0
53         dhy = np.dot(why, dhy) # dhnext = backprop through softmax nonlinearity
54         dhy -= np.sum(dhy) # unnormalized log probabilities for next chars
55         dhy *= -np.log(yst[1])
56         dhy /= vocab_size # normalize by number of remaining chars after softmax
57         dhy = np.zeros((hidden_size, 1))
58         dhy[0] = 1.0
59         dhy = np.dot(wh, dhy) + np.dot(bh, dhy) + np.sum(dhy)
60         dhy = np.zeros((hidden_size, 1))
61     return loss, xs, hs, ys, ps
62     del sample_ix, seed_ix, n
63
64 sample = lambda: np.array([sample(0, 26) for _ in range(seq_length)])
65 s = sample()
66 print(s)
67 print("Initial hidden state: ", hprev)
68 print("Initial hidden state: ", hprev)
69 print("Initial hidden state: ", hprev)
70 print("Initial hidden state: ", hprev)
71 print("Initial hidden state: ", hprev)
72 print("Initial hidden state: ", hprev)
73 print("Initial hidden state: ", hprev)
74 p = np.exp(-lossFun(s, s, hprev))
75 p = p / np.sum(p) # softmax
76 x = np.argmax(p)
77 x = np.zeros((vocab_size, 1))
78 x[int(x)] = 1.0
79 ix.append(x)
80 return ix
81
82 n, p = 0, 0
83 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
84 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
85 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
86
87 while True:
88     # prepare inputs (we're sweeping from left to right in steps seq_length long)
89     if p+seq_length+1 >= len(data) or n == 0:
90         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
91         p = 0 # go from start of data
92     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
93     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
94
95     # sample from the model now and then
96     if n % 100 == 0:
97         sample_ix = sample(hprev, inputs[0], 200)
98         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
99         print '----\n%s\n----' % (txt, )
100
101     # forward seq_length characters through the net and fetch gradient
102     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
103     smooth_loss = smooth_loss * 0.999 + loss * 0.001
104     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
105
106     # perform parameter update with Adagrad
107     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
108                                   [dWxh, dWhh, dWhy, dbh, dby],
109                                   [mWxh, mWhh, mWhy, mbh, mby]):
110         mem += dparam * dparam
111         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
112
113     p += seq_length # move data pointer
114     n += 1 # iteration counter
115
116 # forward pass through the net and fetch gradient
117 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
118                               [dWxh, dWhh, dWhy, dbh, dby],
119                               [mWxh, mWhh, mWhy, mbh, mby]):
120     mem += dparam * dparam
121     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
122
123 p += seq_length # move data pointer
124 n += 1 # iteration counter
125

```

Main loop



RNN Python

[min-char-rnn.py.gist](#)

```
Minibatch character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)

BSD License

1 import numpy as np
2
3 # input I/O
4
5 data = open('input.txt', 'r').read() # should be simple plain text file
6 data_size, vocab_size = len(data), len(chars)
7 print "data has %d characters, %d unique." % (data_size, vocab_size)
8 chars = sorted(list(set(chars)))
9 ix_to_char = {i:ch for i,ch in enumerate(chars)}
10
11 # hyperparameters
12 hidden_size = 200 # size of hidden layer of neurons
13 n_time_steps = 100 # number of steps to unroll the RNN for
14 learning_rate = 1e-1
15
16 # model parameters
17 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
18 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
19 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
20 bh = np.zeros((hidden_size, 1)) # hidden bias
21 by = np.zeros((vocab_size, 1)) # output bias
22
23 # forward pass: compute hidden states going forward
24 def forward(inputs, targets, hprev):
25     """ inputs, targets are both list of integers,
26     hprev is init array of initial hidden state
27     returns the loss, gradients on model parameters, and last hidden state
28     """
29     xs, hs, ys, ps = {}, {}, {}, {}
30     hs[-1] = np.copy(hprev)
31     loss = 0
32     for t in xrange(len(inputs)):
33         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
34         x[inputs[t]] = 1
35         hprev = np.tanh(np.dot(Wxh, x) + np.dot(Whh, hs[t-1]) + bh) # hidden state
36         ypred = np.dot(Why, hprev) # by + unnormalized log probabilities for next chars
37         ps[t] = np.exp(ypred) / np.sum(np.exp(ypred)) # softmax (cross-entropy loss)
38         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
39         y = np.zeros_like(by)
40         y[targets[t]] = 1
41         dy = np.zeros_like(ypred)
42         dy[targets[t]] = 1
43         dh = np.dot(Why.T, dy) # backprop through output nonlinearity
44         dbh = np.zeros_like(bh)
45         dby = np.zeros_like(by)
46         dWxh = np.zeros_like(Wxh)
47         dWhh = np.zeros_like(Whh)
48         for dim in [dh, dbh, dy, dby]:
49             np.add.at(dim, (t-1) * np.ones_like(dim), dim) # accumulate gradients
50     return loss, hs[-1], dx, dy
51
52 # softmax loss function, returns log probability and gradient for next char
53 def smooth_loss(logprobs, targets):
54     """ logprobs[i]: prob of char i for each step """
55     logprob = logprobs[np.arange(len(targets)), targets]
56     logprob -= np.max(logprob) # normalize for numerical stability
57     logprob = np.log(logprob)
58     logprob -= np.log(np.sum(np.exp(logprob)))
59     logprob *= -1
60     return logprob
61
62 # sample from the model, we use topk sampling
63 def sample(hprev, seed_ix, n):
64     """ hprev is current hidden state, seed_ix is previous sampled char, n is size of vocabulary """
65     x = np.zeros((vocab_size, 1))
66     x[seed_ix] = 1
67     for t in range(n):
68         hprev = np.tanh(np.dot(Wxh, x) + np.dot(Whh, hprev) + bh)
69         ypred = np.dot(Why, hprev) # by + unnormalized log probabilities for next chars
70         ypred -= np.max(ypred) # normalize for numerical stability
71         ypred = np.exp(ypred) / np.sum(np.exp(ypred))
72         ix = np.argmax(ypred)
73         x = np.zeros_like(x)
74         x[ix] = 1
75     return ix
76
77 # forward pass: sample on every step
78 def sampleRNN(inputs, targets, hprev, start=0, n_time_steps=100, topk=100):
79     """ inputs, targets, hprev, start, n_time_steps, topk """
80     if start < 0 or start > len(inputs)-1:
81         raise ValueError("start must be between 0 and len(inputs)-1")
82     if n_time_steps < 0:
83         raise ValueError("n_time_steps must be non-negative")
84     if topk < 0:
85         raise ValueError("topk must be non-negative")
86
87     # forward n_time_steps characters through the net and fetch gradients
88     loss, dx, dWxh, dWhh, dWhy, dbh, dby, hs, ys, ps = forward(inputs, targets, hprev)
89     for t in range(start+1, start+n_time_steps):
90         x = np.zeros((vocab_size, 1))
91         x[inputs[t]] = 1
92         hprev = np.tanh(np.dot(Wxh, x) + np.dot(Whh, hs[t-1]) + bh)
93         ypred = np.dot(Why, hprev) # by + unnormalized log probabilities for next chars
94         ypred -= np.max(ypred) # normalize for numerical stability
95         ypred = np.exp(ypred) / np.sum(np.exp(ypred))
96         ix = np.argmax(ypred)
97         x = np.zeros_like(x)
98         x[ix] = 1
99     return loss, dx, dWxh, dWhh, dWhy, dbh, dby, hs, ys, ps
100
101 # backward pass: compute gradients going backwards
102 def backward(inputs, targets, hprev, xs, hs, ys, ps, dy):
103     """ inputs, targets are both list of integers,
104     hprev is Hx1 array of initial hidden state
105     xs, hs, ys, ps = computed forward pass
106     dy is Hx1 array of gradients to backprop into y
107     """
108     for t in reversed(xrange(len(inputs))):
109         dy = np.zeros_like(dy)
110         dy[targets[t]] = 1
111         dy = np.dot(Why.T, dy) # backprop into y
112         dWxh += np.dot(xs[t].T, dy)
113         dWhh += np.dot(hs[t-1].T, dy)
114         dWhy += np.zeros_like(Why)
115         dbh += dy[0]
116         dby += dy[0]
117         dh = np.dot(Why.T, dy) + dhnext # backprop into h
118         ddraw = (1 - hs[t] * hs[t].T) * dh # backprop through tanh nonlinearity
119         dWxh += np.dot(xs[t].T, ddraw)
120         dWhh += np.dot(hs[t-1].T, ddraw)
121         dhnext = np.dot(Whh.T, ddraw)
122
123     return dWxh, dWhh, dWhy, dbh, dby, hs, ys, ps
124
125 # clip gradients to mitigate exploding gradients
126 np.clip(dWxh, -5, 5, out=dWxh)
127 np.clip(dWhh, -5, 5, out=dWhh)
128 np.clip(dWhy, -5, 5, out=dWhy)
129 np.clip(dbh, -5, 5, out=dbh)
130 np.clip(dby, -5, 5, out=dby)
```

Loss function

- forward pass (compute loss)
- backward pass (compute param gradient)

```
def lossFun(inputs, targets, hprev):
    """
    inputs,targets are both list of integers.
    hprev is Hx1 array of initial hidden state
    returns the loss, gradients on model parameters, and last hidden state
    """
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(hprev)
    loss = 0
    # forward pass
    for t in xrange(len(inputs)):
        xs[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
        ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
        ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
        loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
    # backward pass: compute gradients going backwards
    for t in reversed(xrange(len(inputs))):
        dy = np.zeros_like(dy)
        dy[targets[t]] = 1
        dy = np.dot(Why.T, dy) # backprop into y
        dWxh += np.dot(xs[t].T, dy)
        dWhh += np.dot(hs[t-1].T, dy)
        dWhy += np.zeros_like(Why)
        dbh += dy[0]
        dby += dy[0]
        dh = np.dot(Why.T, dy) + dhnext # backprop into h
        ddraw = (1 - hs[t] * hs[t].T) * dh # backprop through tanh nonlinearity
        dWxh += np.dot(xs[t].T, ddraw)
        dWhh += np.dot(hs[t-1].T, ddraw)
        dhnext = np.dot(Whh.T, ddraw)
    for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
        np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
    return loss, dWxh, dWhh, dWhy, dbh, dby, hs[-1]
```

RNN Python

min-char-rnn.py gist

```
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```

$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ y_t &= W_{hy}h_t \end{aligned}$$

Softmax classifier

111 q_length = move data pointer
111 + iteration counter

RNN Python

min-char-rnn.py gist

```
1  Minimal character-level vanilla RNN model, written by Andrej Karpathy (@karpathy)
2  BSD License
3
4  import numpy as np
5
6  # data I/O
7  # Given a file (e.g., test.txt), read() should be similar plain text file
8  chars = list(open(data))
9  data_size, vocab_size = len(chars), len(chars)
10 char_to_ix = {ch:i for i, ch in enumerate(chars)}
11 ix_to_char = {i:ch for i, ch in enumerate(chars)}
12
13 # Hyperparameters
14 hidden_size = 100 # size of hidden layer of neurons
15 seq_length = 25 # number of steps to unroll the RNN for
16 learning_rate = 3e-1
17
18 # Parameters
19 wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
20 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
21 bhh = np.zeros(hidden_size) # bias for hidden state
22 by = np.zeros(vocab_size, 0.01) # output bias
23
24 def loss(inputs, targets, hprev):
25     inputs, targets = both lists of integers.
26     hprev is array of initial hidden state
27     inputs[0] is seed letter, rest are on model parameters, and last hidden state
28     e.g. inputs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
29     h, y, x = 0, 0, 0
30     hprev = np.concatenate([h, 0])
31     loss = 0
32
33     for t in xrange(len(inputs)):
34         wih_t = np.zeros((vocab_size, 1)) # encode in 1-of-K representation
35         wih_t[inputs[t], 0] = 1
36         h[1:] = np.tanh(np.dot(wih_t, x) + np.dot(whh, h[1:]) + bh)
37         pih_t = np.exp(h[1]) / np.sum(np.exp(h[1])) # probabilities for next chars
38         pi_t = np.exp(targets[t] / np.sum(np.exp(y))) # probabilities for next chars
39         loss += -np.log(pi_t[range(targets[t])]) # softmax (cross-entropy loss)
40         x = np.zeros(vocab_size) # reset in case you want to hold
41         x[targets[t]] = 1 # encode as 1-of-K
42         deth, dwhh, dby = np.zeros_like(h), np.zeros_like(whh), np.zeros_like(by)
43         dhext = np.zeros_like(h[1:])
44
45         for t in reversed(xrange(len(inputs)-1)):
46             dy[targets[t+1]] = 1 # backprop into y
47             dbyt = np.zeros_like(by[t+1])
48             dbyt[targets[t+1]] = 1 # backprop into y
49             dy += dy
50             dh = -np.dot(Why, T(dy)) + dhext # backprop through tanh nonlinearity
51             dh += dwhh
52             dwhh = np.outer(dy, h[1:-1])
53             dhext = np.dot(Whh, dh[1:-1])
54             dhext = np.dot(dahh, T(dy))
55             dy = np.tanh(dy) # clip to mitigate exploding gradients
56             dy = np.clip(dy, -5, 5, out=dy) # clip to mitigate exploding gradients
57             dbyt = np.zeros_like(by[t])
58             dbyt[targets[t]] = 1
59             dby = np.zeros_like(by[t])
60             dby[targets[t]] = 1
61             dby = np.tanh(dy) # clip to mitigate exploding gradients
62             dby = np.clip(dby, -5, 5, out=dby) # clip to mitigate exploding gradients
63
64     # sample a sequence of integers from the model
65     h = memory state, seed_ix is seed letter for first time step
66     x = np.zeros((vocab_size, 1))
67     ixes = []
68
69     for t in range(seq_length):
70         x[seed_ix] = 1
71         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
72         y = np.dot(Why, h) + by
73         p = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=p.ravel())
75         x = np.zeros((vocab_size, 1))
76         x[ix] = 1
77         ixes.append(ix)
78
79     return ixes
80
81
82 # sample a sequence of integers from the model
83 # h is memory state, seed_ix is seed letter for first time step
84 # x is np.zeros((vocab_size, 1))
85 # ixes is list of sampled integers
86 # i is index
87 # For t in range(seq_length):
88 #   x[seed_ix] = 1
89 #   h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
90 #   y = np.dot(Why, h) + by
91 #   p = np.exp(y) / np.sum(np.exp(y))
92 #   ix = np.random.choice(range(vocab_size), p=p.ravel())
93 #   x[0] = 0
94 #   x[1] = 1
95 #   ixes.append(ix)
96 # return ixes
97
98
99 # a simple loss function
100 # m: min, max: np.zeros_like(inputs), np.zeros_like(targets), np.zeros_like(why)
101 m, max, min, why = np.zeros_like(inputs), np.zeros_like(targets), np.zeros_like(why)
102 m, max, min, why = np.zeros_like(inputs), np.zeros_like(targets), np.zeros_like(why)
103
104 smooth_loss = np.zeros(seq_length)
105
106 for i in range(seq_length):
107     if i % 100 == 0:
108         print('Sampling step %d' % i)
109     for t in range(seq_length):
110         x = np.zeros((vocab_size, 1))
111         x[seed_ix] = 1
112         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
113         y = np.dot(Why, h) + by
114         p = np.exp(y) / np.sum(np.exp(y))
115         ix = np.random.choice(range(vocab_size), p=p.ravel())
116         x[0] = 0
117         x[1] = 1
118         ixes.append(ix)
119
120         # sample from the model now and then
121         if t % 100 == 0:
122             print('Sampling step %d, time %d' % (i, t))
123             txt = ''
124             for ix in ixes[-100:]:
125                 print('...%s...' % ix, end=' ')
126             print()
127
128         # perform parameter update with gradient
129         for para, para_v in zip([wih, whh, bh, by],
130                                [m, max, min, why]):
131             para_v += learning_rate * np.sum(para * dy, axis=0)
132
133         para -= learning_rate * dy / np.sum(dy + 1e-8) # clip gradient update
134
135         m += dy
136         max = np.max(dy)
137         min = np.min(dy)
```

Time step and Batch

- Time steps are defined by **sequences length**
- The batch size is defined by **user**

Time major representation

- `[max_timestep * batch_size * dimension]`

RNNs typically use time major representation

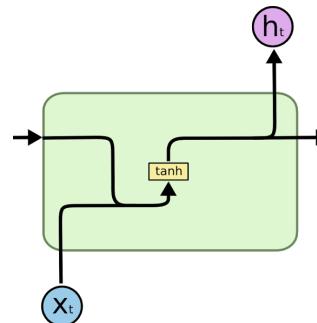
```
outputs = []
#Shape of sentences_in_batch [max_timestep * batch_size * dimension]
for words_in_batch in sentences_in_batch:
    # Shape of words_in_batch [batch_size * dimension]
    some operations here...
    outputs.append(i-th words_output)
#Shape of outputs [max_timestep * batch_size * dimension]
print outputs
```

RNN Cells (Memory Units)

- RNN basic code

```
outputs = []
#Shape of sentences_in_batch [max_timestep * batch_size * dimension]
for words_in_batch in sentences_in_batch:
    # Shape of words_in_batch [batch_size * dimension]
    some operations here...
    outputs.append(i-th words_output)
#Shape of outputs [max_timestep * batch_size * dimension]
print outputs
```

- RNN Cell

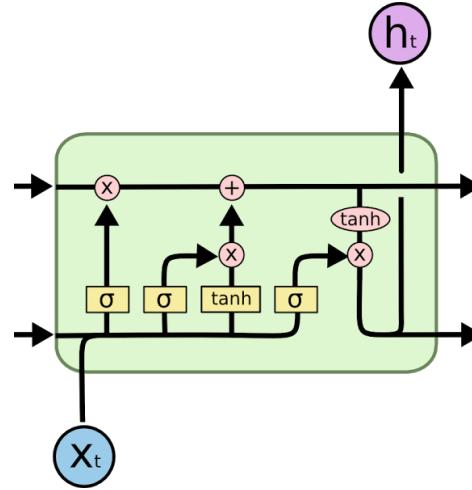


$$h_{t+1} = \tanh(x_t W + h_t U + b)$$

(Image credit: Colah's blog)

RNN Cells (Memory Units)

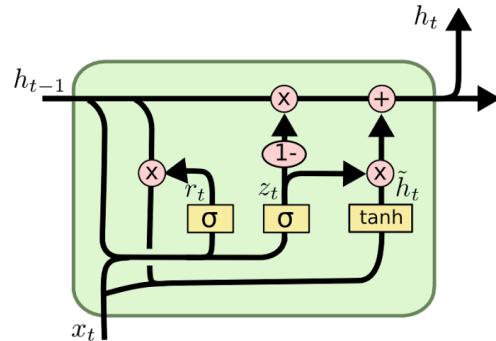
- LSTM Cell



$$\begin{aligned} i &= \sigma(x_t U^i + s_{t-1} W^i) \\ f &= \sigma(x_t U^f + s_{t-1} W^f) \\ o &= \sigma(x_t U^o + s_{t-1} W^o) \\ g &= \tanh(x_t U^g + s_{t-1} W^g) \\ c_t &= c_{t-1} \circ f + g \circ i \\ s_t &= \tanh(c_t) \circ o \end{aligned}$$

RNN Cells (Memory Units)

- GRU Cell



$$z = \sigma(x_t U^z + s_{t-1} W^z)$$

$$r = \sigma(x_t U^r + s_{t-1} W^r)$$

$$h = \tanh(x_t U^h + (s_{t-1} \circ r) W^h)$$

$$s_t = (1 - z) \circ h + z \circ s_{t-1}$$

RNN Loss and Optimization

Simple RNN

$$\begin{aligned}s_t &= \tanh(Ux_t + Ws_{t-1}) \\ \hat{y}_t &= \text{softmax}(Vs_t)\end{aligned}$$

Cross Entropy Loss

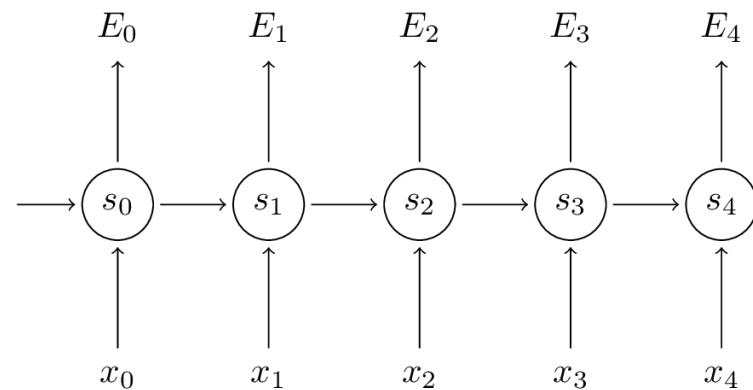
$$\begin{aligned}E(y_t, \hat{y}_t) &= -y_t \log \hat{y}_t \\ E(y, \hat{y}) &= -\sum_t E_t(y_t, \hat{y}_t) \\ &= -\sum_t -y_t \log \hat{y}_t\end{aligned}$$

RNN Loss and Optimization

BackProp

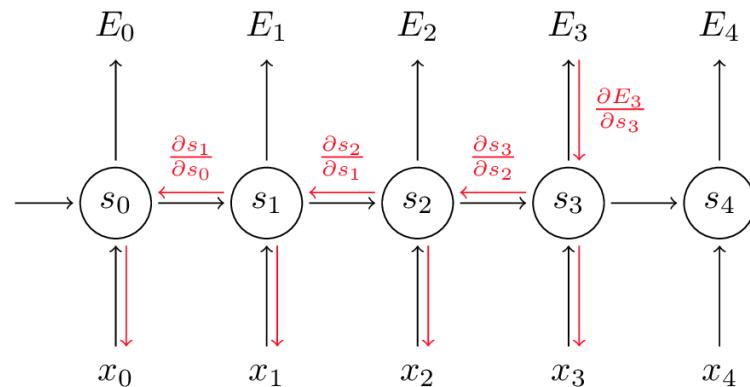
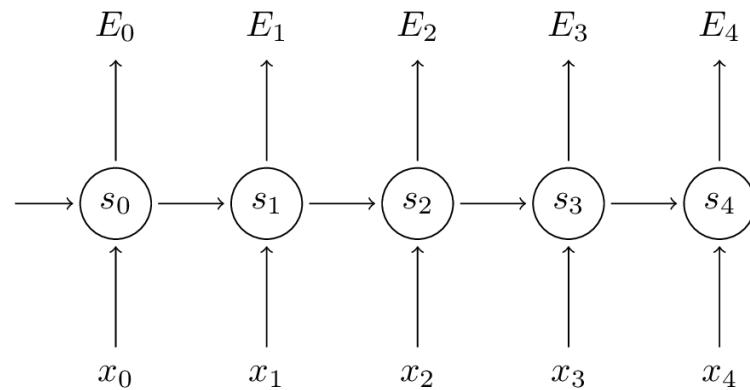
RNN Loss and Optimization

BackProp



RNN Loss and Optimization

BackProp



Tensorflow Basics



Tensorflow Basics

(1) Fetch tensors via `Session.run()`

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
bias = tf.Variable(1.0)

y_pred = x ** 2 + bias      # x -> x^2 + bias
loss = (y - y_pred)**2     # l2 loss?

# Error: to compute loss, y is required as a dependency
print('Loss(x,y) = %.3f' % session.run(loss, {x: 3.0}))

# OK, print 1.000 = (3**2 + 1 - 9)**2
print('Loss(x,y) = %.3f' % session.run(loss, {x: 3.0, y: 9.0}))
```

Tensorflow Basics

(2) Basic operations

```
'''Permuting batch_size and n_steps'''
x = tf.transpose(x, [1, 0, 2])

'''Reshape to (n_steps*batch_size, n_input)'''
x = tf.reshape(x, [-1, n_input])

'''Split to get a list of 'n_steps' tensors of shape (batch_size, n_input)'''
x = tf.split(x, n_steps, 0)

'''Matrix multiplication and bias addition'''
y = tf.matmul(outputs[-1], weights['out']) + biases['out']

'''Data type casting'''
correct_pred = tf.cast(correct_pred, tf.float32)

'''Calculate mean'''
correct_pred = tf.reduce_mean(correct_pred)

'''Adam optimizer'''
tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Tensorflow Basics

(3) Build, Train, Test

```
# Hyper Parameters  
learning_rate = 0.001
```

Tensorflow Basics

(3) Build, Train, Test

```
# Hyper Parameters  
learning_rate = 0.001
```

```
# Network Parameters  
n_input = 28 # MNIST data input (img shape: 28*28)
```

Tensorflow Basics

(3) Build, Train, Test

```
# Hyper Parameters  
learning_rate = 0.001
```

```
# Network Parameters  
n_input = 28 # MNIST data input (img shape: 28*28)
```

```
# tf Graph input  
x = tf.placeholder("float", [None, n_steps, n_input])
```

Tensorflow Basics

(3) Build, Train, Test

```
# Hyper Parameters  
learning_rate = 0.001
```

```
# Network Parameters  
n_input = 28 # MNIST data input (img shape: 28*28)
```

```
# tf Graph input  
x = tf.placeholder("float", [None, n_steps, n_input])
```

```
# Define weights  
weights = {  
    'out': tf.Variable(tf.random_normal([2*n_hidden, n_classes]))  
}
```

Tensorflow Basics

(3) Build, Train, Test

```
# Hyper Parameters  
learning_rate = 0.001
```

```
# Network Parameters  
n_input = 28 # MNIST data input (img shape: 28*28)
```

```
# tf Graph input  
x = tf.placeholder("float", [None, n_steps, n_input])
```

```
# Define weights  
weights = {  
    'out': tf.Variable(tf.random_normal([2*n_hidden, n_classes]))  
}
```

```
# Define model  
def Model(x, weights, biases):  
    Some ops
```

Tensorflow Basics

(3) Build, Train, Test

```
# Get prediction from model  
pred = Model(x, weights, biases)
```

Tensorflow Basics

(3) Build, Train, Test

```
# Get prediction from model  
pred = Model(x, weights, biases)
```

```
# Define loss and optimizer  
cost = tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y)  
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

Tensorflow Basics

(3) Build, Train, Test

```
# Get prediction from model  
pred = Model(x, weights, biases)
```

```
# Define loss and optimizer  
cost = tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y)  
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

```
# Evaluate model  
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))  
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Tensorflow Basics

(3) Build, Train, Test

```
# Get prediction from model  
pred = Model(x, weights, biases)
```

```
# Define loss and optimizer  
cost = tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y)  
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

```
# Evaluate model  
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))  
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

```
# Initializing the variables  
init = tf.initialize_all_variables()
```

Tensorflow Basics

(4) Document

[Tensorflow APIs](#)

[Tensorflow tutorials](#)

[RNN tutorials](#)

Tensorflow APIs for RNN



RNN Cells

Pre-defined class how to calculate the output

```
def call_cell(inputs, state):
    """Most basic RNN"""
    output = new_state = activation(W * input + U * state + B)
    return output
```

- tf.contrib.rnn.RNNCell()
- tf.contrib.rnn.LSTMCell()
- tf.contrib.rnn.GRUCell()

Where do I have to call `call_cell()`

```
outputs = []
for words_in_batch in sentences_in_batch:
    # Shape of words_in_batch [batch_size * dimension]
    some operations here...
    outputs.append(i-th words_output)
#Shape of outputs [max_timestep * batch_size * dimension]
print outputs
```

Using RNN wrapper

```
# Define a lstm cell with tensorflow
lstm_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)

# Get lstm cell output of RNN
outputs, states = rnn.static_rnn(lstm_cell, x)

# Get lstm cell output of BRNN
outputs, output_state_fw, output_state_bw = rnn.static_bidirectional_rnn(lstm_fw_cell, ls
```

RNN Loss

```
#Activation of the last fully connected layer  
pred = tf.matmul(outputs[-1], weights['out']) + biases['out']  
  
#Softmax cross entropy loss  
#Internally calculate softmax  
tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y)
```

Let's Do It

1_practice_rnn.ipynb



MNIST Dataset

- Hand written digits
- 10 classes
- 28 * 28 size images
- Labels are one-hot encoded

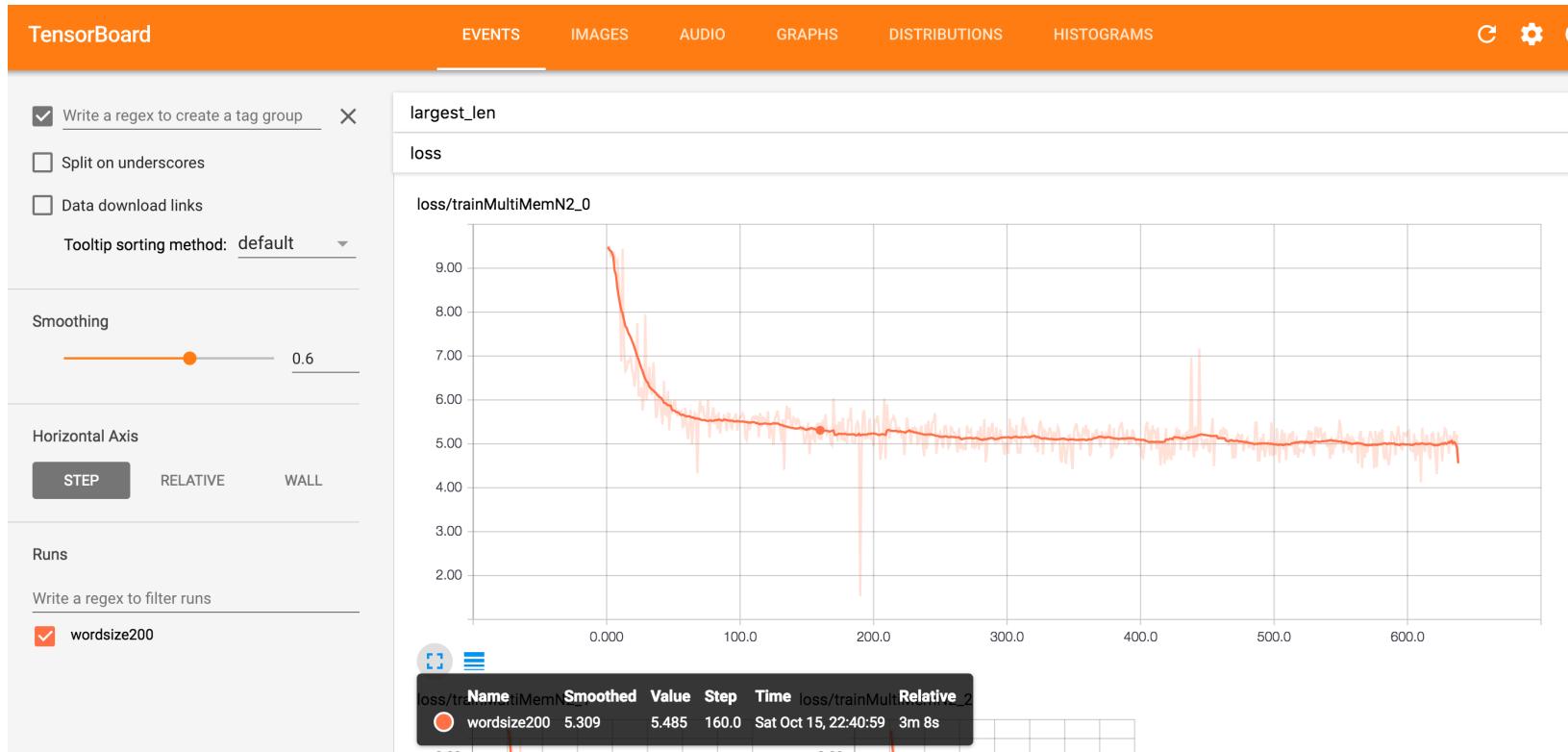


Monitoring

2_practice_monitoring.ipynb



Visualizing Log Data with TensorBoard



Visualizing Log Data with TensorBoard

```
'''Run tensorboard'''
$tensorboard --logdir=./ --port 1234

# Create a summary to monitor cost tensor
tf.summary.scalar("loss", cost)

# Create a summary to monitor accuracy tensor
tf.summary.scalar("accuracy", accuracy)

# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()

# op to write logs to Tensorboard
summary_writer = tf.train.SummaryWriter(logs_path, graph=tf.get_default_graph())

# Write logs at every iteration
summary_writer.add_summary(summary, epoch * total_batch + i)
```

Visualizing Log Data with TensorBoard

```
# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf.initialize_all_variables()

# Create a summary to monitor cost tensor
tf.scalar_summary("loss", cost)

# Create a summary to monitor accuracy tensor
tf.scalar_summary("accuracy", accuracy)

# Merge all summaries into a single op
merged_summary_op = tf.merge_all_summaries()
```

```
# Write logs at every iteration
summary_writer = tf.train.SummaryWriter(logs_path, graph=tf.get_default_graph())

while step * batch_size < training_iters:
    # Calculate batch loss
    loss, acc, summary = sess.run([cost, accuracy, merged_summary_op], \
        feed_dict={x: batch_x, y: batch_y})

    # Write logs at every iteration
    summary_writer.add_summary(summary, step)
```

Advanced



Difficulty of training RNNs

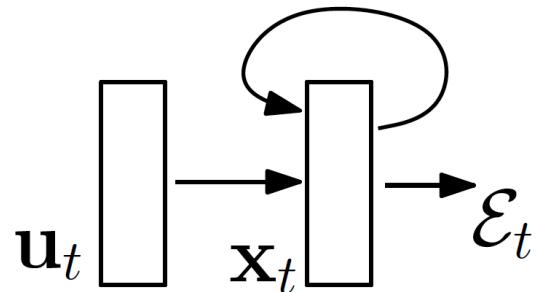


Fig. Schematic of a simple RNN

$$\mathbf{x}_t = F(\mathbf{x}_{t-1}, \mathbf{u}_t, \theta)$$

$$\mathbf{x}_t = \mathbf{W}_{rec}\sigma(\mathbf{x}_{t-1}) + \mathbf{W}_{in}\mathbf{u}_t + \mathbf{b}$$

Difficulty of training RNNs

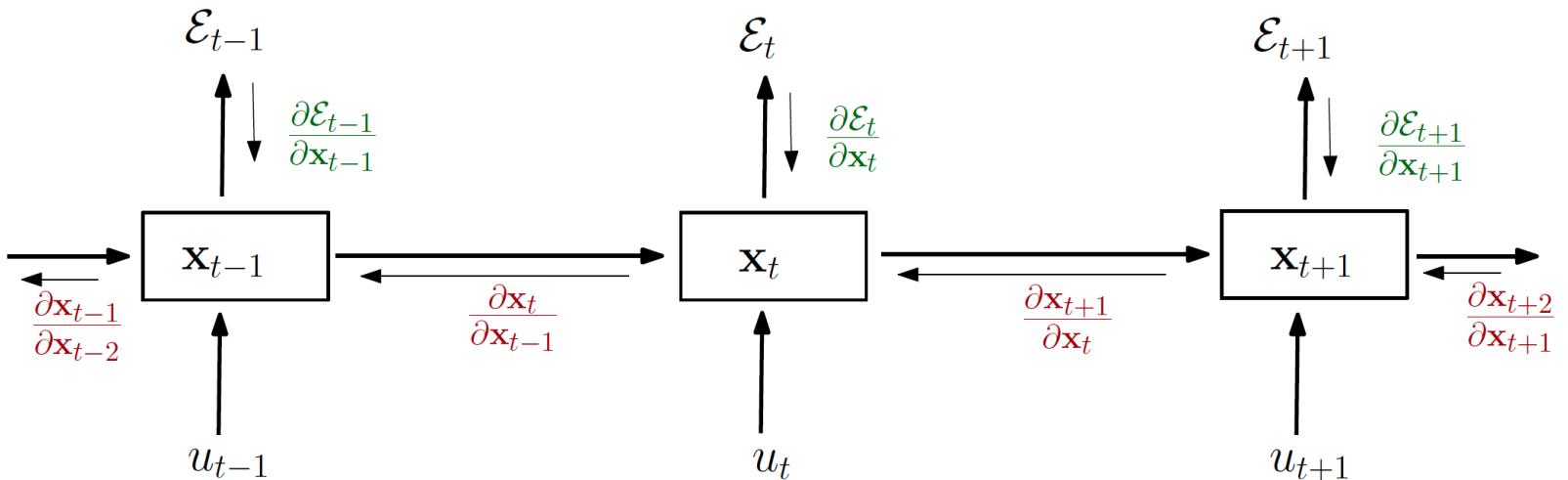


Fig. Unrolling RNNs in time by creating a copy of the model for each time step

$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial \mathcal{E}_t}{\partial \theta}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \leq k \leq t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \geq i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}}$$

Difficulty of training RNNs

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \leq k \leq t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \geq i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}}$$

Eq. Temporal Contributions

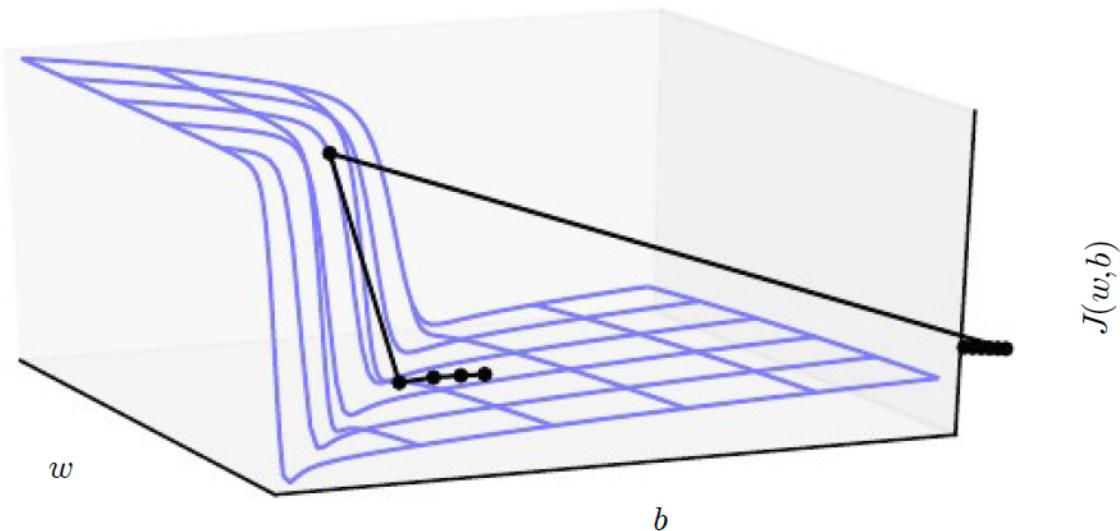
- This Equation shows that how θ at step k affects the cost at step $t > k$.
- The factors $\frac{\partial x_t}{\partial x_k}$ transport the error "in time" from step t back to step k .
- We called it **long term** for which $k \ll t$.

Difficulty of training RNNs

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \geq i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \geq i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1}))$$

Eq. Exploding and Vanishing Gradients

- For long term this equations show that gradients can be exploding or vanishing.



Difficulty of training RNNs

Previous solutions

- Using L1 or L2 penalty on the recurrent weights
 - It can help exploding gradients, but not for vanishing.
 - This approach limits the model to a simple regime, where any information inserted in the model has to die out exponentially fast in time.
- Teacher forcing
 - This can help the model go to the right region of space.
 - It can reduce the chance that gradients exploding.
 - This approach requires a target to be defined at **every time**.
- Memory unit (ex LSTM or GRU)
 - This can help vanishing gradients.
 - This approach does not address explicitly the exploding gradients problem.

Difficulty of training RNNs

Gradient clipping

```
 $\hat{g} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$ 
if  $\|\hat{g}\| \geq threshold$  then
     $\hat{g} \leftarrow \frac{threshold}{\|\hat{g}\|} \hat{g}$ 
end if
```

Alg. Gradient clipping

Simple mechanism to deal with a sudden increase in the norm of the gradients is to rescale them whenever they go over a threshold.

- In experiments, training is not very sensitive to this hyperparameter and the algorithm behaves well even for rather small thresholds.

Gradient Clipping

```
'''Don't use optimizer.minimize()'''

#Calculate gradients
grads = optimizer.compute_gradients(loss)

#Gradient Clipping
clipped_grads_and_vars = [(tf.clip_by_norm(gv[0], self.max_grad_norm), gv[1]) for gv in g

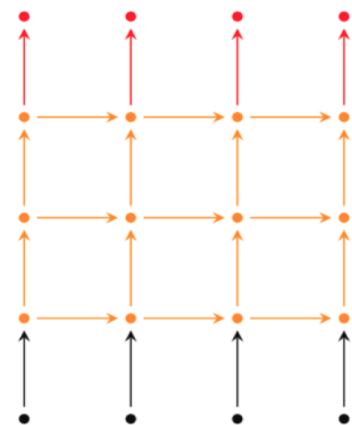
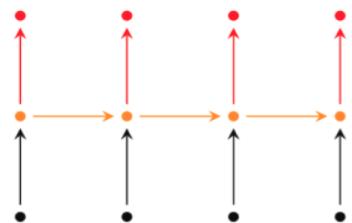
#Apply Gradient
optim = optimizer.apply_gradients(clipped_grads_and_vars, global_step=self.global_step)
```

RNN vs. Dynamic RNN

- `tf.nn.rnn` creates an unrolled graph for a fixed RNN length
 - e.g. creating static graph with 200 RNN steps
 - 1) graph creation is slow
 - 2) unable to pass longer sequences (>200) than originally specified
- `tf.nn.dynamic_rnn` solves this!
 - Internally use `tf.While` to dynamically construct the graph
 - 1) faster creation
 - 2) can feed batches of variable size

In short, **just use `tf.nn.dynamic_rnn`.**

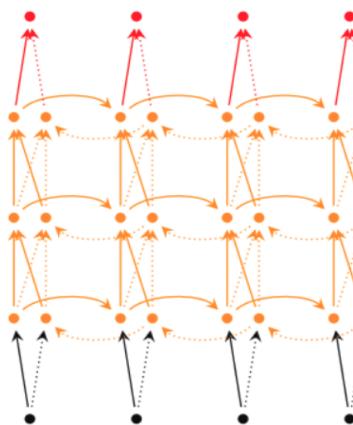
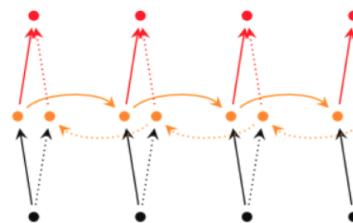
RNN



$$h_t = f(Wx_t + Vh_{t-1} + b) \quad (1)$$

$$y_t = g(Uh_t + c) \quad (2)$$

BRNN



$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b}) \quad (3)$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b}) \quad (4)$$

$$y_t = g(\vec{U}\vec{h}_t + \overleftarrow{U}\overleftarrow{h}_t + c) \quad (5)$$

Dynamic RNN

```
# Create input data
X = np.random.randn(2, 10, 8)

# The second example is of length 6
X[1,6:] = 0
X_lengths = [10,6]

cell = tf.contrib.rnn.LSTMCell(num_units=64, state_is_tuple=True)

outputs, last_states = tf.nn.dynamic_rnn(
    cell=cell,
    dtype=tf.float64,
    sequence_length=X_lengths,
    inputs=X)

result = tf.contrib.learn.run_n(
    {"outputs":outputs, "last_states": last_states},
    n=1,
    feed_dict=None)
```

Bidirectional Dynamic RNN

```
# Create input data
X = np.random.randn(2, 10, 8)

# The second example is of length 6
X[1,6:] = 0
X_lengths = [10,6]

cell = tf.contrib.rnn.LSTMCell(num_units=64, state_is_tuple=True)

outputs, last_states = tf.nn.bidirectional_dynamic_rnn(
    cell_fw=cell,
    cell_bw=cell,
    dtype=tf.float64,
    sequence_length=X_lengths,
    inputs=X)

output_fw, output_bw = outputs
states_fw, states_bw = states
```

Practice : character Level RNN

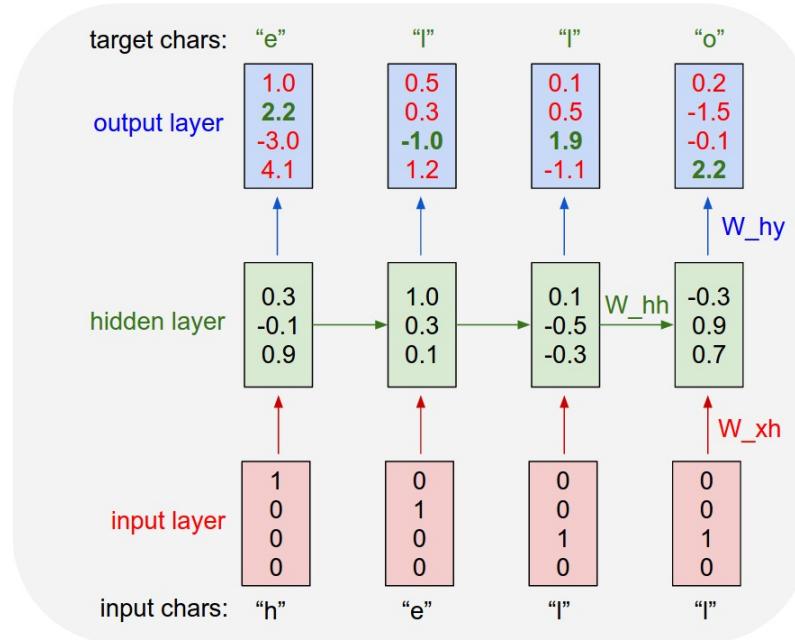
3_char_rnn_train.ipynb, 3_char_rnn_inference.ipynb



Make code generate code!

More results

- Char RNN architecture



- Train Linux kernel code
- RNN learns code pattern in character level

3_char_rnn_train.ipynb

- Load linux kernel code

```
# Load text
data_dir      = "data/linux_kernel"
save_dir      = "data/linux_kernel"
input_file   = os.path.join(data_dir, "input.txt")
with open(input_file, "r") as f:
    data = f.read()
print ("Text loaded from '%s'" % (input_file))
```

And preprocess text

```
# Preprocess Text
# First, count the number of characters
counter = collections.Counter(data)
count_pairs = sorted(counter.items(), key=lambda x: -x[1]) # <= Sort
print ("Type of 'counter.items()' is %s and length is %d"
      % (type(counter.items()), len(counter.items())))
for i in range(5):
    print ("[%d/%d]" % (i, 3)), # <= This comma remove '\n'
    print (list(counter.items())[i])

print (" ")
print ("Type of 'count_pairs' is %s and length is %d"
      % (type(count_pairs), len(count_pairs)))
for i in range(5):
    print ("[%d/%d]" % (i, 3)), # <= This comma remove '\n'
    print (count_pairs[i])
```

Make Dictionary and Vocabulary.

- Map Character to digit (index)

```
# Let's make dictionary
chars, counts = zip(*count_pairs)
vocab = dict(zip(chars, range(len(chars))))
print ("Type of 'chars' is %s and length is %d"
      % (type(chars), len(chars)))
for i in range(5):
    print ("[%d/%d]" % (i, 3)), # <= This comma remove '\n'
    print ("chars[%d] is '%s'" % (i, chars[i]))

print ("")
print ("Type of 'vocab' is %s and length is %d"
      % (type(vocab), len(vocab)))
for i in range(5):
    print ("[%d/%d]" % (i, 3)), # <= This comma remove '\n'
    print ("vocab[%s] is %s" % (chars[i], vocab[chars[i]]))

# Save chars and vocab
with open(os.path.join(save_dir, 'chars_vocab.pkl'), 'wb') as f:
    pickle.dump((chars, vocab), f)
```

Make Dictionary and Vocabulary.

- Map Character to digit (index)

```
Type of 'chars' is <type 'tuple'> and length is 99
```

```
[0/3] chars[0] is ' '
[1/3] chars[1] is 'e'
[2/3] chars[2] is 't'
[3/3] chars[3] is 'r'
[4/3] chars[4] is 'i'
```

```
Type of 'vocab' is <type 'dict'> and length is 99
```

```
[0/3] vocab[' '] is 0
[1/3] vocab['e'] is 1
[2/3] vocab['t'] is 2
[3/3] vocab['r'] is 3
[4/3] vocab['i'] is 4
```

- Converts index to char

```
# Now convert all text to index using vocab!
corpus = np.array(list(map(vocab.get, data)))
print ("Type of 'corpus' is %s, shape is %s, and length is %d"
       % (type(corpus), corpus.shape, len(corpus)))

check_len = 10
print ("\n'corpus' looks like %s" % (corpus[0:check_len]))
for i in range(check_len):
    _wordidx = corpus[i]
    print ("%d/%d] chars[%02d] corresponds to '%s'"
          % (i, check_len, _wordidx, chars[_wordidx]))
```

Type of 'corpus' is <type 'numpy.ndarray'>, shape is (1708871,), and length is 1708871

```
'corpus' looks like [36 22 7 0 22 0 0 13 4 8]
[0/10] chars[36] corresponds to '/'
[1/10] chars[22] corresponds to '*'
[2/10] chars[07] corresponds to '
```

Generate Batch data

```
# Generate batch data
batch_size = 50
seq_length = 200
num_batches = int(corpus.size / (batch_size * seq_length))
# First, reduce the length of corpus to fit batch_size
corpus_reduced = corpus[::(num_batches*batch_size*seq_length)]
xdata = corpus_reduced
ydata = np.copy(xdata)
ydata[:-1] = xdata[1:]
ydata[-1] = xdata[0]
...
```

```
xdata is ... [36 22 7 ..., 11 25 3] and length is 1700000
ydata is ... [22 7 0 ..., 25 3 36] and length is 1700000
```

```
Type of 'xbatches' is <type 'list'> and length is 170
Type of 'ybatches' is <type 'list'> and length is 170
```

```
Type of 'temp' is <type 'list'> and length is 5
Type of 'temp[0]' is <type 'numpy.ndarray'> and shape is (50, 200)
```

Now, we are ready to make our RNN model

```
# Important RNN parameters
vocab_size = len(vocab)
rnn_size   = 128
num_layers = 2
grad_clip  = 5.

def unit_cell():
    return tf.contrib.rnn.BasicLSTMCell(rnn_size, state_is_tuple=True, reuse=tf.get_variable_scope().reuse)

cell = tf.contrib.rnn.MultiRNNCell([unit_cell() for _ in range(num_layers)])

input_data = tf.placeholder(tf.int32, [batch_size, seq_length])
targets    = tf.placeholder(tf.int32, [batch_size, seq_length])
istate     = cell.zero_state(batch_size, tf.float32)
# Weights
with tf.variable_scope('rnnlm'):
    softmax_w = tf.get_variable("softmax_w", [rnn_size, vocab_size])
    softmax_b = tf.get_variable("softmax_b", [vocab_size])
    with tf.device("/cpu:0"):
        embedding = tf.get_variable("embedding", [vocab_size, rnn_size])
        inputs = tf.split(tf.nn.embedding_lookup(embedding, input_data), seq_length, 1)
        inputs = [tf.squeeze(_input, [1]) for _input in inputs]
```

```

# Output
def loop(prev, _):
    prev = tf.nn.xw_plus_b(prev, softmax_w, softmax_b)
    prev_symbol = tf.stop_gradient(tf.argmax(prev, 1))
    return tf.nn.embedding_lookup(embedding, prev_symbol)

"""
loop_function: If not None, this function will be applied to the i-th output
in order to generate the i+1-st input, and decoder_inputs will be ignored,
except for the first element ("GO" symbol).
"""

outputs, last_state = tf.contrib.rnn.static_rnn(cell, inputs, istate
                                                , scope='rnnlm')
output = tf.reshape(tf.concat(outputs, 1), [-1, rnn_size])
logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)
probs = tf.nn.softmax(logits)

# Loss
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], # Input
    [tf.reshape(targets, [-1])], # Target
    [tf.ones([batch_size * seq_length])], # Weight
    vocab_size)

# Optimizer
cost = tf.reduce_sum(loss) / batch_size / seq_length
final_state = last_state
lr = tf.Variable(0.0, trainable=False)
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), grad_clip)
_optm = tf.train.AdamOptimizer(lr)
optm = _optm.apply_gradients(zip(grads, tvars))

```

Training code

```
# Train the model!
num_epochs      = 50
save_every      = 500
learning_rate   = 0.002
decay_rate      = 0.97

sess = tf.Session()
sess.run(tf.initialize_all_variables())
summary_writer = tf.summary.FileWriter(save_dir, graph=sess.graph)
saver = tf.train.Saver(tf.all_variables())
init_time = time.time()
for epoch in range(num_epochs):
    # Some Training code!
```

Instructions for updating:

Please use `tf.global_variables` instead.

```
[0/8500] cost: 4.6006 / Each batch learning took 2.2222 sec
model saved to 'data/linux_kernel/model.ckpt'
[100/8500] cost: 3.1259 / Each batch learning took 0.3366 sec
[200/8500] cost: 2.5992 / Each batch learning took 0.3258 sec
[300/8500] cost: 2.4603 / Each batch learning took 0.3260 sec
[400/8500] cost: 2.2591 / Each batch learning took 0.3136 sec
[500/8500] cost: 2.0035 / Each batch learning took 0.3140 sec
```

3_char_rnn_inference.ipynb

- Load data (Like training script)
- Set same network
- And generate sample

```
# Sampling function
def weighted_pick(weights):
    t = np.cumsum(weights)
    s = np.sum(weights)
    return(int(np.searchsorted(t, np.random.rand(1)*s)))

# Sample using RNN and prime characters
prime = /* "
state = sess.run(cell.zero_state(1, tf.float32))
for char in prime[:-1]:
    x = np.zeros((1, 1))
    x[0, 0] = vocab[char]
    state = sess.run(last_state, feed_dict={input_data: x, istate:state})

# Sample 'num' characters
ret = prime
char = prime[-1] # <= This goes IN!
num = 1000
```

```

for n in range(num):
    x = np.zeros((1, 1))
    x[0, 0] = vocab[char]
    [probsval, state] = sess.run([probs, last_state]
        , feed_dict={input_data: x, istate:state})
    p      = probsval[0]

    sample = weighted_pick(p)
    # sample = np.argmax(p)

    pred   = chars[sample]
    ret    = ret + pred
    char   = pred

```

Sampling Done.

```

/* struct auditued oq,
   struct audit_enables can in a
* the->module */
   else /* SMP a signals, ne-time bew releas for rebining routine  SIBGR.
*
* Copy: state if exits/help6Counter don't be NULL field "olp.
*/
void sysctl_sched_kobjed(&watchdalleep_state, int sys_timek_ops, *mask, struct filt, unsigned int

```

More results

- Latex generation. The resulting sampled Latex almost compiles.

Proof. Omitted. \square

Lemma 0.1. Let \mathcal{C} be a set of the construction.

Let \mathcal{C} be a gerber covering. Let \mathcal{F} be a quasi-coherent sheaves of \mathcal{O} -modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

Proof. This is an algebraic space with the composition of sheaves \mathcal{F} on $X_{\text{étale}}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morph}_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where \mathcal{G} defines an isomorphism $\mathcal{F} \rightarrow \mathcal{F}$ of \mathcal{O} -modules. \square

Lemma 0.2. This is an integer \mathcal{Z} is injective.

Proof. See Spaces, Lemma ??.

Lemma 0.3. Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \rightarrow Y' \rightarrow Y \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X.$$

be a morphism of algebraic spaces over S and Y .

Proof. Let X be a nonzero scheme of X . Let X be an algebraic space. Let \mathcal{F} be a quasi-coherent sheaf of \mathcal{O}_X -modules. The following are equivalent

- (1) \mathcal{F} is an algebraic space over S .
- (2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. \square

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccccc}
 S & \xrightarrow{\quad} & & & \\
 \downarrow & & & & \\
 \xi & \xrightarrow{\quad} & \mathcal{O}_{X'} & \xrightarrow{\quad} & \\
 \text{gor}_s & & \uparrow & \searrow & \\
 & & =\alpha' & \longrightarrow & \\
 & & \uparrow & & \\
 & & =\alpha' & \longrightarrow & \alpha \\
 & & & & \\
 \text{Spec}(K_\psi) & & \text{Mor}_{\text{Sets}} & & d(\mathcal{O}_{X_{/\mathbb{A}}}, \mathcal{G}) \\
 & & & & \\
 & & & & X \\
 & & & & \downarrow
 \end{array}$$

is a limit. Then \mathcal{G} is a finite type and assume S is a flat and \mathcal{F} and \mathcal{G} is a finite type f_* . This is of finite type diagrams, and

- the composition of \mathcal{G} is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

Proof. We have see that $X = \text{Spec}(R)$ and \mathcal{F} is a finite type representable by algebraic space. The property \mathcal{F} is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U . \square

Proof. This is clear that \mathcal{G} is a finite presentation, see Lemmas ??.

A reduced above we conclude that U is an open covering of \mathcal{C} . The functor \mathcal{F} is a “field”

$$\mathcal{O}_{X,x} \rightarrow \mathcal{F}_{\bar{x}} \dashv (\mathcal{O}_{X_{\text{étale}}}) \rightarrow \mathcal{O}_{X_{\bar{x}}}^{-1} \mathcal{O}_{X_{\bar{x}}}(\mathcal{O}_{X_{\bar{x}}}^{\text{pt}})$$

is an isomorphism of covering of $\mathcal{O}_{X_{\bar{x}}}$. If \mathcal{F} is the unique element of \mathcal{F} such that X is an isomorphism.

The property \mathcal{F} is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme \mathcal{O}_X -algebra with \mathcal{F} are opens of finite type over S .

If \mathcal{F} is a scheme theoretic image points. \square

If \mathcal{F} is a finite direct sum $\mathcal{O}_{X_{\bar{x}}}$ is a closed immersion, see Lemma ?? . This is a sequence of \mathcal{F} is a similar morphism.

More results

- Latex generation. The resulting sampled Latex almost compiles.

For $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n} = 0$, hence we can find a closed subset \mathcal{H} in \mathcal{H} and any sets \mathcal{F} on X , U is a closed immersion of S , then $U \rightarrow T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fppf} and $U \rightarrow U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ???. Hence we obtain a scheme S and any open subset $W \subset U$ in $\text{Sh}(G)$ such that $\text{Spec}(R') \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\text{GL}_{S'}(x'/S'')$ and we win. \square

To prove study we see that $\mathcal{F}|_U$ is a covering of X' , and \mathcal{T}_i is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and \mathcal{F}_p exists and let \mathcal{F}_i be a presheaf of \mathcal{O}_X -modules on \mathcal{C} as a \mathcal{F} -module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)^{\text{opp}}_{fppf}, (\text{Sch}/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longrightarrow (U, \text{Spec}(A))$$

is an open subset of X . Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S .

Proof. See discussion of sheaves of sets. \square

The result for prove any open covering follows from the less of Example ???. It may replace S by $X_{\text{spaces},\text{étale}}$ which gives an open subspace of X and T equal to S_{Zar} , see Descent, Lemma ???. Namely, by Lemma ?? we see that R is geometrically regular over S .

Lemma 0.1. Assume (3) and (3) by the construction in the description.

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{\text{Proj}}_X(\mathcal{A}) = \text{Spec}(B)$ over U compatible with the complex

$$\text{Set}(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \rightarrow \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S . Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) f is locally of finite type. Since $S = \text{Spec}(R)$ and $Y = \text{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X . But given a scheme U and a surjective étale morphism $U \rightarrow X$. Let $U \cap U = \coprod_{i=1,\dots,n} U_i$ be the scheme X over S at the schemes $X_i \rightarrow X$ and $U = \lim_i X_i$. \square

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{x,\dots,0}$.

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq p$ is a subset of $\mathcal{J}_{n,0} \circ \mathcal{A}_2$ works.

Lemma 0.3. In Situation ???. Hence we may assume $q' = 0$.

Proof. We will use the property we see that p is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F -algebra where δ_{n+1} is a scheme over S . \square

Practice : (Korean) Novel Writer (Char RNN)

4_kor_char_rnn_train.ipynb, 4_kor_char_rnn_inference.ipynb



RNN generate korean novel

- Train korean raw text data
- Convert korean character to sequence pattern
 - Hangulpy - <https://github.com/rhobot/Hangulpy>
- RNN learns sequence pattern and generate similar one.

"크아아아아"

드래곤중에서도 최강의 투명드래곤이 울부짖었다
투명드래곤은 졸라짱께서 드래곤중에서 최강이었다
신이나 마족도 이겼따 다덤벼도 이겼따 투명드래곤은
세상에서 하나였다 어쨌든 개가 울부짖었다

"으악 제기랄 도망가자"

발록들이 도망갔다 투명드래곤이 짱이었따
그래서 발록들은 도망간 것이다

계속

```
import chardet # https://github.com/chardet/chardet
from TextLoader import *
from Hangulpy import *
print ("PACKAGES LOADED")
```

Conversion Function to utf-8 format

```
def conv_file(fromfile, tofile):
    with open(fromfile, "rb") as f:
        sample_text=f.read(10240)
    pred = chardet.detect(sample_text)
    if not pred[ 'encoding' ] in ('EUC-KR', 'UTF-8', 'CP949', 'UTF-16LE'):
        print ("WARNING! Unknown encoding! : %s = %s" % (fromfile, pred[ 'encoding' ]))
        pred[ 'encoding' ] = "CP949" # 못찾으면 기본이 CP949
        fromfile = fromfile + ".unknown"
    elif pred[ 'confidence' ] < 0.9:
        print ("WARNING! Unsure encofing! : %s = %s / %s")
        % (fromfile, pred[ 'confidence' ], pred[ 'encoding' ])
        fromfile = fromfile + ".notsure"
    with codecs.open(fromfile, "r", encoding=pred[ 'encoding' ], errors="ignore") as f:
        with codecs.open(tofile, "w+", encoding="utf8") as t:
            all_text = f.read()
            t.write(all_text)
```

Convert raw text data to utf-8 format

```
# Downloaded data
dataname = 'invisible_dragon'
# SOURCE TXT FILE
fromfile = os.path.join("data",dataname,"rawtext.txt")
# TARGET TXT FILE
tofile   = os.path.join("data",dataname,"rawtext_utf8.txt")
conv_file(fromfile, tofile)
print ("UTF8-CONVERTING DONE")
print (" [{}] IS GENERATED" % (tofile))
```

```
def dump_file(filename):
    result=u"" # <= UNICODE STRING
    with codecs.open(filename, "r", encoding="UTF8") as f:
        for line in f.readlines():
            line = tuple(line)
            result = result + decompose_text(line)
    return result

parsed_txt = dump_file(tofile).encode("utf8")
```

Parsing data/invisible_dragon/rawtext_utf8.txt done
여러분 재가 드디어 글을...
○ㅋㄹㄹㅓㅓㅂㅡㄴㅓㅈㅐㅓㄱㅏㅓㄷㅡㅓㄷㅣㅓㅇㅓㅓㄱㅡㄹㅓㅓㅡ?

- Generate "input.txt" (Parsed pattern)

```
with open(os.path.join("data",dataname,"input.txt"), "w") as text_file:  
    text_file.write(parsed_txt)  
print ("Saved to a txt file")  
print (text_file)
```

- Generate "vocab.pkl" and "data.npy"

```
data_dir      = "data/invisible_dragon"  
batch_size   = 50  
seq_length   = 50  
data_loader = TextLoader(data_dir, batch_size, seq_length)
```

- Shift + Enter repeat. From start to end

4_kor_char_rnn_train.ipynb

Load preprocessed dataset with text loader

```
corpus_name = "invisible_dragon" # "nine_dreams"  
  
data_dir      = "data/" + corpus_name  
batch_size    = 10  
seq_length   = 100  
data_loader = TextLoader(data_dir, batch_size, seq_length)
```

Generate Batch.

```
x, y = data_loader.next_batch()
```

Type of 'x' is <type 'numpy.ndarray'>. Shape is (50, 50)

x looks like

```
[[ 7  6  1 ...,  3  0 37]
 [15 18  0 ...,  0 19  3]
 [38  3  0 ..., 61 50  7]
 ...
 [ 0  5  3 ...,  0  2 21]
 [ 8  0  2 ..., 12  5  0]
 [ 3  1  0 ..., 21 12  0]]
```

Type of 'y' is <type 'numpy.ndarray'>. Shape is (50, 50)

y looks like

```
[[ 6  1 20 ...,  0 37  7]
 [18  0  5 ..., 19  3  4]
 [ 3  0  2 ..., 50  7  6]
 ...
 [ 5  3  0 ...,  2 21 12]
 [ 0  2 19 ...,  5  0 15]
 [ 1  0  1 ..., 12  0 11]]
```

Define a multilayer LSTM network graph

```
rnn_size    = 512
num_layers  = 3
grad_clip   = 5. # <= GRADIENT CLIPPING (PRACTICALLY IMPORTANT)
vocab_size  = data_loader.vocab_size

# SELECT RNN CELL (MULTI LAYER LSTM)
def unit_cell():
    return tf.contrib.rnn.BasicLSTMCell(rnn_size, state_is_tuple=True, reuse=tf.get_variable_scope().reuse)
cell = tf.contrib.rnn.MultiRNNCell([unit_cell() for _ in range(num_layers)])

# Set paths to the graph
input_data = tf.placeholder(tf.int32, [batch_size, seq_length])
targets    = tf.placeholder(tf.int32, [batch_size, seq_length])
initial_state = cell.zero_state(batch_size, tf.float32)

# Set Network
with tf.variable_scope('rnnlm'):
    softmax_w = tf.get_variable("softmax_w", [rnn_size, vocab_size])
    softmax_b = tf.get_variable("softmax_b", [vocab_size])
    with tf.device("/cpu:0"):
        embedding = tf.get_variable("embedding", [vocab_size, rnn_size])
        inputs = tf.split(tf.nn.embedding_lookup(embedding, input_data), seq_length, 1)
        inputs = [tf.squeeze(input_, [1]) for input_ in inputs]
print ("Network ready")
```

Define a function

```
# Output of RNN
outputs, last_state = tf.contrib.rnn.static_rnn(cell, inputs, initial_state,
                                                scope='rnnlm')

output = tf.reshape(tf.concat(outputs,1), [-1, rnn_size])
logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)

# Next word probability
probs = tf.nn.softmax(logits)
print ("FUNCTIONS READY")
```

Define Loss functions

```
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], # Input
    [tf.reshape(targets, [-1])], # Target
    [tf.ones([batch_size * seq_length])], # Weight
    vocab_size)
print ("LOSS FUNCTION")
```

Define Cost functions

```
cost = tf.reduce_sum(loss) / batch_size / seq_length

# GRADIENT CLIPPING !
lr = tf.Variable(0.0, trainable=False) # <= LEARNING RATE
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), grad_clip)
_optm = tf.train.AdamOptimizer(lr)
optm = _optm.apply_gradients(zip(grads, tvars))

final_state = last_state
print ("NETWORK READY")
```

Optimize Network

```
num_epochs      = 5000
save_every      = 500
learning_rate   = 0.001
decay_rate      = 0.999

save_dir = 'data/' + corpus_name
sess = tf.InteractiveSession()

sess.run(tf.initialize_all_variables())
```

Run Training! (It takes very long time)

```
summary_writer = tf.summary.FileWriter(save_dir
                                      , graph=sess.graph)
saver = tf.train.Saver(tf.all_variables())
for e in range(num_epochs): # for all epochs
    # LEARNING RATE SCHEDULING
    sess.run(tf.assign(lr, learning_rate * (decay_rate ** e)))

    data_loader.reset_batch_pointer()
    state = sess.run(initial_state)
    for b in range(data_loader.num_batches):
        start = time.time()
        x, y = data_loader.next_batch()
        feed = {input_data: x, targets: y, initial_state: state}
        # Train!
        train_loss, state, _ = sess.run([cost, final_state, optm], feed)
        end = time.time()
        # PRINT
        if b % 100 == 0:
            print ("%d/%d (epoch: %d), loss: %.3f, time/batch: %.3f"
                  % (e * data_loader.num_batches + b
                     , num_epochs * data_loader.num_batches
                     , e, train_loss, end - start))
```

```
# SAVE MODEL
if (e * data_loader.num_batches + b) % save_every == 0:
    checkpoint_path = os.path.join(save_dir, 'model.ckpt')
    saver.save(sess, checkpoint_path
               , global_step = e * data_loader.num_batches + b)
    print("model saved to {}".format(checkpoint_path))
```

4_kor_char_rnn_inference.ipynb

Set corpus name

```
corpus_name = "invisible_dragon" # "nine_dreams"
data_dir     = "data/" + corpus_name
batch_size   = 10
seq_length   = 100
data_loader = TextLoader(data_dir, batch_size, seq_length)
# This makes "vocab.pkl" and "data.npy" in "data/nine_dreams"
# from "data/nine_dreams/input.txt"
vocab_size = data_loader.vocab_size
vocab = data_loader.vocab
chars = data_loader.chars
print( "type of 'data_loader' is %s, length is %d"
      % (type(data_loader.vocab), len(data_loader.vocab)) )
print( "\n" )
print( "data_loader.vocab looks like \n%s " %
      (data_loader.vocab))
print( "\n" )
print( "type of 'data_loader.chars' is %s, length is %d"
      % (type(data_loader.chars), len(data_loader.chars)) )
print( "\n" )
print( "data_loader.chars looks like \n%s " % (data_loader.chars,) )
```

Sample character from predicted output

```
sample = int(np.random.choice(len(p), p=p))
# sample = weighted_pick(p)
# sample = np.argmax(p)
pred = chars[sample]
ret += pred
char = pred
return ret
print ("sampling function done.")
```

Inference Result

```
save_dir = 'data/' + corpus_name
prime = decompose_text(u"누구 ")

print ("Prime Text : %s => %s" % (automata(prime), "".join(prime)))
n = 4000

sess = tf.Session()
sess.run(tf.initialize_all_variables())
saver = tf.train.Saver(tf.all_variables())
ckpt = tf.train.get_checkpoint_state(save_dir)

# load_name = u'data/nine_dreams/model.ckpt-0'
load_name = os.path.join(save_dir, 'model.ckpt-1900')

print (load_name)

if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, load_name)
    sampled_text = sample(sess, chars, vocab, probs, n, prime)
    #print ("")
#    print (u"SAMPLED TEXT = %s" % sampled_text)
#    print ("")
print ("-- RESULT --")
print (automata("".join(sampled_text)))
```

Inference Result

- Step 2500

투명드래곤 선보았터 조노 놓옹가다 가다망어흔

즈시가 진장간 타어져 질롱해드름대 뒤크은 그기각드어레 무됨데~?~

수저질개널 진짜 은이얼에주야나트 ㅋㅏ ㅆ다! .

"투명드래곤오 시빡임언당.

뚜크이~!!!

티멍드른곤언 셩ㄱㅂㄹ에 수대아오가격핵말타 숙테몇에 ?~
삽승 때개근!

세복 막5ㅋ~ㄹㅋㄱ 바터리자기짬토 투명드래곤았본비잇다
야 처차들을 이개이사 하누빠니리 실낵나가 함산의 지치 망"

Inference Result

- Step 7000

투명드래곤이 이버네요 향마신아코 본해서랄투명드래곤하고 자힐을었다

"케케케케"

바겹지가 적대시 쪼 꽂꽝

하지won...

나항 졸라 중남을 쳐참다

퀄여냐! 넌 니제 고치왕이로 날아조함네 꽂꽝교

"이전. 제계볼뽀태 이죄가꼬케"

꽈산히하들만 투명드래곤이...

께속

Try other corpus!

- Set `corpus_name` variable to 'nine_dreams' dataset.
 - Both train and inference scripts.
- Let's download any text data and try to train RNN!
- Larger corpus gives better result. But training time is much longer.

Inference Result - nine_dreams

- Epoch 99000

오늘 미연한 죽기고치를 면파지 못하고 마침께 수량궁처래마마)에
분행하니 부처님께서는 소유가 시녀 들어가 재주를
만나니 말을 건골이도, 양한림의 천재 다만의 암을 물함과 세 살이 술있다 남이
금강제의 슬픔을 떨치어 꽂고 진생을 따라가셨다.

"시비 크게 연화봉으로 돌아와 감화록 선비의 소설해지고 이 노후의 공중을 서서함이 도적을 치었다.
양한림이 칵을 수 일면서 의심하였나이다."

양원수가 들으려 다시 중에서는 내전하를 부적없이를 언제하여 계랑과 더불어 딸이 들와 돌아발히 여찌하
꾸짖어서 펴로 어렵게 여겨 공경에 지방 읊으셨으니 어찌(삼 되는 말이옵니까?"

이를 경공자들이 참은케 눌러져기 크리리오?"

하고, 이곳으로벼 생각하며 눈우면제신 1통해 배색가룰맘이 어찌 위남여 도리에서는
크게 궁기쁜 두 구성과 더불어 가처를 감추고
있었다.

두 시뱅과 정소저, 전교, 조정이 백릉 둘이 궁경에는 아생과 더불어
가하로자 생각하시기에 낭자가 이미 족자리로 보여 있는데 손을 잡시 명세코 별 한 손)을 듣고노며,
한림은 여중 골오는 사음에 침녀 무매로다 인사들이 대사께 돌아가도록 하라."

이 아가서는 새 돌아오니, 어찌 가히 변천학장, 해로다."

하는데, 전일에 용명을 튼 보해하며 가약을 이루거늘 말하기
대부인을 목안하셨던 것에 그 뮤은 끊어였다.

Thank You!

@youngjae yu

Special Thanks to: Byeongchang Kim, Jongwook Choi, Cesc Park, Sungjoon Choi
Slideshow created using [remark](#).