

RNNs (part 2)

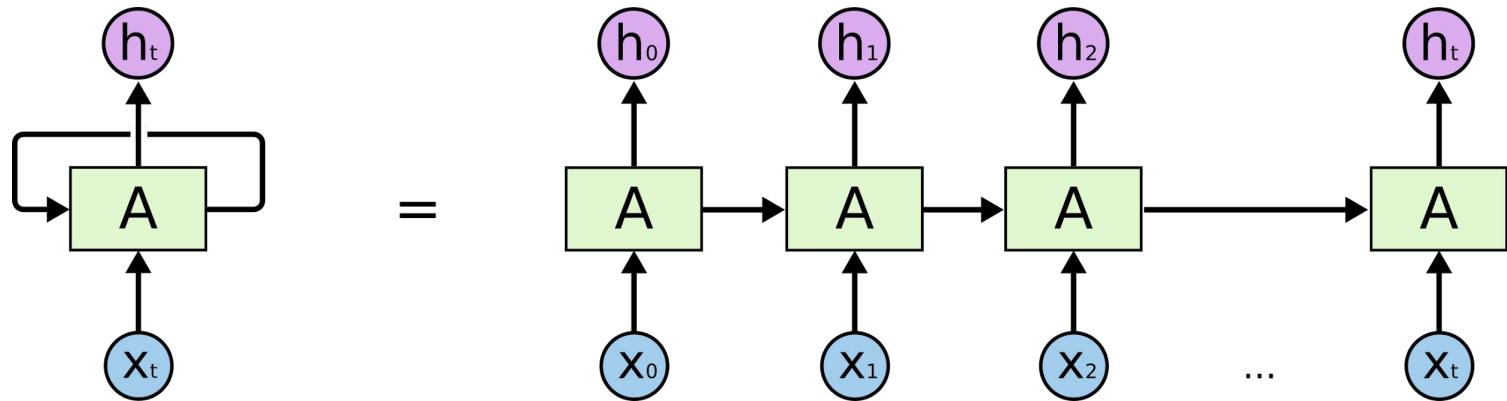
RNN

- Along with CNNs, RNNs are the most important class of models in DL
- Applicable in
speech
recognition(<https://research.googleblog.com/2015/09/google-voice-search-faster-and-more.html>)
machine
translation(<https://research.googleblog.com/2016/09/a-neural-network-for-machine.html>)
and many other task especially in NLP

Overview

- Basic RNN recap
- Language modelling example - character level RNN
- LSTM recap
- RNN extensions - bidirectional RNN, multilayer RNNs, RvNN
- Using RNNs with Tensorflow
- Case studies
- Quora Kaggle competition
- Lab instructions

Basic RNN



$$RNN : h_{t-1}, x_t \rightarrow h_t$$

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

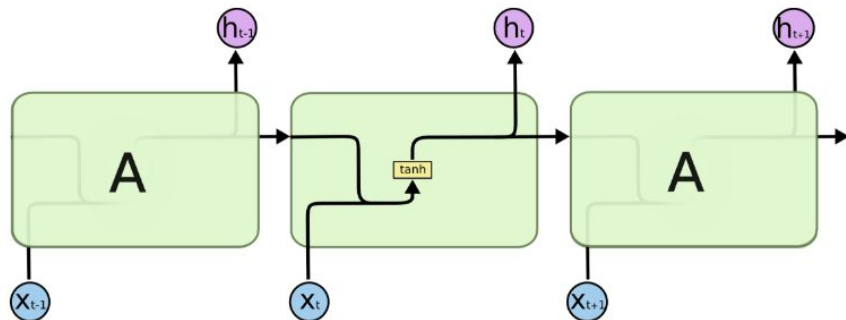
- W_h, W_x are the same across all timesteps

Basic RNN implementation

```
def build_model(self, x, steps_n, hidden_n, input_n):  
    # input_n dimension of the input at each timestep  
    # x has shape mb_size X steps_n X input_n  
    with tf.variable_scope('rnn'):  
        self.W = tf.Variable(name='W', dtype=tf.float32,  
                              initial_value=normal(scale=0.1, size=(input_n + hidden_n, hidden_n))  
        bias_value = np.zeros((hidden_n,), dtype='float32')  
        self.bias = tf.Variable(name='b', initial_value=bias_value, dtype=tf.float32)  
        self.h_0 = tf.Variable(name='h_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)  
  
    h = {}  
    h[-1] = tf.tile(self.h_0, [tf.shape(x)[0], 1])  
  
    for t in xrange(steps_n):  
        x_t = x[:, t, :]  
        input = tf.concat([x_t, h[t - 1]], axis=1)  
        h[t] = tf.tanh(tf.matmul(input, self.W) + self.bias)
```

$$RNN : h_{t-1}, x_t \rightarrow h_t$$

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$
$$\tanh\left(W \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b\right)$$

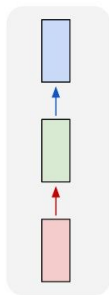


The repeating module in a standard RNN contains a single layer.

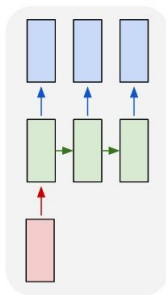
What to do with h_t s

- We have h_t for each t how do we use it?
- In “many to one” case we could just take h_{last} , and process it further to arrive at our final prediction
- In “many to many” case, we would have some other net take h_t and produce y_t , we would apply same net at each timestep
- other cases possible

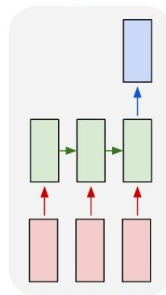
one to one



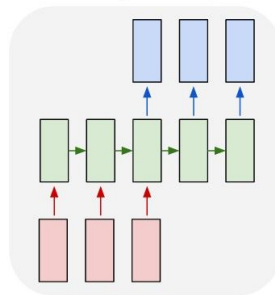
one to many



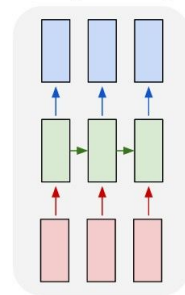
many to one



many to many



many to many



Basic RNN

There are few important technical issues with this code

- it requires `steps_n` to be constant, at the time of graph creation
- sometimes we would like different number of steps for each example in a minibatch
- what to do with very long inputs?

Language modelling

Train RNN to model

$$p(w_{n+1} | [w_1, w_2, w_3, \dots, w_n])$$

$$p(\text{"you"} | [\text{"I"}, \text{"like"}]) > p(\text{"like"} | [\text{"I"}, \text{"like"}])$$

- A lot of training data
- We can use such model to choose more probable sentences, ex. in speech recognition system, translation systems
- Or we could let it run and generate some text

Language modelling

- Give our RNN sequence of words and ask for the probability distribution of the next word

$$p(\text{"you"} | [\text{"I"}, \text{"like"}]) > p(\text{"like"} | [\text{"I"}, \text{"like"}])$$

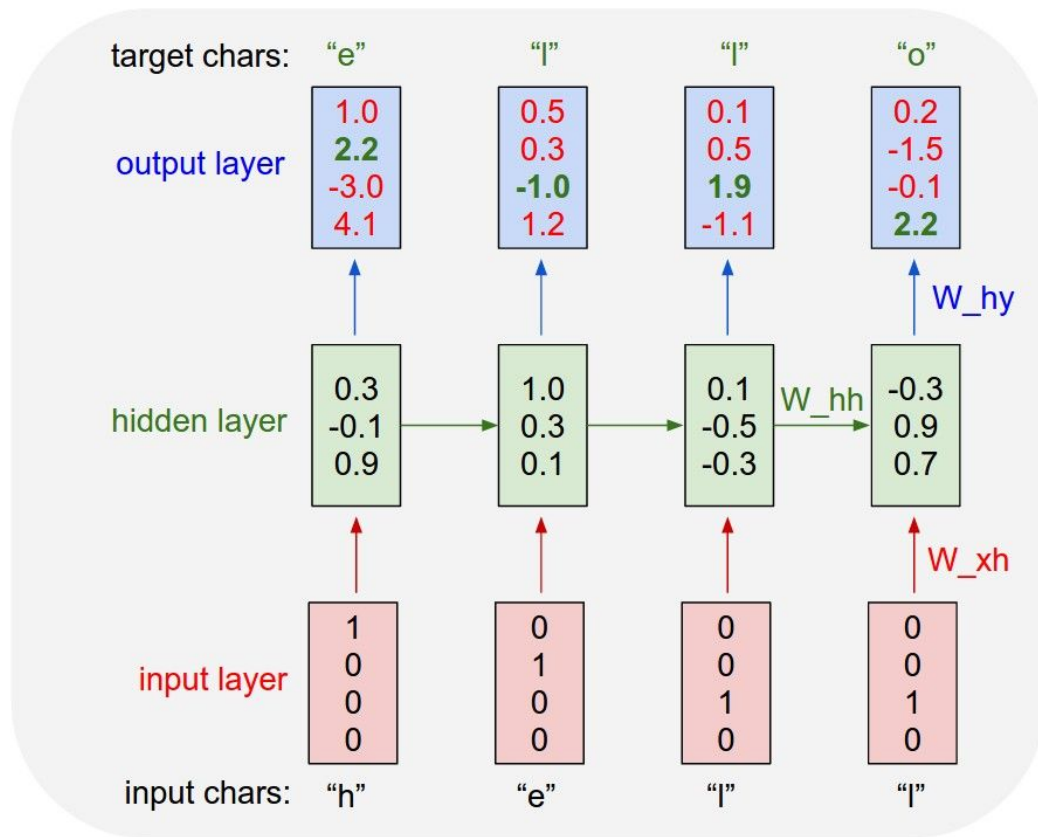
- Give our RNN sequence of characters and ask for the probability distribution of the next character

$$p(\text{"o"} | [\text{"h"}, \text{"e"}, \text{"l"}, \text{"l"}]) > p(\text{"w"} | [\text{"h"}, \text{"e"}, \text{"l"}, \text{"l"}])$$

Language modelling char rnn

$$RNN : h_{t-1}, x_t \rightarrow h_t$$
$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

Training
sequence:
"hello"



hidden state
size = 3

min-char-rnn.py gist: 112 lines of Python

```
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) }
14
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 wnh = 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
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 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(wnh, 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
44     # backward pass: compute gradients going backwards
45     dxnh, dwhh, dwhy = np.zeros_like(wnh), np.zeros_like(whh), np.zeros_like(why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnxt = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(ps[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dwhy += np.dot(dy, hs[t].T)
52         dby += dy
53         dh = np.dot(why.T, dy) + dhnxt # backprop into h
54         dhrw = (1 - hs[t]**2) * hs[t]**2 * dh # backprop through tanh nonlinearity
55         dbh += dhrw
56         dwhh += np.dot(dhrw, xs[t].T)
57         dxnh += np.dot(dhrw, hs[t-1].T)
58         dhnxt = np.dot(whh.T, dhrw)
59     for dparam in [dxnh, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dxnh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

```
62
63 def sample(h, seed_ix, n):
64     """
65     sample a sequence of integers from the model
66     h is memory state, seed_ix is seed letter for first time step
67     """
68     x = np.zeros((vocab_size, 1))
69     x[seed_ix] = 1
70     ixes = []
71     for t in xrange(n):
72         h = np.tanh(np.dot(wnh, x) + np.dot(whh, h) + bh)
73         y = np.dot(why, h) + by
74         p = np.exp(y) / np.sum(np.exp(y))
75         ix = np.random.choice(range(vocab_size), p=p.ravel())
76         x = np.zeros((vocab_size, 1))
77         x[ix] = 1
78         ixes.append(ix)
79     return ixes
80
81 n, p = 0, 0
82 mnh, mwhh, mwhy = np.zeros_like(wnh), 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 >= 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 '---->%s\n' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dxnh, dwhh, dwhy, dbh, dby, 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([wnh, whh, why, bh, by],
106                                   [dxnh, dwhh, dwhy, dbh, dby],
107                                   [mnh, 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
```

(<https://gist.github.com/karpathy/d4dee566867f8291f086>)

 *initial character, level variable MM model. written by Andrei Kozlov (kkozlov@)

```
800 License
***
support numpy as np

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

```

1  """
2  Minimal character-level Vanilla RNN model. Written by Andre Karpathy (@karpathy)
3  800 Lines
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_idx = { c:i for i, c in enumerate(chars) }
13 idx_to_char = { i:c for i,c in enumerate(chars) }

```

```

10 # Hyperparameters
11 hidden_size = 100 # size of hidden layer of neurons
12 seq_length = 25 # number of steps to unroll the RNN for
13 learning_rate = 0e-1
14
15 # model parameters
16 w1 = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
17 w2 = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
18 w3 = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
19 b1 = np.zeros((hidden_size, 1)) # hidden bias
20 b2 = np.zeros((hidden_size, 1)) # output bias

```

```
27 def LossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers
30     """
```

```

#prev is 4x1 array of initial hidden state
#returns the loss, gradients on model parameters, and last hidden state
'''
'''

```

```
24  Wa[-1] = np.copy(Wprev)
25  loss = 0
26  # Forward pass
```

```

38     xs[t] = np.zeros([vocab_size, 1]) + wordvec in 1-of-K representation
39     xs[t][inputs[t]] = 1
40     hs[t] = np.tanh(np.dot(wmh, xs[t]) + np.dot(bmh, hs[t-1]) + bh) + hidden state

```

```

ps[i] = np.exp(logp[i]) / np.sum(np.exp(logp[i])) # probabilities for next state
loss += -np.log(ps[i][targets[i],0]) # softmax (cross-entropy) loss

```

```

20  xoh, xoh, wh, = np.zeros_like(xoh), np.zeros_like(wh), np.zeros_like(wh)
21  yb, dy = np.zeros_like(yb), np.zeros_like(dy)
22  shwexi = np.zeros_like(ha[0])

```

```

49     dy = np.copy(dy[t])
50     dy[target(t)] += 1 + backward_loss * 2
51     ddy = np.dot(dy, fn[t].T)

```

```

13  dh = np.dot(why.T, dy) + dnext + backward_lstm_b
14  draw = (1 - ha[t] * ha[t]) * dh + backward_through_tanh_collaterally
15  dhi += draw

```

```

97     ddrh += np.dot(drrhs, hs[t-1].T)
98     drrhs = np.dot(ddrh.T, drrhs)
99     for drrhs in ldrhs, rddh, drrhs, ddrh, ddrh:

```

```

    np.clip(dbores, -0.5, 0.5, out=dbores) # clip to mitigate exploding gradients
    return loss, dwh, ddb, dhy, dbh, dbx, ns[ns==ispsuts-1]
def sample(x, seed_ix, n):
    """
    """

```

```

sample a sequence of integers from the well
// h is memory state, seed_ix is seed letter for first time step
//
// x = an exponentially sized list

```

```

x[ans, ix] = 1
ans = []
for i in xrange(n):
    b = np.argmax(np.dot(w0f, x) + np.dot(w0f, b) + b0f)

```

```

p = np.exp(wy) / np.sum(np.exp(y))
ix = np.random.choice(range(vocab_size), p=p.ravel())
x = np.zeros([vocab_size, 1])

```

```

11     lres.append(lr)
12     return lres
13
14

```

```
07 msh, mshx, msy = np.zeros_like(msh), np.zeros_like(msh), np.zeros_like(msy)
08 msh, msy = np.zeros_like(msh), np.zeros_like(msy) # memory variables for Adagrad
09 smooth_loss = -np.log(1./vocab_size**seq_length) / loss at iteration 0
```

```

    # prepare seqs: we're working from left to right so seqs seq_lengths [seq]
    if p==seq_lengths+1 == len(data) or x == 0:
        #prev = up Jones (hidden_size,1) # reset this memory
        x = 0 # on from start of data

```

```
targets = [char_to_ix[ch] for ch in data[p:p+seq_length+1]]
```

```

99     sample_ix = sample(fargs, inputs[i], 200)
100     txt = ''.join(is.to_char[ix] for ix in sample_ix)
101     print '----- %s %s ----' % (txt, i)

```

```

100 loss, dash, ddln, dely, ddu, ddy, kprev = DesFuncInputs, targets, kprev)
101 smooth_loss = smooth_loss * 0.999 + loss * 0.001

```

```

534     a perform parameter update with nograd
535     for param, dparam, new in zip(w0s, w0s, w0s):
536         (data, data_grad, data_grad2)

```

```

100     param += learning_rate * dparam / np.sqrt(param + 1e-8) * integral_update
101

```

Initializations

```
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
```

```
20 # model parameters
```

```
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
```

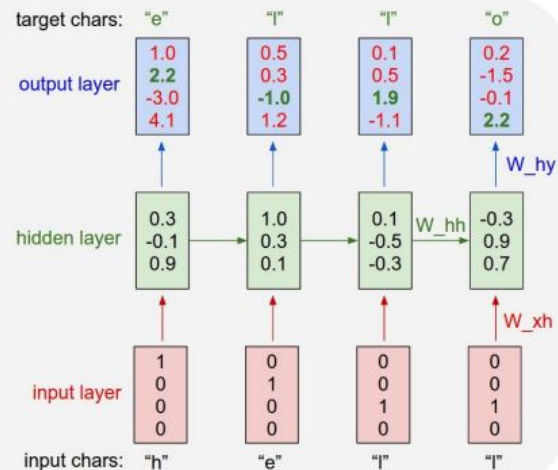
```
22 W_hh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
```

```
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
```

```
24 bh = np.zeros((hidden_size, 1)) # hidden bias
```

```
25  bv = np.zeros((vocab_size, 1)) # output bias
```

recall:



[illegible]

```

n, p = 0, 0
82 mwxx, mwxx, mwxx = np.zeros_like(Wxx), np.zeros_like(Wxx), np.zeros_like(Wxx)
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, dwxx, dwxx, dwxx, dbh, dby, 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([Wxx, Wxx, Wxx, bh, by],
106                                   [dwxx, dwxx, dwxx, dbh, dby],
107                                   [mwxx, mwxx, mwxx, 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

```

```

10  num, den, mlen = np.zeros(100000), np.zeros(100000), np.zeros(100000)
11  num, den = np.zeros(100000), np.zeros(100000) # memory variables for integral
12  mlen, sum = np.logit, np.zeros(100000, length) # sum as accumulator
13  while True:
14      # extract inputs for computing from left to right in num, length [len]
15      if num[length] == 0: break # if 0, stop
16      num = np.zeros(100000, 100000) # reset num memory
17      if 0 in den: den = den + 1
18      # get n from input
19      input = input[0, 100000] # extract num, length[]
20      input = input[100000, 100000] # input = [den, 100000, length]
21      # sample from the model num and den
22      if 0 in num: break
23      sample = np.random.choice(100000, 1000)
24      num = np.zeros(100000, 1000) # for n to sample, 1000
25      num += num * input[0, 1000] + 1 * input[1, 1000]
26      # forward pass through the model and forward gradients
27      num, den, mlen, den, den, num = sample[nums, targets, num]
28      num, den = num * len, den * len
29      if 0 in num or 0 in den: print "num, den, len = 0", num, den, len
30
31 # iterative backward passes with AD-grad
32 for num, den, num, den in num, den, num, den, den, den:
33     [num, den, mlen, den, den, num]
34     [num, den, mlen, den, den, num]
35
36 num = num * 0.999999 # gradient
37 den = den * 0.999999 # gradient
38 num = num * 0.999999 # gradient
39 den = den * 0.999999 # gradient
40 num = num * 0.999999 # gradient
41 den = den * 0.999999 # gradient

```

```

n, p = 0, 0
mWxh, mWwh, mWhy = np.zeros_like(Wxh), np.zeros_like(Wwh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
    # prepare inputs (we're sweeping from left to right in steps seq_length long)
    if p+seq_length+1 >= len(data) or n == 0:
        hprev = np.zeros((hidden_size,1)) # reset RNN memory
        p = 0 # go from start of data
    inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
    targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

    # sample from the model now and then
    if n % 100 == 0:
        sample_ix = sample(hprev, inputs[0], 200)
        txt = ''.join(ix_to_char[ix] for ix in sample_ix)
        print '----\n %s \n----' % (txt, )

    # forward seq_length characters through the net and fetch gradient
    loss, dWxh, dWwh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + loss * 0.001
    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

    # perform parameter update with Adagrad
    for param, dparam, mem in zip([Wxh, Wwh, Why, bh, by],
                                   [dWxh, dWwh, dWhy, dbh, dby],
                                   [mWxh, mWwh, mWhy, mbh, mby]):
        mem += dparam * dparam
        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

    p += seq_length # move data pointer
    n += 1 # iteration counter

```

[illegible]

```

Main loop

like(Wxh), np.zeros_1
np.zeros_like(by) #
ab_size)*seq_length

eeping from left to r
ta) or n == 0:
size,1)) # reset RNN
data
r ch in data[p:p+seq
or ch in data[p+1:p+

and then

inputs[0], 200)
ix] for ix in sample
% (txt, )

ters through the net
, dby, hprev = lossF
0.999 + loss * 0.00
r %d, loss: %f' % (n

with Adagrad
ip([Wxh, Whh, Why, b
[dwxh, dwhh, dwhy
[mwxh, mw hh, mwhy

dparam / np.sqrt(me

pointer

```

Main loop

```

81 n, p = 0, 0
82 mwxx, mwxx, mwxx = np.zeros_like(Wxx), np.zeros_like(Wxx), np.zeros_like(Wxx)
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, dwxx, dwxx, dwxx, dbh, dby, 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([Wxx, Wxx, Wxx, bh, by],
106                                   [dwxx, dwxx, dwxx, dbh, dby],
107                                   [mwxx, mwxx, mwxx, 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

```



```

10  def __init__(self, n):
11      self.n = n
12      self.m = 0
13      self.m = 0
14      self.m = 0
15      self.m = 0
16      self.m = 0
17      self.m = 0
18      self.m = 0
19      self.m = 0
20      self.m = 0
21      self.m = 0
22      self.m = 0
23      self.m = 0
24      self.m = 0
25      self.m = 0
26      self.m = 0
27      self.m = 0
28      self.m = 0
29      self.m = 0
30      self.m = 0
31      self.m = 0
32      self.m = 0
33      self.m = 0
34      self.m = 0
35      self.m = 0
36      self.m = 0
37      self.m = 0
38      self.m = 0
39      self.m = 0
40      self.m = 0
41      self.m = 0
42      self.m = 0
43      self.m = 0
44      self.m = 0
45      self.m = 0
46      self.m = 0
47      self.m = 0
48      self.m = 0
49      self.m = 0
50      self.m = 0
51      self.m = 0
52      self.m = 0
53      self.m = 0
54      self.m = 0
55      self.m = 0
56      self.m = 0
57      self.m = 0
58      self.m = 0
59      self.m = 0
60      self.m = 0
61      self.m = 0
62      self.m = 0
63      self.m = 0
64      self.m = 0
65      self.m = 0
66      self.m = 0
67      self.m = 0
68      self.m = 0
69      self.m = 0
70      self.m = 0
71      self.m = 0
72      self.m = 0
73      self.m = 0
74      self.m = 0
75      self.m = 0
76      self.m = 0
77      self.m = 0
78      self.m = 0
79      self.m = 0
80      self.m = 0
81      self.m = 0
82      self.m = 0
83      self.m = 0
84      self.m = 0
85      self.m = 0
86      self.m = 0
87      self.m = 0
88      self.m = 0
89      self.m = 0
90      self.m = 0
91      self.m = 0
92      self.m = 0
93      self.m = 0
94      self.m = 0
95      self.m = 0
96      self.m = 0
97      self.m = 0
98      self.m = 0
99      self.m = 0
100     self.m = 0

```

```

81 n, p = 0, 0
82 mWxh, mWwh, mWhy = np.zeros_like(Wxh), np.zeros_like(Wwh), 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, dWwh, dWhy, dbh, dby, 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, Wwh, Why, bh, by],
106                                   [dWxh, dWwh, dWhy, dbh, dby],
107                                   [mWxh, mWwh, 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

```

[illegible]

```

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, dby, 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, dby],
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

```

min-char-rnn.py gist

```

1 """
2 Minimal character-level vanilla rnn model. written by Andrej Karpathy (dhrupaty)
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) }
14
15 # Hyperparameters
16 hidden_size = 200 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the rnn for
18 learning_rate = 0.01
19
20 # Model parameters
21 wnh = 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
26
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         xi[i] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
39         xi[inputs[t]] = 1
40         hi[i] = np.tanh(np.dot(wnh, xi[i]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         yi[i] = np.dot(why, hi[i]) + by # unnormalized log probabilities for next char
42         ps[i] = np.exp(yi[i]) / np.sum(np.exp(yi[i])) # probabilities for next char
43         loss += -np.log(ps[i][targets[t]]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     dwhx, dwhy, dwhy = np.zeros_like(wnh), np.zeros_like(whh), np.zeros_like(why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnext = np.zeros_like(hs[-1])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.zeros_like(yi[t])
50         dy[targets[t]] = 1 - ps[t][targets[t]] # backprop into y
51         dby += dy
52         dh = np.dot(why.T, dy) + dhnext # backprop into h
53         dhrdw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
54         dbh += dhrdw
55         dwxh += np.dot(dhrdw, xi[t].T)
56         dwhh += np.dot(dhrdw, hs[t-1].T)
57         dhnext = np.dot(whh.T, dhrdw)
58     for dparam in [dwhx, dwhh, dwhy, dbh, dby]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dwhx, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
61

```

Loss function

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

```

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)
44     # backward pass: compute gradients going backwards
45     dwhx, dwhh, dwhy = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnext = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(ps[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dwhy += np.dot(dy, hs[t].T)
52         dby += dy
53         dh = np.dot(why.T, dy) + dhnext # backprop into h
54         dhrdw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += dhrdw
56         dwxh += np.dot(dhrdw, xs[t].T)
57         dwhh += np.dot(dhrdw, hs[t-1].T)
58         dhnext = np.dot(whh.T, dhrdw)
59     for dparam in [dwhx, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwhx, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]

```


[illegible]

```
def convolve(inputs, targets, Numpy):
    """
    Inputs, targets are back lists of integers.
    Numpy is list array of initial hidden states
    Returns the loss, gradients on model parameters, and last hidden state"""
    ex, hy, ys, ps = [], [], [], []
    hid[-1] = Numpy.zeros(targets)
    for i in range(1, len(inputs)):
        x = inputs[i]
        y = targets[i]
        h = Numpy.dot(hid[-1], W) + b
        h = sigmoid(h)
        p = Numpy.dot(h, Wout) + bout
        ex.append(x)
        hy.append(y)
        ps.append(p)
```

[illegible]

```

10  sample(1:n, size=k, replace=T)
11  cat
12  creates a sequence of integers from the number
13  1 to n. In memory space, word=k is used letter for First time size
14  word=k
15  n = n + n*random(1:n, size, 1)
16  n = n + n*random(1:n, size, 1)
17  for i in 1:n
18  for i in 1:n
19  n = n + n*random(1:n, size, 1)
20  n = n + n*random(1:n, size, 1)
21  n = n + n*random(1:n, size, 1)
22  n = n + n*random(1:n, size, 1)
23  n = n + n*random(1:n, size, 1)
24  n = n + n*random(1:n, size, 1)
25  n = n + n*random(1:n, size, 1)
26  n = n + n*random(1:n, size, 1)
27  n = n + n*random(1:n, size, 1)
28  n = n + n*random(1:n, size, 1)
29  n = n + n*random(1:n, size, 1)
30  n = n + n*random(1:n, size, 1)
31  n = n + n*random(1:n, size, 1)
32  n = n + n*random(1:n, size, 1)
33  n = n + n*random(1:n, size, 1)
34  n = n + n*random(1:n, size, 1)
35  n = n + n*random(1:n, size, 1)
36  n = n + n*random(1:n, size, 1)
37  n = n + n*random(1:n, size, 1)
38  n = n + n*random(1:n, size, 1)
39  n = n + n*random(1:n, size, 1)
40  n = n + n*random(1:n, size, 1)
41  n = n + n*random(1:n, size, 1)
42  n = n + n*random(1:n, size, 1)
43  n = n + n*random(1:n, size, 1)
44  n = n + n*random(1:n, size, 1)
45  n = n + n*random(1:n, size, 1)
46  n = n + n*random(1:n, size, 1)
47  n = n + n*random(1:n, size, 1)
48  n = n + n*random(1:n, size, 1)
49  n = n + n*random(1:n, size, 1)
50  n = n + n*random(1:n, size, 1)
51  n = n + n*random(1:n, size, 1)
52  n = n + n*random(1:n, size, 1)
53  n = n + n*random(1:n, size, 1)
54  n = n + n*random(1:n, size, 1)
55  n = n + n*random(1:n, size, 1)
56  n = n + n*random(1:n, size, 1)
57  n = n + n*random(1:n, size, 1)
58  n = n + n*random(1:n, size, 1)
59  n = n + n*random(1:n, size, 1)
60  n = n + n*random(1:n, size, 1)
61  n = n + n*random(1:n, size, 1)
62  n = n + n*random(1:n, size, 1)
63  n = n + n*random(1:n, size, 1)
64  n = n + n*random(1:n, size, 1)
65  n = n + n*random(1:n, size, 1)
66  n = n + n*random(1:n, size, 1)
67  n = n + n*random(1:n, size, 1)
68  n = n + n*random(1:n, size, 1)
69  n = n + n*random(1:n, size, 1)
70  n = n + n*random(1:n, size, 1)
71  n = n + n*random(1:n, size, 1)
72  n = n + n*random(1:n, size, 1)
73  n = n + n*random(1:n, size, 1)
74  n = n + n*random(1:n, size, 1)
75  n = n + n*random(1:n, size, 1)
76  n = n + n*random(1:n, size, 1)
77  n = n + n*random(1:n, size, 1)
78  n = n + n*random(1:n, size, 1)
79  n = n + n*random(1:n, size, 1)
80  n = n + n*random(1:n, size, 1)
81  n = n + n*random(1:n, size, 1)
82  n = n + n*random(1:n, size, 1)
83  n = n + n*random(1:n, size, 1)
84  n = n + n*random(1:n, size, 1)
85  n = n + n*random(1:n, size, 1)
86  n = n + n*random(1:n, size, 1)
87  n = n + n*random(1:n, size, 1)
88  n = n + n*random(1:n, size, 1)
89  n = n + n*random(1:n, size, 1)
90  n = n + n*random(1:n, size, 1)
91  n = n + n*random(1:n, size, 1)
92  n = n + n*random(1:n, size, 1)
93  n = n + n*random(1:n, size, 1)
94  n = n + n*random(1:n, size, 1)
95  n = n + n*random(1:n, size, 1)
96  n = n + n*random(1:n, size, 1)
97  n = n + n*random(1:n, size, 1)
98  n = n + n*random(1:n, size, 1)
99  n = n + n*random(1:n, size, 1)
100 n = n + n*random(1:n, size, 1)

```

```

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)

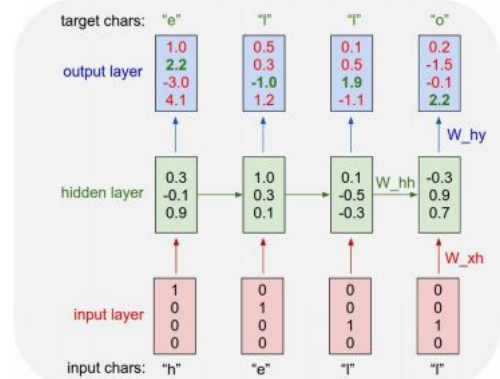
```

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

Softmax classifier

[illegible][illegible]

recall:



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

[illegible]

```

63 def sample(h, seed_ix, n):
64     """
65     sample a sequence of integers from the model
66     h is memory state, seed_ix is seed letter for first time step
67     """
68     x = np.zeros((vocab_size, 1))
69     x[seed_ix] = 1
70     ixes = []
71     for t in xrange(n):
72         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
73         y = np.dot(Why, h) + by
74         p = np.exp(y) / np.sum(np.exp(y))
75         ix = np.random.choice(range(vocab_size), p=p.ravel())
76         x = np.zeros((vocab_size, 1))
77         x[ix] = 1
78         ixes.append(ix)
79     return ixes

```

at first:

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

↓
train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuw y 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 offer.

↓
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.

How to improve this code?

- GPU
- Add minibatches
- LSTM
- More powerful model(multilayer LSTM)
- Regularization (Recurrent Neural Network Regularization, <https://arxiv.org/abs/1409.2329>)
- <https://github.com/karpathy/char-rnn>

Recap architectures LSTM - Long short term memory

LSTM: $\mathcal{X}_t, h_{t-1}, c_{t-1} \rightarrow h_t, c_t$

$$i = \text{sigm}(W_i \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_i)$$

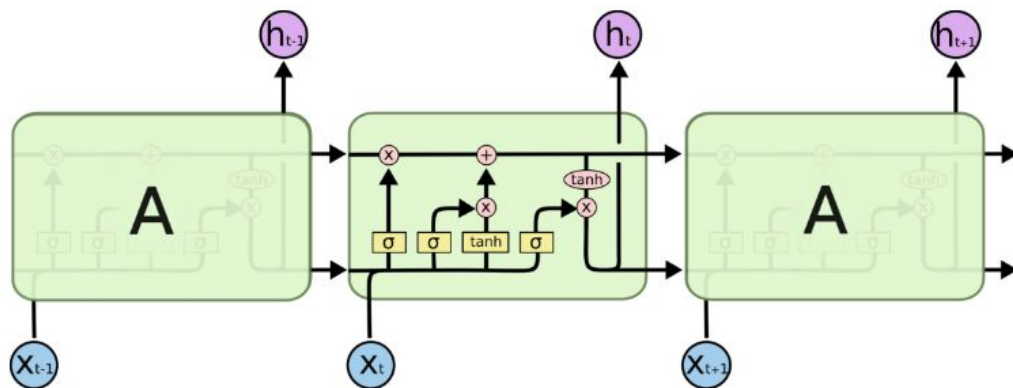
$$f = \text{sigm}(W_f \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_f)$$

$$o = \text{sigm}(W_o \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_o)$$

$$g = \tanh(W_g \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_g)$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$



The repeating module in an LSTM contains four interacting layers.

LSTM implementation

```
def build_model(self, x, steps_n, hidden_n, input_n, forget_bias=1.0):  
    # input_n dimension of the input at each timestep  
    # x has shape mb_size X steps_n X input_n  
    with tf.variable_scope('lstm'):  
        self.W = tf.Variable(name='W', dtype=tf.float32,  
                              initial_value=normal(scale=0.1, size=(input_n + hidden_n, 4 * hidden_n)))  
        bias_value = np.zeros((4 * hidden_n,), dtype='float32')  
        bias_value[1 * hidden_n: 2 * hidden_n] += forget_bias  
  
        self.bias = tf.Variable(name='b', initial_value=bias_value, dtype=tf.float32)  
  
        self.c_0 = tf.Variable(name='c_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)  
        self.h_0 = tf.Variable(name='h_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)
```

LSTM implementation

```
c, h = {}, {}  
c[-1] = tf.tile(self.c_0, [tf.shape(x)[0], 1])  
h[-1] = tf.tile(self.h_0, [tf.shape(x)[0], 1])  
  
for t in xrange(steps_n):  
    x_t = x[:, t, :]  
    input = tf.concat([x_t, h[t - 1]], axis=1)  
  
    z = tf.matmul(input, self.W) + self.bias  
    i = tf.sigmoid(z[:, 0 * hidden_n: 1 * hidden_n])  
    f = tf.sigmoid(z[:, 1 * hidden_n: 2 * hidden_n])  
    o = tf.sigmoid(z[:, 2 * hidden_n: 3 * hidden_n])  
    g = tf.tanh(z[:, 3 * hidden_n: 4 * hidden_n])  
  
    c[t] = f * c[t - 1] + i * g  
    h[t] = o * tf.tanh(c[t])
```

LSTM : $\mathcal{X}_t, h_{t-1}, c_{t-1} \rightarrow h_t, c_t$

$$i = \text{sigm}(W_i \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_i)$$

$$f = \text{sigm}(W_f \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_f)$$

$$o = \text{sigm}(W_o \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_o)$$

$$g = \text{tanh}(W_g \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_g)$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

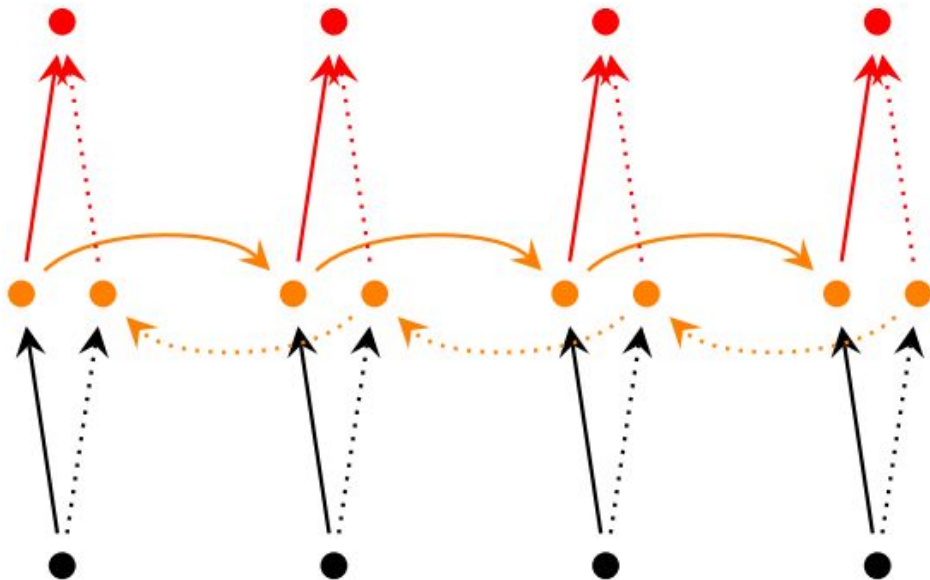
Bidirectional RNN

- Nothing special about them
- Just run one RNN from left to right
- The other one from right to left
- Concatenate their respective hidden states

$$h_{bidirect}[t] = \text{concat}(h_{forward}[t], h_{backward}[t])$$

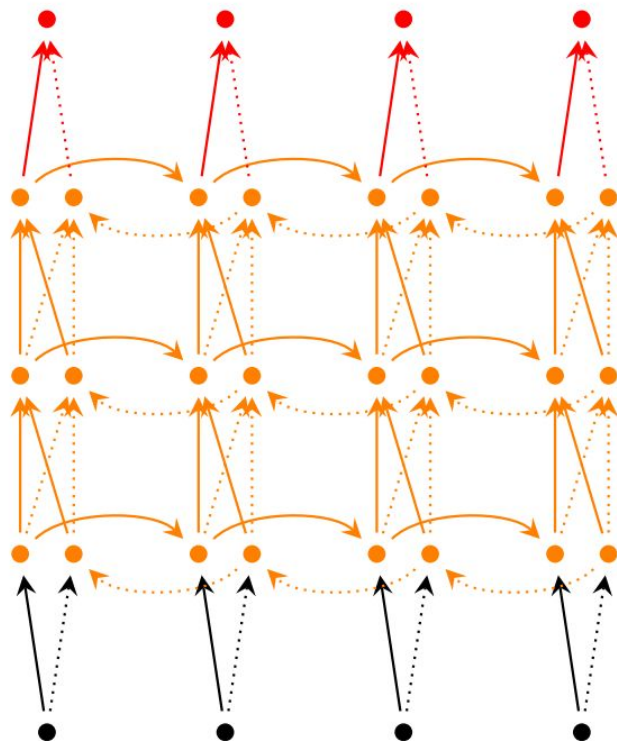
- Or just take
$$h = \text{concat}(h_{forward}[T], h_{backward}[0])$$

when doing classification
- Used when you want h_t to depend not only on the “past”, but also the “future”



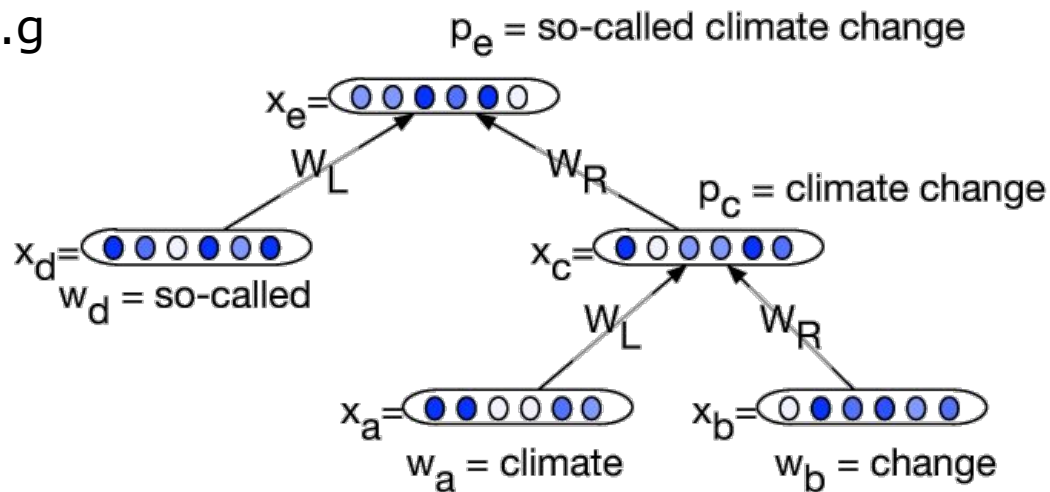
MultiLayer RNN

- Just run one RNN on the outputs of another one
- We can combine this with bidirectional RNN and make Deep Bidirectional RNN
- Or Deep Bidirectional LSTM



Recursive neural networks

- Works on a tree structure, e.g. suppose you want to do sentiment analysis and have parse tree of the sentence
- A RNN is just a special case of RvNN, a linear tree



Tensorflow RNN API

Using RNNs/LSTMs is very easy using tensorflow.

```
# Unstack to get a list of 'n_steps' tensors of shape (batch_size, n_input)  
x = tf.unstack(x, n_steps, 1)
```

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

```
# output, state = lstm_cell(x_t, state)
```

```
# Get lstm cell output  
outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)
```

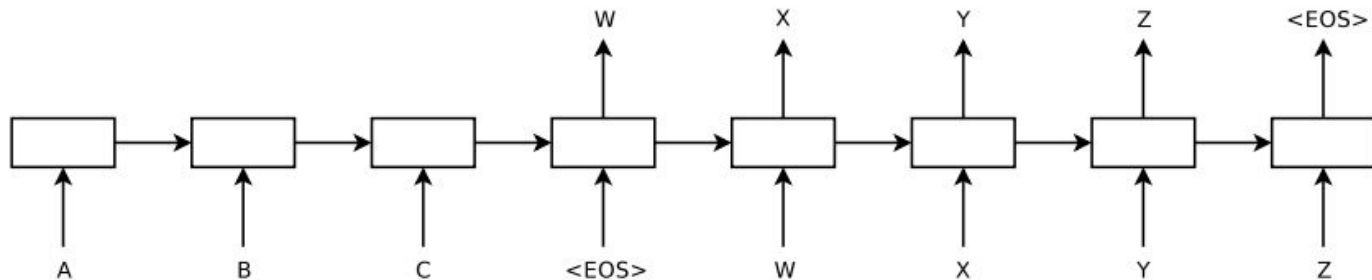
<https://www.tensorflow.org/tutorials/recurrent>

Case studies

- seq2seq
- Machine translation with attention
- Caption generation
- Neural Turing Machine

seq2seq

- We want to translate from English to French
- WMT'14 dataset, 12M sentences consisting of 348M French words and 304M English words
- Basic idea: train some RNN to take source sentence one word at a time(encoder), and other RNN to take this encoded state and produce one word of translation at a time.



Sequence to Sequence Learning with Neural Networks,
Ilya Sutskever et al.

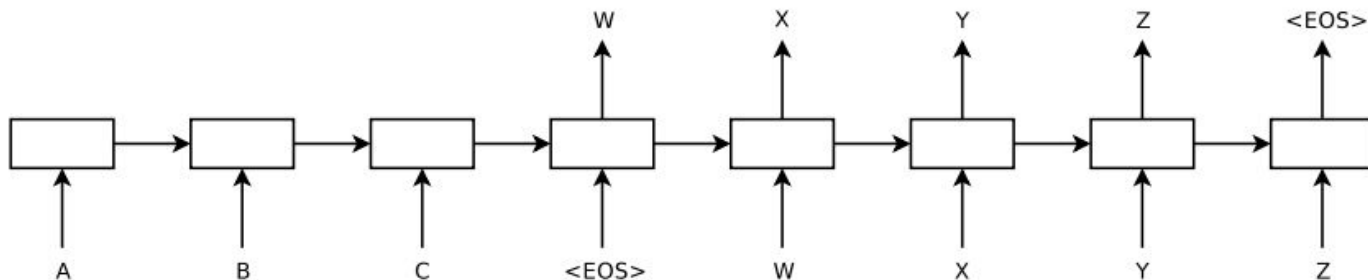
<https://arxiv.org/abs/1409.3215>

How to treat words? - Word embeddings

- We give each word in a vocabulary a d -dimensional real vector
- We learn those vectors jointly with our whole model
- The word embedding of a word can learn to represent important “things” about this word, e.g. if this is verb/noun?, is it positive/negative? and many other things that are not easily described in words
- Often we initialize these word embedding with some precomputed vectors, trained in unsupervised manner, like word2vec
- 1000 dimensional word embeddings used in this paper

seq2seq model details

- Words embeddings at input each timestep
- Two different LSTMs, one for encoding the source sentence and one for decoding
- “we found it extremely valuable to reverse the order of the words of the input sentence”
- “we found that deep LSTMs significantly outperformed shallow LSTMs, so we chose an LSTM with four layers”
- Beam search for selecting best translation
- Training was not easy, had to split the model to many gpus
- Trained on 8 GPU, 4 of them just for computing softmax



seq2seq

WMT '14 English-to-French translation task

Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	34.81

Table 1: The performance of the LSTM on WMT'14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

- Did not beat state of the art in 2014
- But in 2016 DL base methods dominated the WMT 2016

Method	test BLEU score (ntst14)
Baseline System [29]	33.30
Cho et al. [5]	34.54
State of the art [9]	37.0
Rescoring the baseline 1000-best with a single forward LSTM	35.61
Rescoring the baseline 1000-best with a single reversed LSTM	35.85
Rescoring the baseline 1000-best with an ensemble of 5 reversed LSTMs	36.5
Oracle Rescoring of the Baseline 1000-best lists	~45

Table 2: Methods that use neural networks together with an SMT system on the WMT'14 English to French test set (ntst14).

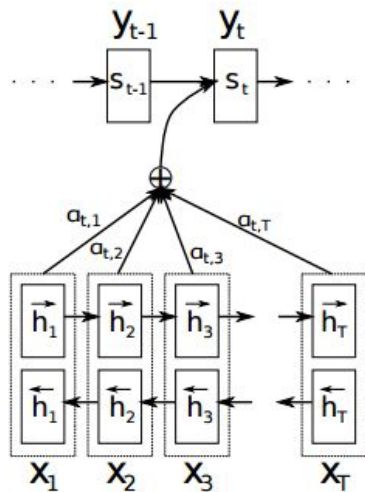
Attention in neural networks

- focusing on part of the information
- e.g. we will see, RNN attending to certain parts of other RNN's output
- Andrej Karpathy: *The concept of **attention** is the most interesting recent architectural innovation in neural networks.*

Machine translation with attention

- Same task: machine translation.
- High level idea:
In seq2seq encoder RNN encoded the source task into some fixed dimensional hidden vector.
Then the decoder RNN decoded this state one word at a time.
- Now we want the RNN outputting the words of translation to be able to attend to parts of the source sentence it “thinks” are useful for generating the output word.

Machine translation with attention



$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

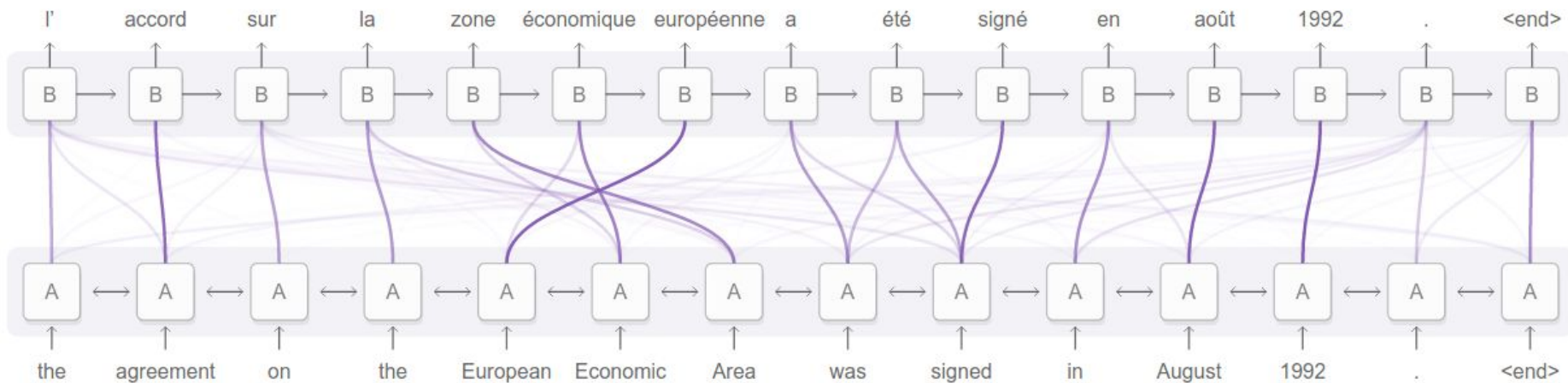
$$h_j = \left[\vec{h}_j^\top; \overleftarrow{h}_j^\top \right]^\top$$

- Use bidirectional LSTM to compute hidden states h_j
- When generating translation at each timestep i compute attention score e_{ij}
- c_i is the context generated by attention mechanism. Use this context, with current state in LSTM to generate output
- The attention used here is soft, the whole model is differentiable
-

Neural machine translation by jointly learning to align and translate.

<https://arxiv.org/abs/1409.0473>

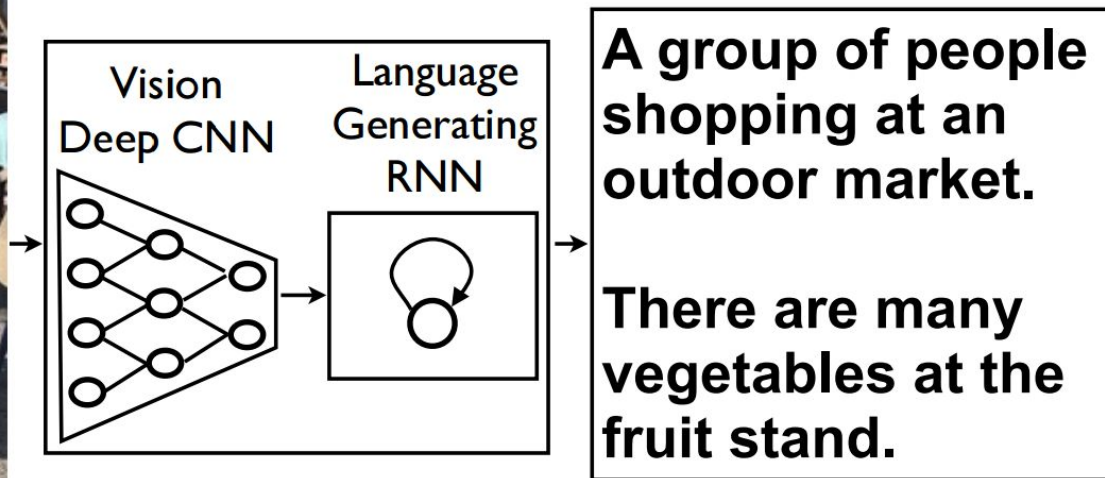
Machine translation with attention



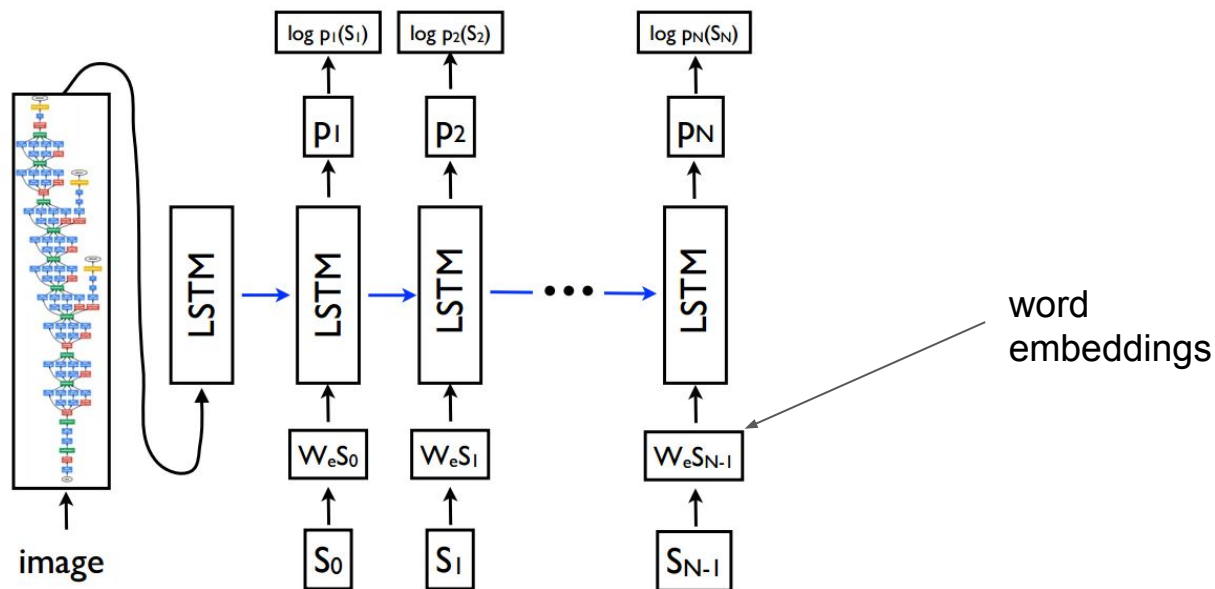
Neural machine translation by jointly learning to align and translate.

<https://arxiv.org/abs/1409.0473>

Caption generation



Caption generation



Show, tell

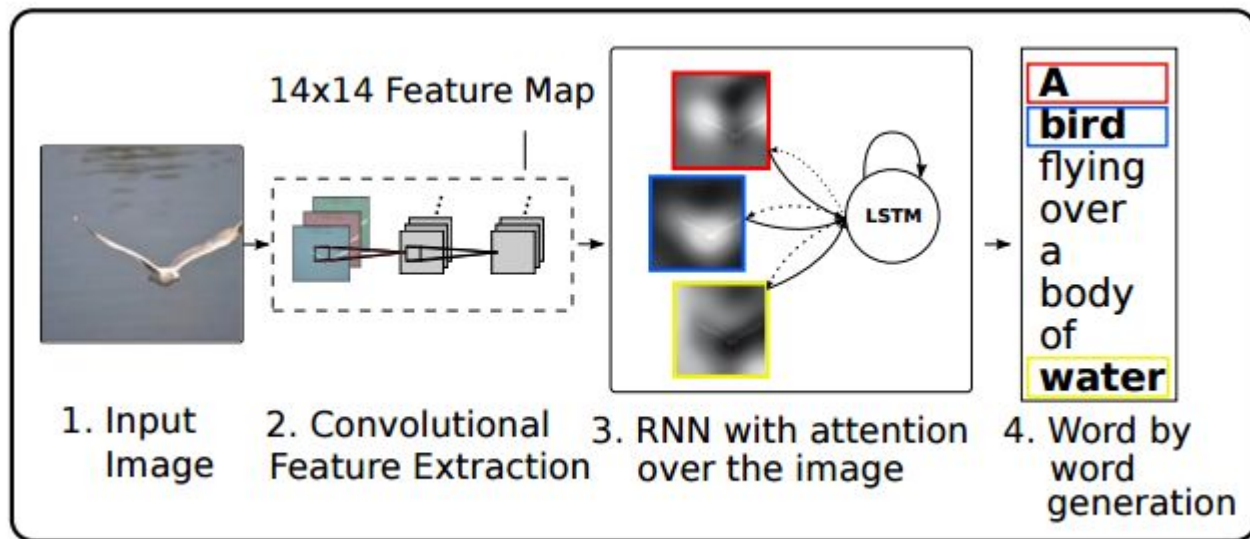
Not easy to automatically evaluate the model



Figure 5. A selection of evaluation results, grouped by human rating.

Show, attend, tell

The extractor produces L vectors, each of which is a D -dimensional representation corresponding to a part of the image.

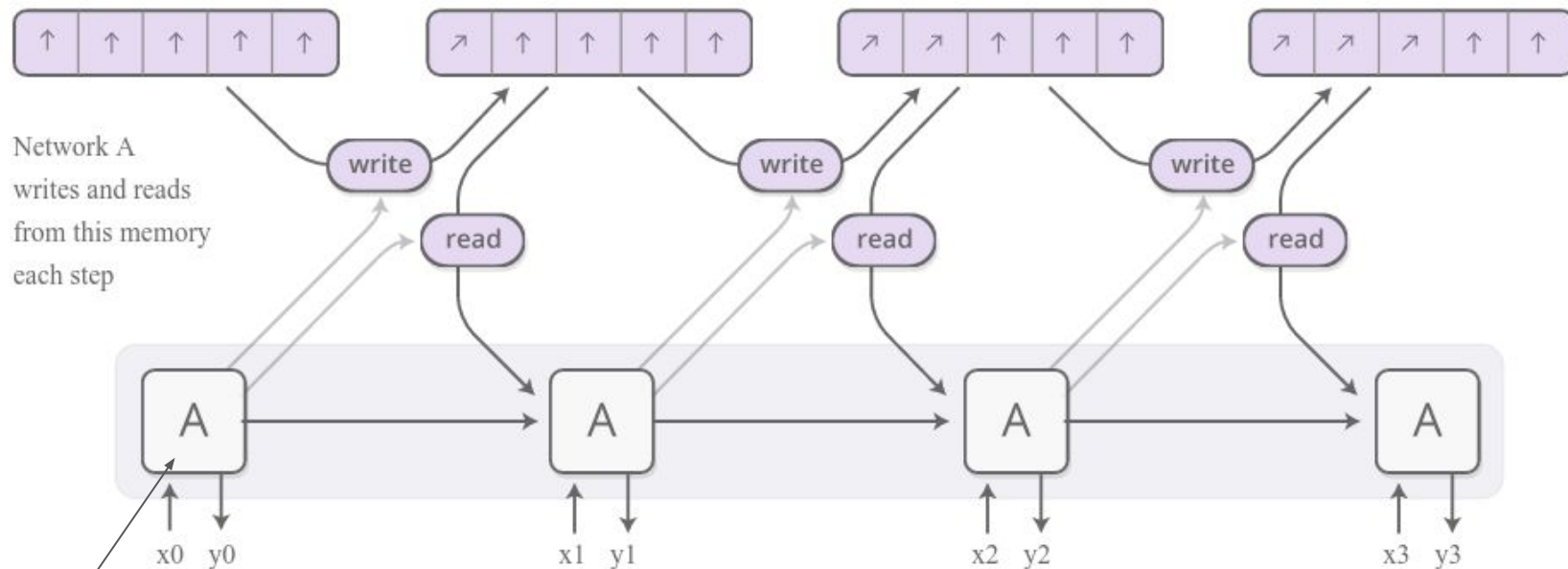


Neural Turing Machine

- Add explicit memory to the RNN
 - Use attention to read and write from the memory
 - Trained on simple “algorithmic” task like, copying input sequence
- the important results is that the NTM often

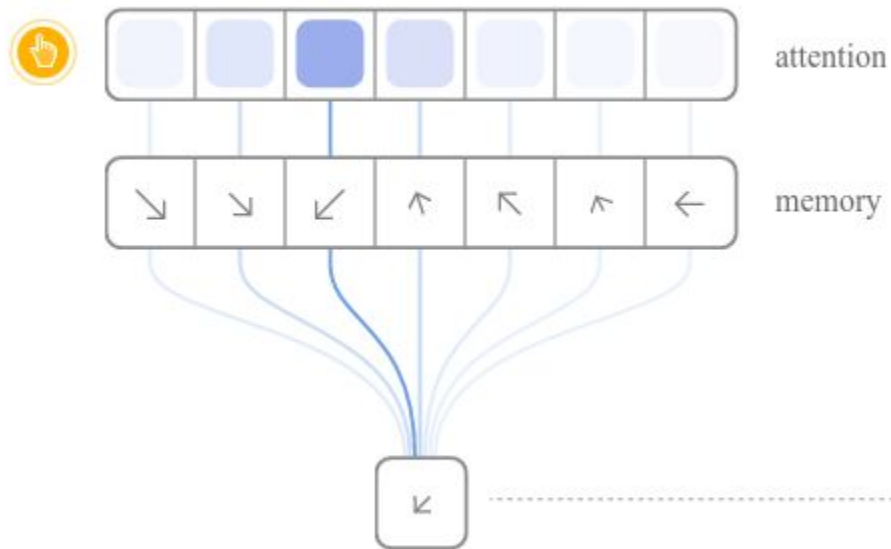
Neural Turing Machine

Memory is an array of vectors



Controller

Reading from memory

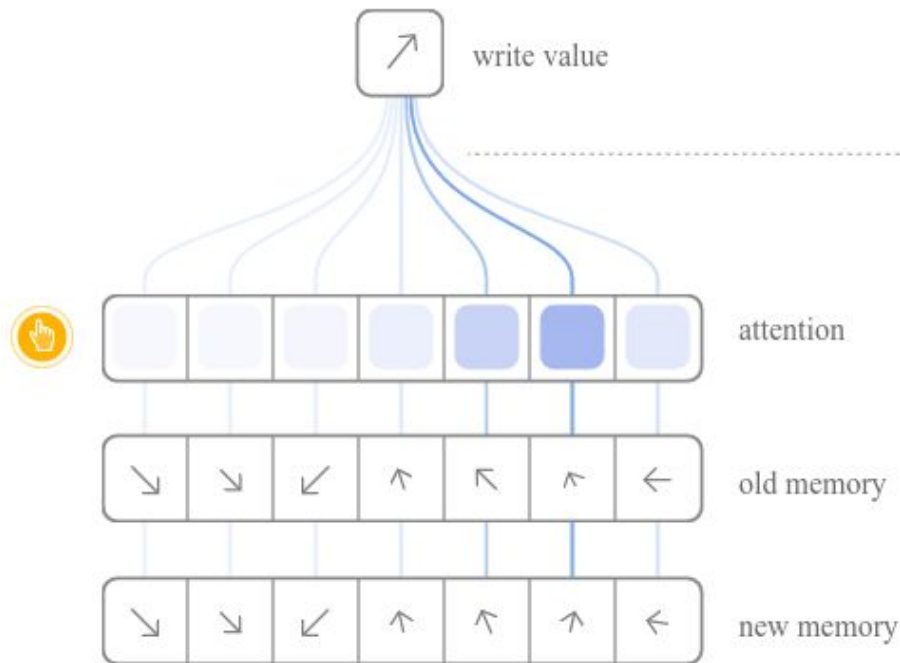


The RNN gives an attention distribution which describe how we spread out the amount we care about different memory positions

The read result is a weighted sum.

$$r \leftarrow \sum_i a_i M_i$$

Writing to memory



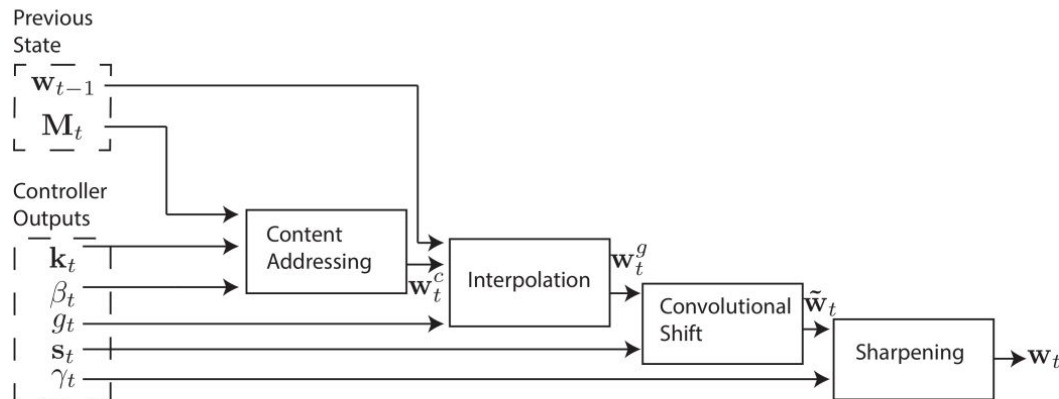
Instead of writing to one location, we write everywhere, just to different extents.

The RNN gives an attention distribution, describing how much we should change each memory position towards the write value.

$$M_i \leftarrow a_i w + (1 - a_i) M_i$$

Where to read/write

- Content-based addressing (similar to attention in the translation model) - compare some query vector with every cell in the memory
- Location-based addressing - do some operation on the attention weighting, like shifting it to the right (this is realised using a convolution). Enables the NTM controller to iterate through memory cells
- Pretty complicated and other ways to do also possible



Generalization LSTM vs NTM

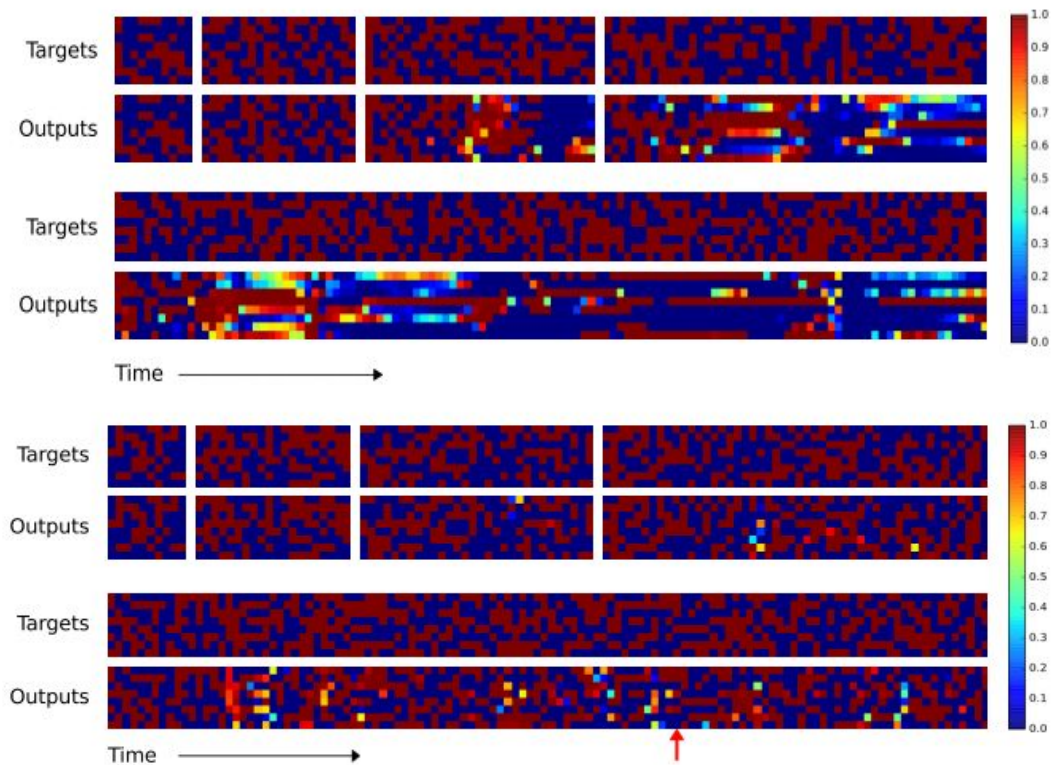
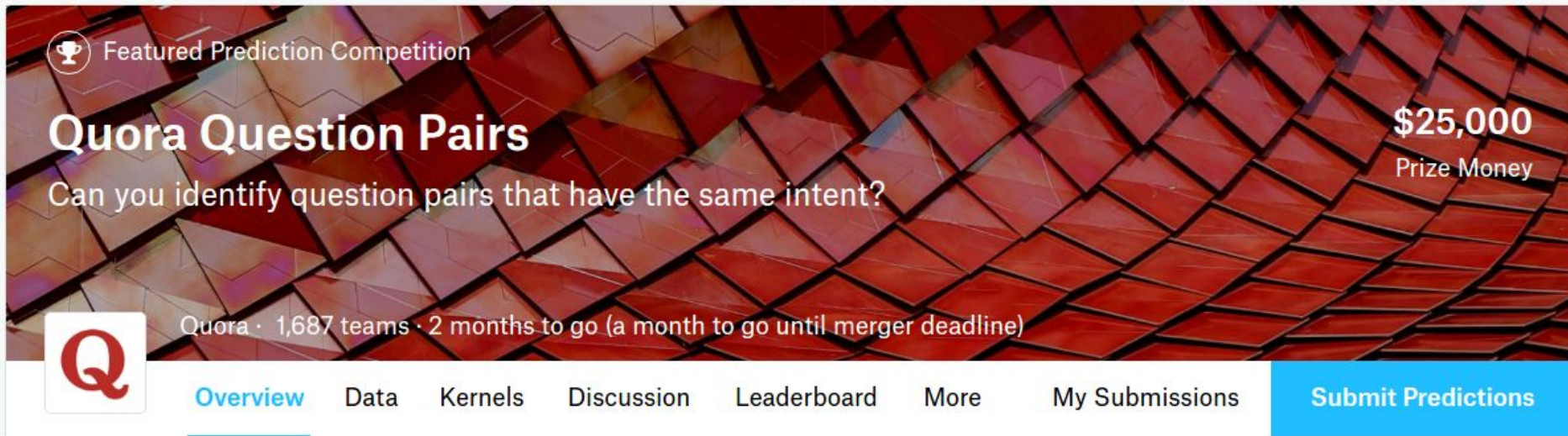


Figure 4: NTM Generalisation on the Copy Task. The four pairs of plots in the top row

NTM

- Nice explanation in <http://distill.pub/2016/augmented-rnns/>
- Lots of follow up papers exploring similar directions. Adding some external memory to a network. Sometime called neural calculators.

Quora competition

The banner features a background of red, overlapping, geometric shapes resembling a stylized roof or a mosaic. In the top left corner, there is a trophy icon followed by the text "Featured Prediction Competition". The main title "Quora Question Pairs" is prominently displayed in white, bold font. Below it, a subtitle asks "Can you identify question pairs that have the same intent?". On the right side, the prize money "\$25,000" is shown in a large, bold font, with "Prize Money" written below it. At the bottom left, the Quora logo (a red 'Q' in a white square) is present, followed by the text "Quora · 1,687 teams · 2 months to go (a month to go until merger deadline)". A navigation bar at the bottom contains links: "Overview" (underlined), "Data", "Kernels", "Discussion", "Leaderboard", "More", "My Submissions", and a blue button labeled "Submit Predictions".


Featured Prediction Competition

Quora Question Pairs

Can you identify question pairs that have the same intent?

\$25,000
Prize Money

Quora · 1,687 teams · 2 months to go (a month to go until merger deadline)

 [Overview](#) [Data](#) [Kernels](#) [Discussion](#) [Leaderboard](#) [More](#) [My Submissions](#) [Submit Predictions](#)

<https://www.kaggle.com/c/quora-question-pairs/>

<https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs>

Quora competition

same intent: "What are the best ways to lose weight?", "How can a person reduce weight?", "What are effective weight loss plans?"

not same intent: "Who is Darth Vader's father?", "Does Darth Vader know Yoda is still alive?"

- ~400k sentence pairs in the training set
- Ends on June 6, 2017

Quora competition

- What engineers at Quora tried:
<https://engineering.quora.com/Semantic-Question-Matching-with-Deep-Learning>
- Currently using some random forest approach
- Experimenting w DL approaches

RNN approach 1

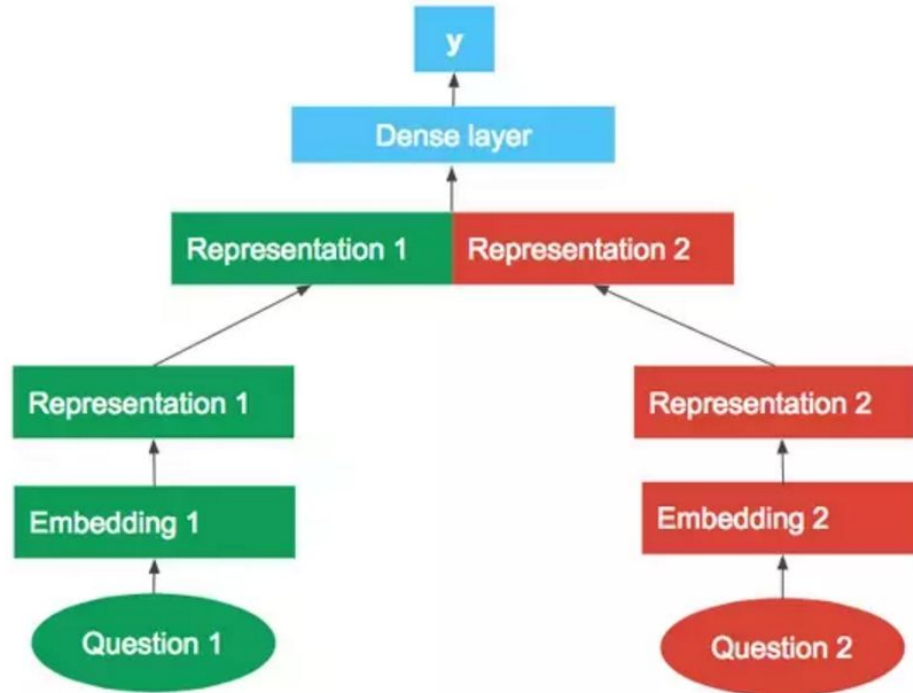


Figure 1: Architecture of approach 1, “LSTM with concatenation”

Lab instructions

Week 8 folder on Dropbox, mnist_rnn.py:

Task 1. Write simple RNN to recognize MNIST digits.

The image is 28x28. Flatten it to a 784 vector.

Pick some divisor d of 784, e.g. $d = 28$.

At each timestep the input will be d bits of the image.

Thus the sequence length will be $784 / d$

You should be able to get over 93% accuracy

Write your own implementation of RNN, you can look at the one from the slide, but do not copy it blindly.

Task 2.

Same, but use LSTM instead of simple RNN.

What accuracy do you get.

Experiment with choosing d , compare RNN and LSTM.

Again do not use builtin Tensorflow implementation. Write your own :)

Task 3*.

Make LSTM a deep bidirectional, multilayer LSTM.

Links

- <http://cs224d.stanford.edu/> - Great NLP course from Stanford
- <http://distill.pub/2016/augmented-rnns/>
- <https://engineering.quora.com/Semantic-Question-Matching-with-Deep-Learning>
- Show and Tell: A Neural Image Caption Generator, <https://arxiv.org/abs/1411.4555>
- Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, <https://arxiv.org/abs/1502.03044>
- Neural Turing Machine, <https://arxiv.org/abs/1410.540>
- Sequence to Sequence Learning with Neural Networks, <https://arxiv.org/abs/1409.3215>
- min-char-rnn <https://gist.github.com/karpathy/d4dee566867f8291f086>
- <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- <https://www.tensorflow.org/tutorials/recurrent>