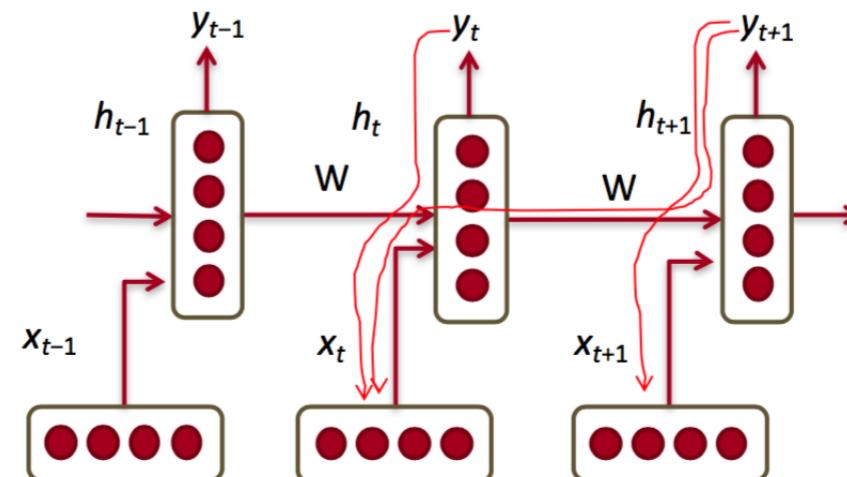


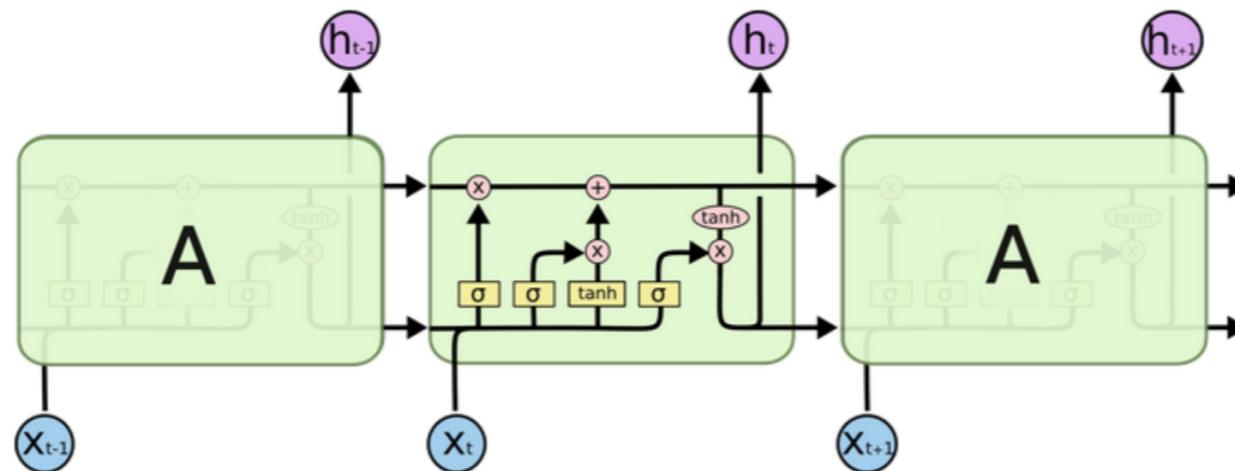
# 8. RNN 应用

# 复习一下

- BPTT算法



- LSTM 模型



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

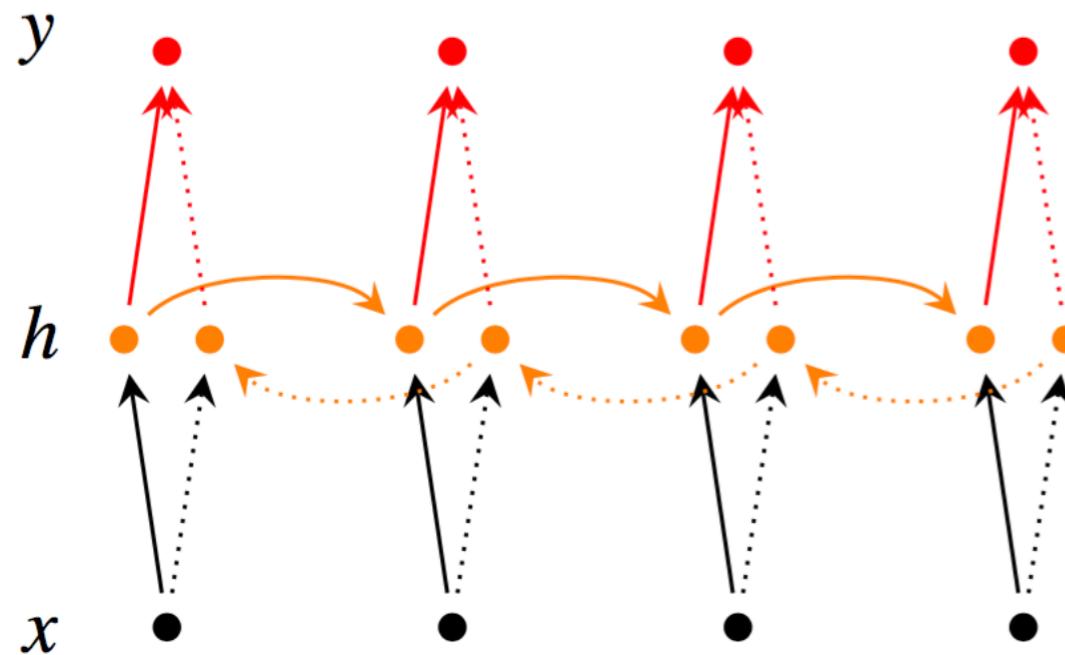
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

# 双向RNN

- 利用前后序列的信息

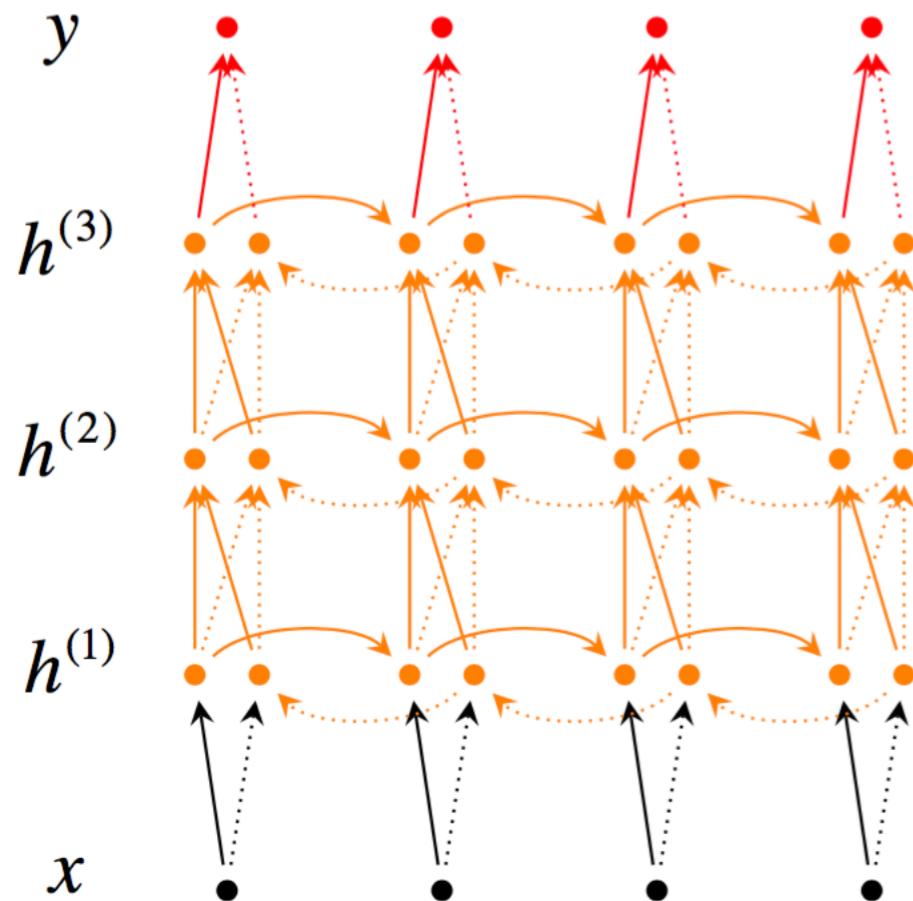


$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

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

$$y_t = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

- 复杂的多层次双向RNN模型



$$\overset{\rightarrow}{h}_t^{(i)} = f(\overset{\rightarrow}{W}^{(i)} \overset{\rightarrow}{h}_t^{(i-1)} + \overset{\rightarrow}{V}^{(i)} \overset{\rightarrow}{h}_{t-1}^{(i)} + \overset{\rightarrow}{b}^{(i)})$$

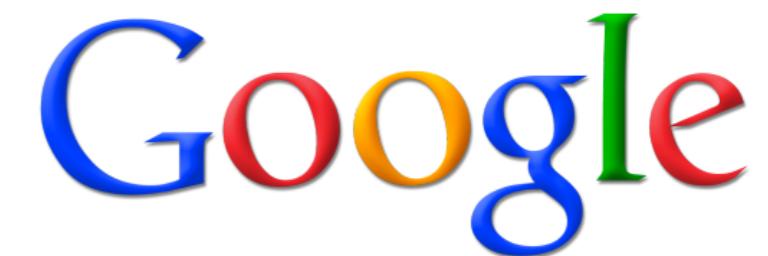
$$\overset{\leftarrow}{h}_t^{(i)} = f(\overset{\leftarrow}{W}^{(i)} \overset{\leftarrow}{h}_t^{(i-1)} + \overset{\leftarrow}{V}^{(i)} \overset{\leftarrow}{h}_{t+1}^{(i)} + \overset{\leftarrow}{b}^{(i)})$$

$$y_t = g(U[\overset{\rightarrow}{h}_t^{(L)}; \overset{\leftarrow}{h}_t^{(L)}] + c)$$

注意：整个网络依然是有向无环的网络，依然可以计算BP

# Learning to Execute

by Wojciech Zaremba and Ilya Sutskever  
ref: <http://arxiv.org/abs/1410.4615>



[https://github.com/wojciechz/learning\\_to\\_execute](https://github.com/wojciechz/learning_to_execute)

# Examples

**Input:**

```
i=8827  
c=(i-5347)  
print((c+8704) if 2641<8500 else  
      5308)
```

**Target:** 12184.

**Input:**

```
j=8584  
for x in range(8):  
    j+=920  
b=(1500+j)  
print((b+7567))
```

**Target:** 25011.

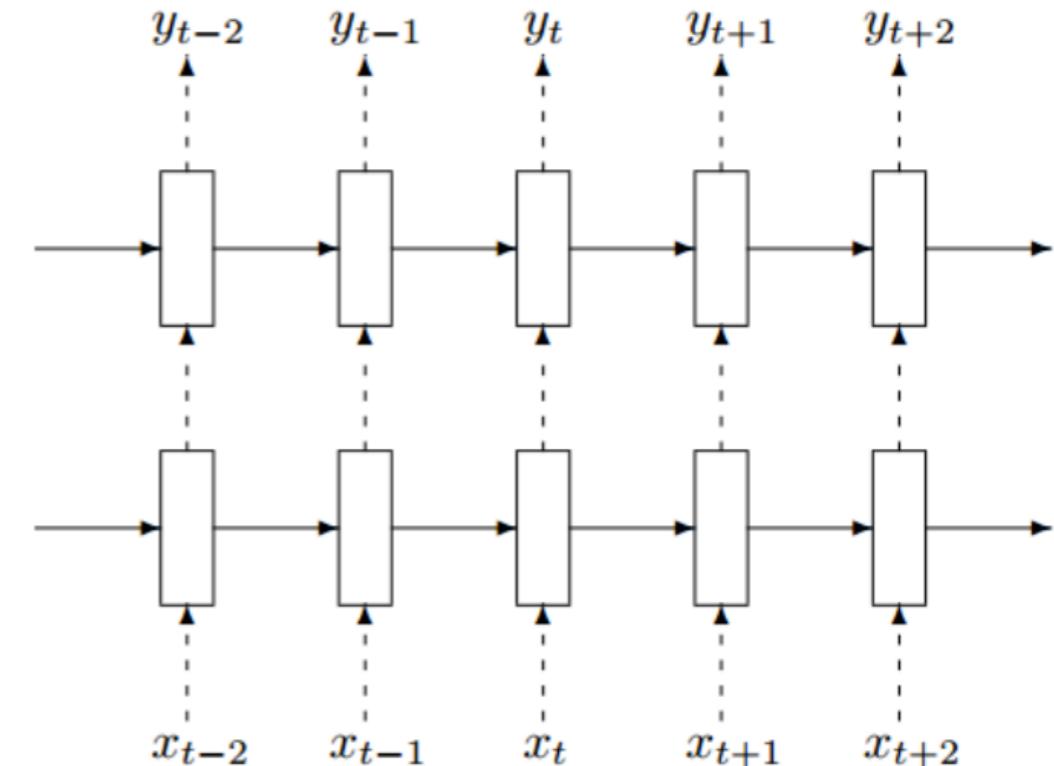
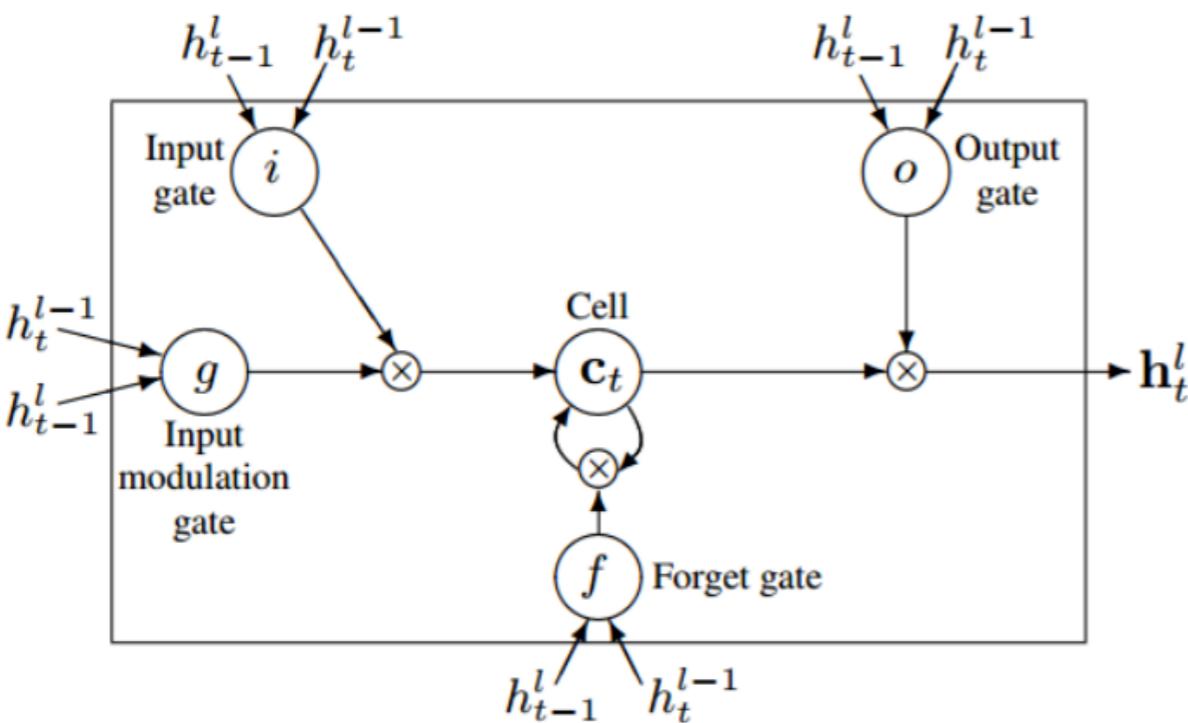
Sequence of character on the input and on the output.

# 序列数据复杂性

- 序列中相关距离可能很长
- 需要有记忆功能（变量代码）
- 代码中又有分支
- 多种任务

# Our model - RNN

- 2 layers
- 400 units each
- trained with SGD
- cross-entropy loss
- Input vocabulary size 42
- Output vocabulary size 11



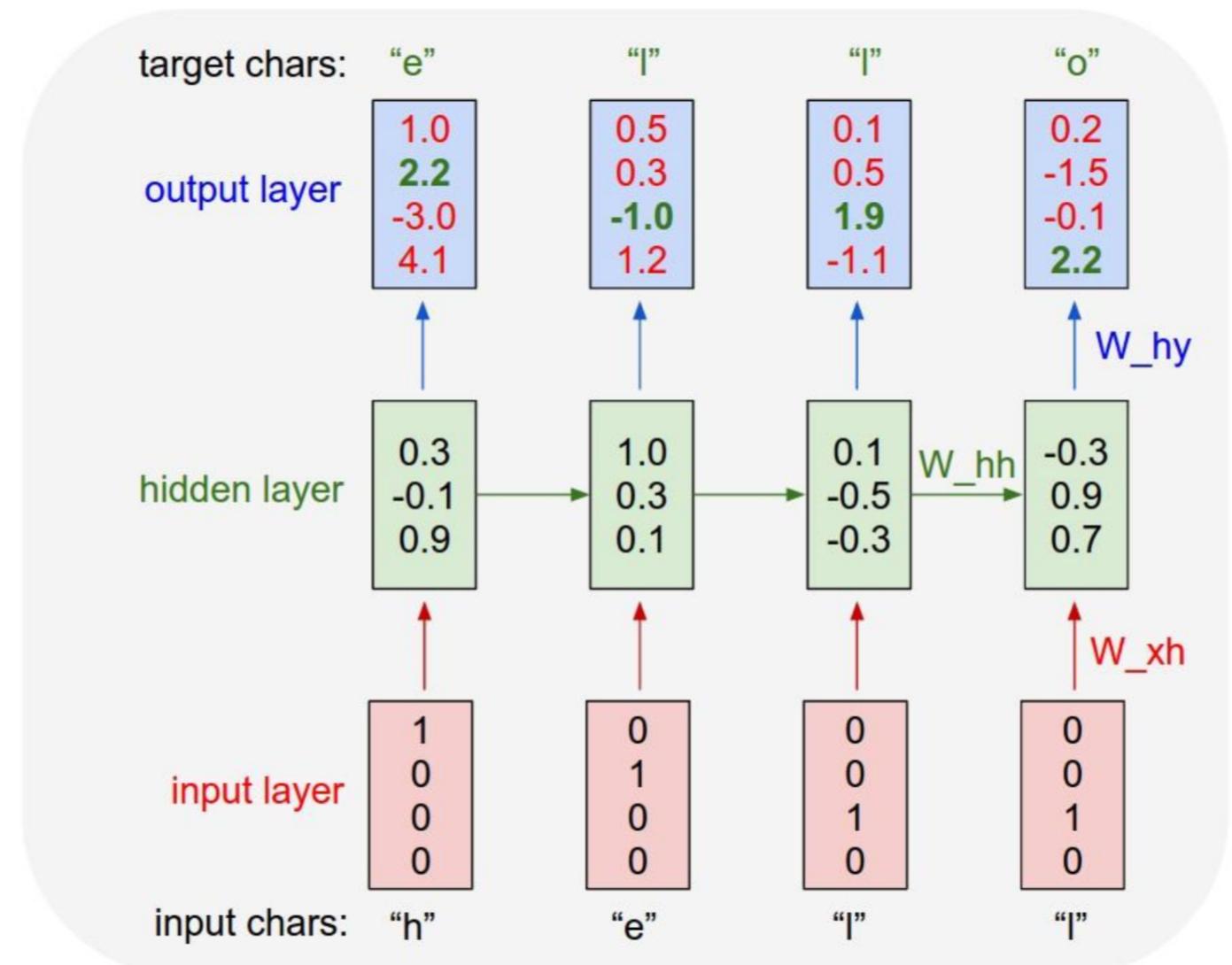
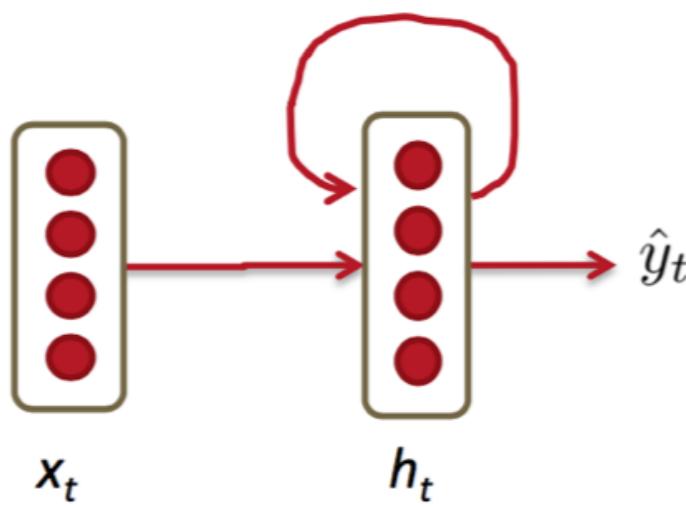
# 关于模型

- 如何训练?
  - 样本的顺序: 先易后难 vs 难易交替
  - 样本的类型: 循环代码 vs 解析代码
- 理解了代码 vs 记忆太多了的代码?

# 案例2

# 字符语言模型

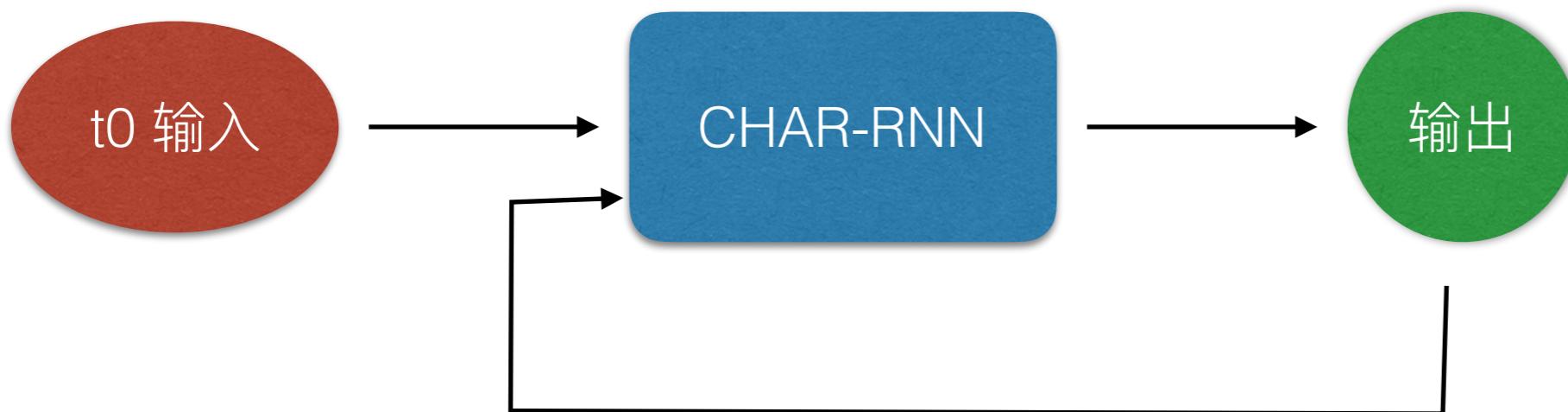
- 字符序列输入
- 预测下一个字符



<https://github.com/karpathy/char-rnn>

# 文本生成

- 在通过大量的样本训练好预测模型之后，我们可以利用这个模型来生产我们需要的文本



# 理解一下完整实现

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

<https://github.com/Teaonly/beginlearning/blob/master/july/min-char-rnn.py>

## min-char-rnn.py gist

```
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 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
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         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]]) # softmax (cross-entropy loss)
44         # backward pass: compute gradients going backwards
45         dnh = np.zeros_like(bh) # dh0 (at start of unrolled loop)
46         dhy, dby, dbh, dWxh, dWhh, dWhy = np.zeros_like(why), np.zeros_like(whh), np.zeros_like(bh)
47         dnext = np.zeros_like(hs[0])
48         for t in reversed(xrange(len(inputs))):
49             dy = np.copy(ps[t]) # y = target
50             dy[targets[t]] = 1 # backprop into y
51             dhy = np.dot(dy, hs[t].T)
52             dby += dhy
53             dnh = np.dot(dhy, Whh.T) + dnh # backprop into h
54             ddraw = (1 - hs[t]**2) * dnh # backprop through tanh nonlinearity
55             dWhh += np.dot(ddraw, hs[t-1].T)
56             dWxh += np.dot(draw, xs[t].T)
57             dbh += np.dot(draw, hs[t-1].T)
58             dnext = np.dot(Whh.T, ddraw)
59             for dparam in [dnh, dhy, dby, dbh, dWxh]:
60                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61             return loss, dnh, dhy, dbh, dWxh, hs[-1]
62
63 def sample(h, seed_ix):
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 range(seq_length):
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[ix] = 1
77         ixes.append(ix)
78     return ixes
79
80 n, D = 0, 0
81 dWxh, dWhh, dhy = np.zeros_like(why), np.zeros_like(whh), np.zeros_like(bh)
82 dbh, dby = np.zeros_like(bh), np.zeros_like(by)
83 smooth_loss = np.log(1.0/vocab_size)*seq_length # loss at iteration 0
84 while True:
85     # prepare inputs (we're sweeping from left to right in steps seq_length long)
86     if pseq_length == len(data) or n == 0:
87         inputs = np.zeros((seq_length, vocab_size)) # reset RNN memory
88         p = a = np.zeros((vocab_size, 1)) # from start of data
89     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
90     targets = [char_to_ix[ch] for ch in data[p+seq_length+1]]
91
92     # sample from the model now and then
93     if n % 100 == 0:
94         sample_ix = sample(hprev, inputs[0], 200)
95         txt = ''.join(ix_to_char[i] for i in sample_ix)
96         print '----\n' + txt + '----'
97
98     # forward seq_length characters through the net and fetch gradient
99     loss, dnh, dhy, dbh, dWxh, dWhh, dhy = lossFun(inputs, targets, hprev)
100    smooth_loss = smooth_loss * 0.999 + loss * 0.001
101    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss)
102
103    # perform parameter update with adam
104    for param, dparam, mem in zip([why, whh, bh], [dWhy, dWhh, dbh], [dWxh, dWhh, dhy]):
105        mem += dparam * dparam
106        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
107
108    p += seq_length # move data pointer
109    n += 1 # iteration counter
```

## 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) }
```

## min-char-rnn.py gist

```

1 /**
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4 ***
5
6 import numpy as np
7
8 # data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print('data has %d characters, %d unique.' % (data_size, vocab_size))
13 char_to_ix = { ch:i for i,ch in enumerate(chars) }
14 ix_to_char = { i:ch for i,ch in enumerate(chars) }
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # model parameters
22 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
23 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
24 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
25 bh = np.zeros((hidden_size, 1)) # hidden bias
26 by = np.zeros((vocab_size, 1)) # output bias
27
28 def lossFun(inputs, targets, hprev):
29     """
30     inputs,targets are both list of integers.
31     hprev is mx1 array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     xs, hs, ys, ps = [], [], [], []
35     hs[0] = np.copy(hprev)
36     loss = 0
37     for t in range(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]) / np.sum(np.exp(ys[t])) # softmax (cross-entropy loss)
44         # backward pass: compute gradients going backwards
45         dsh, dWxh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46         dbh, dy = np.zeros_like(bh), np.zeros_like(by)
47         dhnext = np.zeros_like(hs[0])
48         for t in reversed(range(len(inputs))):
49             dy = np.copy(ps[t])
50             dy[targets[t]] -= 1 # backprop into y
51             dyh = np.dot(Why.T, dy) + dhnext # backprop into h
52             dhy = (1 - hs[t] * hs[t].T) * dyh # backprop through tanh nonlinearity
53             dsh = np.dot(dhy, xs[t].T)
54             dWxh += np.dot(dsh, xs[t].T)
55             dWhy += np.dot(dsh, hs[t-1].T)
56             dhnext = np.dot(Whh.T, dhy)
57         for dparam in [dsh, dWxh, dWhy, dbh, dy]:
58             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
59     return loss, dsh, dWxh, dWhy, dbh, dy, hs[-1]
60
61 def sample(h, seed_ix, n):
62     """
63     sample a sequence of integers from the model
64     h is memory state, seed_ix is seed letter for first time step
65     """
66     x = np.zeros((vocab_size, 1))
67     x[seed_ix] = 1
68     ixes = []
69     for t in range(n):
70         h = np.tanh(np.dot(Whh, x) + np.dot(Wxh, h) + bh)
71         y = np.dot(Why, h) + by
72         p = np.exp(y) / np.sum(np.exp(y))
73         ix = np.random.choice(range(vocab_size), p=p.ravel())
74         x = np.zeros((vocab_size, 1))
75         x[ix] = 1 # reset
76         ixes.append(ix)
77     return np.array(ixes)
78
79 n = 0
80 Wxh, Whh, Why = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
81 dbh, dy = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
82 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
83 while True:
84     # forward seq_length inputs (we're moving from left to right in steps seq_length long)
85     if p+seq_length >= len(data) or n == 0:
86         hprev = np.zeros((hidden_size,1)) # reset non memory
87         p = 0 # go from start of data
88         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
89         targets = [char_to_ix[ch] for ch in data[p+seq_length+1:p+2*seq_length+1]]
90
91     # sample from the model now and then
92     if n % 100 == 0:
93         sample_ix = sample(hprev, inputs[0], 200)
94         txt = ''.join(ix_to_char[i] for i in sample_ix)
95         print('----\n' + txt + '\n----')
96
97     # forward seq_length characters through the net and fetch gradient
98     loss, dsh, dWxh, dWhy, dbh, dy, hprev = lossFun(inputs, targets, hprev)
99     smooth_loss = smooth_loss * 0.999 + loss * 0.001
100    if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
101
102    # perform parameter update with Adagrad
103    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
104                                 [dWxh, dWhh, dWhy, dbh, dy],
105                                 [mem, mem, mem, mem, mem]):
106        mem += dparam * dparam
107        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update
108
109    p += seq_length # move data pointer
110    n += 1 # iteration counter

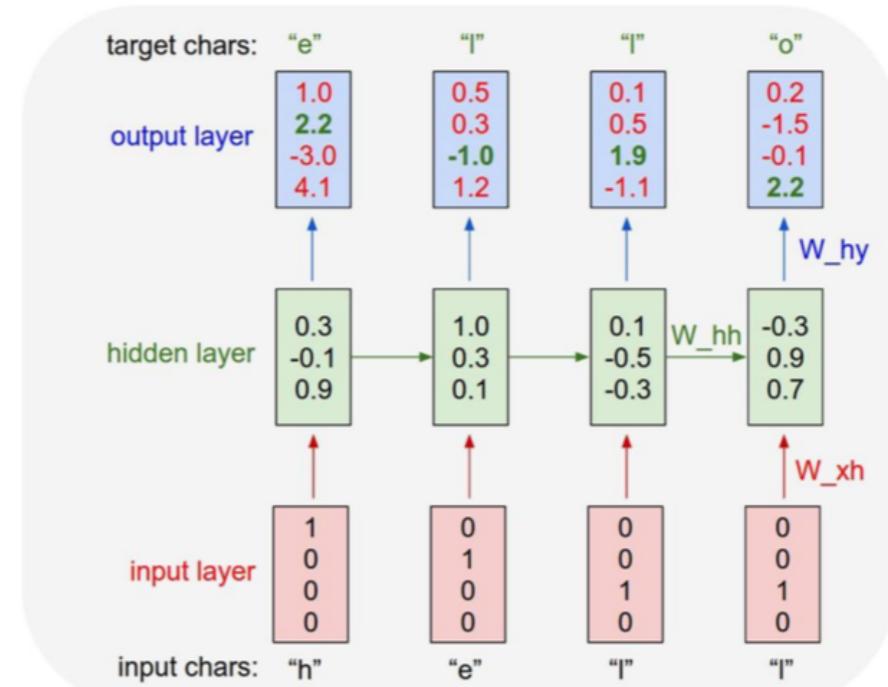
```

# hyperparameters  
**hidden\_size = 100** # size of hidden layer of neurons  
**seq\_length = 25** # number of steps to unroll the RNN for  
**learning\_rate = 1e-1**

# model parameters  
**Wxh = np.random.randn(hidden\_size, vocab\_size)\*0.01** # input to hidden  
**Whh = np.random.randn(hidden\_size, hidden\_size)\*0.01** # hidden to hidden  
**Why = np.random.randn(vocab\_size, hidden\_size)\*0.01** # hidden to output  
**bh = np.zeros((hidden\_size, 1))** # hidden bias  
**by = np.zeros((vocab\_size, 1))** # output bias

## Initializations

recall:



## min-char-rnn.py gist

```
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 Wxh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.rand(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 hx 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     for t in xrange(len(inputs)):
37         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
38         xs[t][inputs[t]] = 1
39         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
40         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
41         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
42         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
43     # backward pass: compute gradients going backwards
44     dех, dехn, dmy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
45     dех, dmy = np.zeros_like(bh), np.zeros_like(by)
46     dhnxt = np.zeros_like(hs[0])
47     for t in reversed(xrange(len(inputs))):
48         dy = np.copy(ps[t])
49         dy[targets[t]] -= 1 # backprop into y
50         dmy += np.dot(dy, hs[t].T)
51         dy *= Why.T
52         dех += np.dot(dmy, xs[t].T) # backprop through tanh nonlinearity
53         dhnxt = (1 - hs[t]**2) * dhnxt # backprop through tanh nonlinearity
54         dехn += np.dot(dhnxt, xs[t].T)
55         dhnxt = np.dot(Whh, hs[t-1].T)
56         dhnxt += np.dot(Wxh, hs[t].T)
57     for dparam, deх, dmy, dhn, dby in zip([dех, dехn, dmy, dhn, dby], [dех, dехn, dmy, dhn, dby]):
58         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
59     return loss, dех, dехn, dmy, dhn, dby, hs[-1]
60
61 def sample(h, seed_ix, n):
62     """
63     sample a sequence of integers from the model
64     h is memory state, seed_ix is seed letter for first time step
65     """
66     x = np.zeros((vocab_size, 1))
67     x[seed_ix] = 1
68     ixes = []
69     for t in xrange(n):
70         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
71         p = np.exp(h) / np.sum(np.exp(h))
72         ix = np.random.choice(range(vocab_size), p=p.ravel())
73         x = np.zeros((vocab_size, 1))
74         x[ix] = 1
75         ixes.append(ix)
76     return ixes
77
78 n, p = 0, 0
79 mesh, mdeh, mmy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
80 mdeh, mmy = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
81 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
82 while True:
83     # prepare inputs (we're sweeping from left to right in steps seq_length long)
84     if p+seq_length+1 >= len(data) or n == 0:
85         hprev = np.zeros((hidden_size,1)) # reset RNN memory
86         p = 0 # go from start of data
87     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
88     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
89
90     # sample from the model now and then
91     if n % 100 == 0:
92         sample_ix = sample(hprev, inputs[0], 200)
93         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
94         print '----\n %s \n----' % (txt, )
95
96     # forward seq_length characters through the net and fetch gradient
97     loss, dех, dехn, dmy, dhn, dby, hprev = lossFun(inputs, targets, hprev)
98     smooth_loss = smooth_loss * 0.999 + loss * 0.001
99
100    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
101
102    # perform parameter update with Adagrad
103    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
104                                 [dех, dехn, dmy, dhn, dby],
105                                 [mdeh, mWhh, mWhy, mbh, mby]):
106        mem += dparam * dparam
107        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
108
109    p += seq_length # move data pointer
110    n += 1 # iteration counter
111
112    p += seq_length # move data pointer
113    n += 1 # iteration counter
```

## Main loop



## min-char-rnn.py gist

```

1 """
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4 """
5
6 import numpy as np
7
8 # data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = { ch:i for i,ch in enumerate(chars) }
14 ix_to_char = { i:ch for i,ch in enumerate(chars) }
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # model parameters
22 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
23 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
24 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
25 bh = np.zeros((hidden_size, 1)) # hidden bias
26 by = np.zeros((vocab_size, 1)) # output bias
27
28 def lossFun(inputs, targets, hprev):
29     """
30     inputs,targets are both list of integers.
31     hprev is Hx1 array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     xs, hs, ys, ps = [], [], [], []
35     hs[-1] = np.copy(hprev)
36     loss = 0
37     n = 0
38     for x in inputs:
39         # forward pass
40         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
41         x[0][int(x)] = 1
42         hs_t = np.tanh(np.dot(Wxh, xs[-1]) + np.dot(Whh, hs[-1]) + bh) # hidden state
43         ys_t = np.dot(Why, hs_t) + by # unnormalized log probabilities for next chars
44         ps_t = np.exp(ys_t) / np.sum(np.exp(ys_t)) # probabilities for next chars
45         loss += -np.log(ps_t[int(targets[n])]) # softmax (cross-entropy loss)
46         # backward pass: compute gradients going backwards
47         dws, dwh, dby = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
48         dbh, dy = np.zeros_like(bh), np.zeros_like(by)
49         dnext = np.zeros_like(hs[0])
50         for t in reversed(xrange(len(inputs))):
51             dy = np.copy(dy[t])
52             dy[targets[t]] -= 1 # backprop into y
53             dhy = -np.dot(dy, hs[t].T)
54             dy += dhy
55             dh = -np.dot(Why.T, dy) * dnext # backprop into h
56             dhw = (1 - hs[t]**2) * hs[t] * dh # backprop through tanh nonlinearity
57             dbh += dh
58             dhy += np.dot(dhw, hs[t].T)
59             dnext = np.dot(Whh.T, dhw)
60             dby += np.sum(dhy)
61             for dparam in [dws, dwh, dbh, dby]:
62                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
63             return loss, dws, dwh, dbh, dby, dbh, dy, hs[-len(inputs)-1]
64
65 def sample(h, seed_ix, n):
66     """
67     h is memory state, seed_ix is seed letter for first time step
68     """
69     x = np.zeros((vocab_size, 1))
70     x[seed_ix] = 1
71     ixes = []
72     for t in range(n):
73         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
74         y = np.dot(Why, h) + by
75         p = np.exp(y) / np.sum(np.exp(y))
76         ix = np.random.choice(range(vocab_size), p=p.ravel())
77         x[0][int(ix)] = 1
78         ixes.append(ix)
79     return ixes
80
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
86 while True:
87     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length+1 >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size,1)) # reset RNN memory
90         p = 0 # go from start of data
91     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print '----\n %s \n----' % (txt, )
99
100    # forward seq_length characters through the net and fetch gradient
101    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102    smooth_loss = smooth_loss * 0.999 + loss * 0.001
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

```

## Main loop



## min-char-rnn.py gist

```
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 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
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     for t in range(len(inputs)):
37         xs[t] = np.zeros([vocab_size,1]) # encode in 1-of-k representation
38         xs[t][inputs[t]] = 1
39         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
40         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
41         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
42         loss += -np.log(ps[t])[targets[t], 0] # softmax (cross-entropy loss)
43
44     # backward pass: compute gradients going backwards
45     dWxh, 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         dhy = np.dot(Why.T, dy) + dhnext # backprop into h
52         dh = np.dot(Whh.T, dhy) + dhy # backprop through tanh nonlinearity
53         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
54         dbh += np.sum(ddraw)
55         dWxh += np.dot(ddraw, xs[t].T)
56         dWhh += np.dot(ddraw, hs[t-1].T)
57         dhnext = np.dot(Whh.T, ddraw)
58     for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dWxh, dWhh, dWhy, dbh, dby, hs[-1]
61
62 def sample(h, seed_ix, n):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     if type(h) == np.ndarray:
68         h = np.zeros([vocab_size, 1])
69     ixes = []
70     for t in range(n):
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         ixs.append(ix)
78     return ixs
79
80 n, p = 0, 0
81 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
82 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
83 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
84
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
113
114    if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
115
116    # perform parameter update with Adagrad
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
```

## Main loop



## min-char-rnn.py gist

```

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 Wxh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.rand(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 hxi array of initial hidden state
31 returns the loss, gradients on model parameters, and last hidden state
32 """
33 xs, hs, ys, ps = ([], [], [], 0)
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
45 # backward pass: compute gradients going backwards
46 dprev, dh0, dwhy = np.zeros_like(wh0), np.zeros_like(whh), np.zeros_like(why)
47 dbh, dy = np.zeros_like(bh), np.zeros_like(by)
48 dnext = np.zeros_like(hs[0])
49 for t in reversed(xrange(len(inputs))):
50     dy = np.copy(dy[t])
51     dy[targets[t]] = 1 # backprop into y
52     dhy = -np.dot(dy, hs[t].T)
53     dprev = np.dot(why.T, dy) # dnext = backprop into h
54     dhraw = (1 - hs[t] * hs[t]) * dhy # dh = backprop through tanh nonlinearity
55     dbh += dhy
56     dwhy += np.dot(dhraw, xs[t].T)
57     dhs = np.dot(whh, hs[t-1].T)
58     dnext = np.dot(whh, dhs)
59
60 for dparam in [dprev, dh0, dwhy, dbh, dby]:
61     np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
62 return loss, dprev, dh0, dwhy, dbh, dby, hs[-1]
63
64 def sample(h, seed_ix, n):
65 """
66 sample a sequence of integers from the model
67 h is memory state, seed_ix is seed letter for first time step
68 """
69 x = np.zeros((vocab_size, 1))
70 ixes = []
71 for t in range(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
80
81 P = R = 0
82 mwh0, mwhh, mwwhy = np.zeros_like(wh0), 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     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length+1 >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
90         p = 0 # go from start of data
91         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print '----\n%s\n----' % (txt, )
99
100    # forward seq_length characters through the net and fetch gradient
101    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102    smooth_loss = smooth_loss * 0.999 + loss * 0.001
103
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, mwwhy, mbh, mbby]):
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

```

## Main loop



```

81 n, p = 0, 0
82 mWxh, mWhh, mwwhy = 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     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length+1 >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
90         p = 0 # go from start of data
91         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print '----\n%s\n----' % (txt, )
99
100    # forward seq_length characters through the net and fetch gradient
101    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102    smooth_loss = smooth_loss * 0.999 + loss * 0.001
103
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, mwwhy, mbh, mbby]):
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

```

## min-char-rnn.py gist

```

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 vocab_size = len(chars), len(chars)
11 print 'data has %d characters, %d unique.' % (vocab_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 = 20 # 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
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is mx1 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(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
45     # backward pass: compute gradients going backwards
46     dех, dWhh, dWhy, dbh, dby = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
47     dhnxt = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.zeros((vocab_size, 1))
50         dy[targets[t]] = 1 # backprop into y
51         dby += dy
52         dh = np.dot(Why.T, dy) + dhnxt # backprop into h
53         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
54         dWhh += np.dot(ddraw, hs[t-1].T)
55         dbh += dWhh
56         dWxh += np.dot(ddraw, xs[t].T)
57         dhnxt = np.dot(Whh, ddraw)
58
59     for dparam in [dех, dWhh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dех, dWhh, dWhy, dbh, dby, hs[-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     if seed_ix is None:
69         ix = np.zeros((vocab_size, 1))
70     else:
71         ix = np.zeros((vocab_size, 1))
72         ix[seed_ix] = 1
73
74     for t in xrange(n):
75         h = np.tanh(np.dot(Whh, h) + bh)
76         y = np.dot(Why, h) + by
77         p = np.exp(y) / np.sum(np.exp(y))
78         ix = np.argmax(np.random.multinomial(1, p))
79         ix = np.zeros((vocab_size, 1))
80         ix[ix] = 1
81         ixes.append(ix)
82
83    return ixes
84
85 n, p = 0, 0
86 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
87 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
88 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
89
90 while True:
91     # prepare inputs (we're sweeping from left to right in steps seq_length long)
92     if p+seq_length+1 >= len(data) or n == 0:
93         hprev = np.zeros((hidden_size,1)) # reset RNN memory
94         p = 0 # go from start of data
95     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
96     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
97
98     # sample from the model now and then
99     if n % 100 == 0:
100         sample_ix = sample(hprev, inputs[0], 200)
101         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
102         print '----\n %s \n----' % (txt, )
103
104     # forward seq_length characters through the net and fetch gradient
105     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
106     smooth_loss = smooth_loss * 0.999 + loss * 0.001
107
108     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
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, mby]):
114         mem += dparam * dparam
115         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
116
117         p += seq_length # move data pointer
118         n += 1 # iteration counter
119
120         n += 1 # iteration counter

```

## Main loop



## min-char-rnn.py gist

```

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 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
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         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     dех, dехy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46     dby = np.zeros_like(by), 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         dу += np.dot(dy, hs[t].T)
52         dh = np.dot(Why.T, dy) + dhnext # backprop into h
53         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
54         dх += ddraw
55         dех += np.dot(ddraw, xs[t].T)
56         dехy += np.dot(ddraw, hs[t-1].T)
57         dhnext = np.dot(Whh.T, ddraw)
58
59     for dparam in [dех, dехy, dехy, dby, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dех, dехy, dехy, 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(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
80
81 n, p = 0, 0
82 mesh, dех, dехy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 dех, dехy = np.zeros_like(by), 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 to 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 %d: %s' % (n, txt)
98
99     # forward seq_length characters through the net and fetch gradient
100 loss, dех, dехy, dехy, 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, mem in zip([Wxh, Whh, Why, bh, by],
106                       [dех, dехh, dехy, dех, dby]):
107     mem += dparam * dparam
108     param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
109
110 p += seq_length # move data pointer
111 n += 1 # iteration counter

```

## 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
45     # backward pass: compute gradients going backwards
46     dех, dехh, dехy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
47     dех, dехy = np.zeros_like(by), np.zeros_like(by)
48     dhnext = np.zeros_like(hs[0])
49     for t in reversed(xrange(len(inputs))):
50         dy = np.copy(ps[t])
51         dy[targets[t]] -= 1 # backprop into y
52         dу += np.dot(dy, hs[t].T)
53         dh = np.dot(Why.T, dy) + dhnext # backprop into h
54         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dх += ddraw
56         dех += np.dot(ddraw, xs[t].T)
57         dехh += np.dot(ddraw, hs[t-1].T)
58         dhnext = np.dot(Whh.T, ddraw)
59
60     for dparam in [dех, dехh, dехy, dех, dехy]:
61         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
62     return loss, dех, dехh, dехy, dех, dехy, hs[len(inputs)-1]

```

## min-char-rnn.py gist

```

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)) # unique characters
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 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
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         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
45     # backward pass: compute gradients going backwards
46     dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
47     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
48     dhnext = np.zeros_like(hs[0])
49     for t in reversed(xrange(len(inputs))):
50         dy = np.copy(ps[t])
51         dy[targets[t]] -= 1 # backprop into y
52         dy *= np.dot(dy, hs[t].T)
53         dy *= dy
54         dh = np.dot(Why.T, dy) + dhnext # backprop into h
55         dhraw = (1 - hs[t]**2) * dh # backprop through tanh nonlinearity
56         dWxh += np.dot(dhraw, xs[t].T)
57         dWhh += np.dot(dhraw, hs[t-1].T)
58         dhnext = np.dot(Why.T, dhraw)
59     for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dWxh, dWhh, 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     if seed_ix is None:
69         x = np.zeros((vocab_size, 1))
70         x[seed_ix] = 1
71     else:
72         x = np.zeros((vocab_size, 1))
73         x[seed_ix] = 1
74     ixes = []
75     for t in range(n):
76         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
77         y = np.dot(Why, h) + by
78         p = np.exp(y) / np.sum(np.exp(y))
79         ix = np.random.choice(range(vocab_size), p=p.ravel())
80         x = np.zeros((vocab_size, 1))
81         x[ix] = 1
82         ixes.append(ix)
83     return ixes
84
85 n, p = 0, 0
86 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
87 dbh, dby = np.zeros_like(bh), np.zeros_like(by)
88 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
89
90 while True:
91     # prepare inputs (we're sweeping from left to right in steps seq_length long)
92     if p+seq_length-1 >= len(data) or n == 0:
93         hprev = np.zeros((hidden_size,1)) # reset RNN memory
94         p = 0 # go from start of data
95     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
96     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
97
98     # sample from the model now and then
99     if n % 100 == 0:
100         sample_ix = sample(hprev, inputs[0], 200)
101         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
102         print '----\n' + txt + '\n----'
103
104     # forward seq_length characters through the net and fetch gradient
105     loss, dWxh, dWhh, dbh, dby, hprev = lossFun(inputs, targets, hprev)
106     smooth_loss = smooth_loss * 0.999 + loss * 0.001
107     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss)
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                                 [mem, mem, mem, mem, mem]):
113         mem += dparam * dparam
114         param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update
115
116     p += seq_length # move data pointer

```

```

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)

```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

## min-char-rnn.py gist

```

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 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
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is Mx1 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]]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     dWxh, 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         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += ddraw
56         dWxh += np.dot(ddraw, xs[t].T)
57         dWhh += np.dot(ddraw, hs[t-1].T)
58         dhnext = np.dot(Whh.T, ddraw)
59     for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61
62     return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
63
64 def sample(h, seed_ix, n):
65     """
66     sample a sequence of integers from the model
67     h is memory state, seed_ix is seed letter for first time step
68     """
69     x = np.zeros((vocab_size, 1))
70     x[seed_ix] = 1
71     ixes = []
72     for t in xrange(n):
73         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
74         y = np.dot(Why, h) + by
75         p = np.exp(y) / np.sum(np.exp(y))
76         ix = np.random.choice(range(vocab_size), p=p.ravel())
77         x[0] = 0
78         ixes.append(ix)
79     return ixes
80
81 n, p = 0, 0
82 mem, mem_dot, mhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 dbh, dby = 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 '----\n' + txt + '\n----'
98
99     # forward seq_length characters through the net and fetch gradient
100    loss, dWxh, dWhh, 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)
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                                 [mem, mem, mem, mem, mem]):
108        mem += dparam * dparam
109        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update
110
111    p += seq_length # move data pointer
112    n += 1 # iteration counter

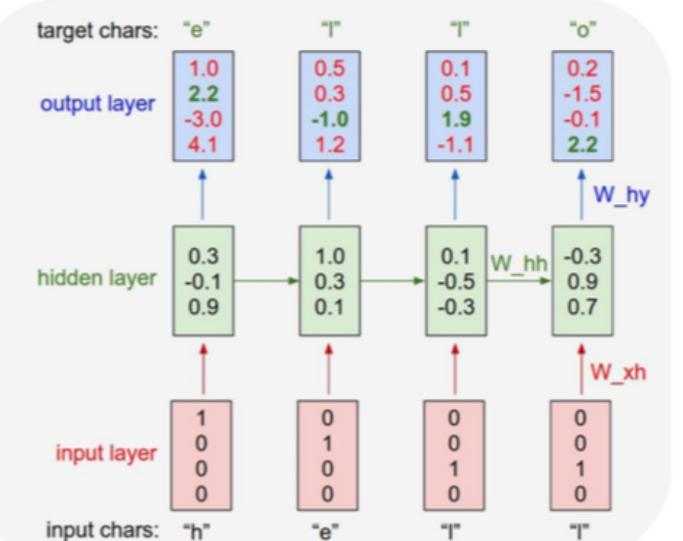
```

```

44     # backward pass: compute gradients going backwards
45     dWxh, 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         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += ddraw
56         dWxh += np.dot(ddraw, xs[t].T)
57         dWhh += np.dot(ddraw, hs[t-1].T)
58         dhnext = np.dot(Whh.T, ddraw)
59     for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61
62     return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
63
64 def sample(h, seed_ix, n):
65     """
66     sample a sequence of integers from the model
67     h is memory state, seed_ix is seed letter for first time step
68     """
69     x = np.zeros((vocab_size, 1))
70     x[seed_ix] = 1
71     ixes = []
72     for t in xrange(n):
73         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
74         y = np.dot(Why, h) + by
75         p = np.exp(y) / np.sum(np.exp(y))
76         ix = np.random.choice(range(vocab_size), p=p.ravel())
77         x[0] = 0
78         ixes.append(ix)
79     return ixes
80
81 n, p = 0, 0
82 mem, mem_dot, mhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 dbh, dby = 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 '----\n' + txt + '\n----'
98
99     # forward seq_length characters through the net and fetch gradient
100    loss, dWxh, dWhh, 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)
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                                 [mem, mem, mem, mem, mem]):
108        mem += dparam * dparam
109        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update
110
111    p += seq_length # move data pointer
112    n += 1 # iteration counter

```

recall:



## min-char-rnn.py gist

```

***  

Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)  

BSD License  

***  

import 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) }  

# hyperparameters  

hidden_size = 100 # size of hidden layer of neurons  

seq_length = 25 # number of steps to unroll the RNN for learning_rate = 1e-1  

# model parameters  

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

Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden  

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

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

by = np.zeros((vocab_size, 1)) # output bias  

def lossfun(inputs, targets, hprev):  

    """  

    inputs,targets are both list of integers.  

    hprev is hxi 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.exp(why[hs[t]]) / np.sum(np.exp(ys[t])) # unnormalized log probabilities for next chars  

        ps[t] = np.exp(ys[t]) / np.sum(ps[t]) # probabilities for next chars  

        loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)  

        # backward pass: compute gradients going backwards  

        dех, dWhh, dWуh = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)  

        dbh, dyh = np.zeros_like(bh), np.zeros_like(by)  

        dnext = np.zeros_like(hs[0])  

        for t in reversed(xrange(len(inputs))):  

            dy = np.copy(ps[t])  

            dy[targets[t]] -= 1 # backprop into y  

            dWhh += np.dot(dyh, hs[t].T)  

            dyh *= dy  

            dh = np.dot(why.T, dy) + dnext # backprop into h  

            ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity  

            dbh += ddraw  

            dWhh += np.dot(ddraw, xs[t].T)  

            dWhh += np.dot(ddraw, hs[t-1].T)  

            dnext = np.dot(Whh.T, ddraw)  

            for dparam in [dех, dWhh, dWуh, dbh, dyh]:  

                np.clip(dparam, -5, 5, out=dpParam) # clip to mitigate exploding gradients  

            loss -= np.sum(dy * xs[t])  

            dnext = dWhh + np.dot(Whh, dnext)  

    return loss, dех, dWhh, dWуh, dbh, dyh, hs[len(inputs)-1]  

def sample(h, seed_ix, n):  

    """  

    sample a sequence of integers from the model  

    h is memory state, seed_ix is seed letter for first time step  

    """  

    if seed_ix is None:  

        seed_ix = 0  

    ixes = []  

    for t in xrange(n):  

        h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)  

        y = np.dot(Why, h) + by  

        p = np.exp(y) / np.sum(np.exp(y))  

        ix = np.random.choice(range(vocab_size), p=p.ravel())  

        x = np.zeros((vocab_size, 1))  

        x[ix] = 1  

        ixes.append(ix)  

    return ixes  

mesh, mesh, mesh, mesh = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)  

mbh, mbh, mbh, mbh = 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:  

    # fetch next inputs (we're moving from left to right in steps seq_length long)  

    if p+seq_length > len(data) or n == 0:  

        hprev = np.zeros((hidden_size, 1)) # reset RNN memory  

        p = 0 # 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 '----> %s ----' % (txt, )  

    # forward seq_length characters through the net and fetch gradient  

    loss, dех, dWhh, dWуh, dbh, dyh, hprev = lossfun(inputs, targets, hprev)  

    smooth_loss = smooth_loss * 0.999 + loss * 0.001  

    if n % 100 == 0: print 'iter %d, loss: %.3f' % (n, smooth_loss) # print progress  

    # perform parameter update with Adagrad  

    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],  

                                 [dех, dWhh, dWуh, dbh, dyh],  

                                 [mesh, mesh, mesh, mbh, mbh]):  

        mem += dparam * dparam  

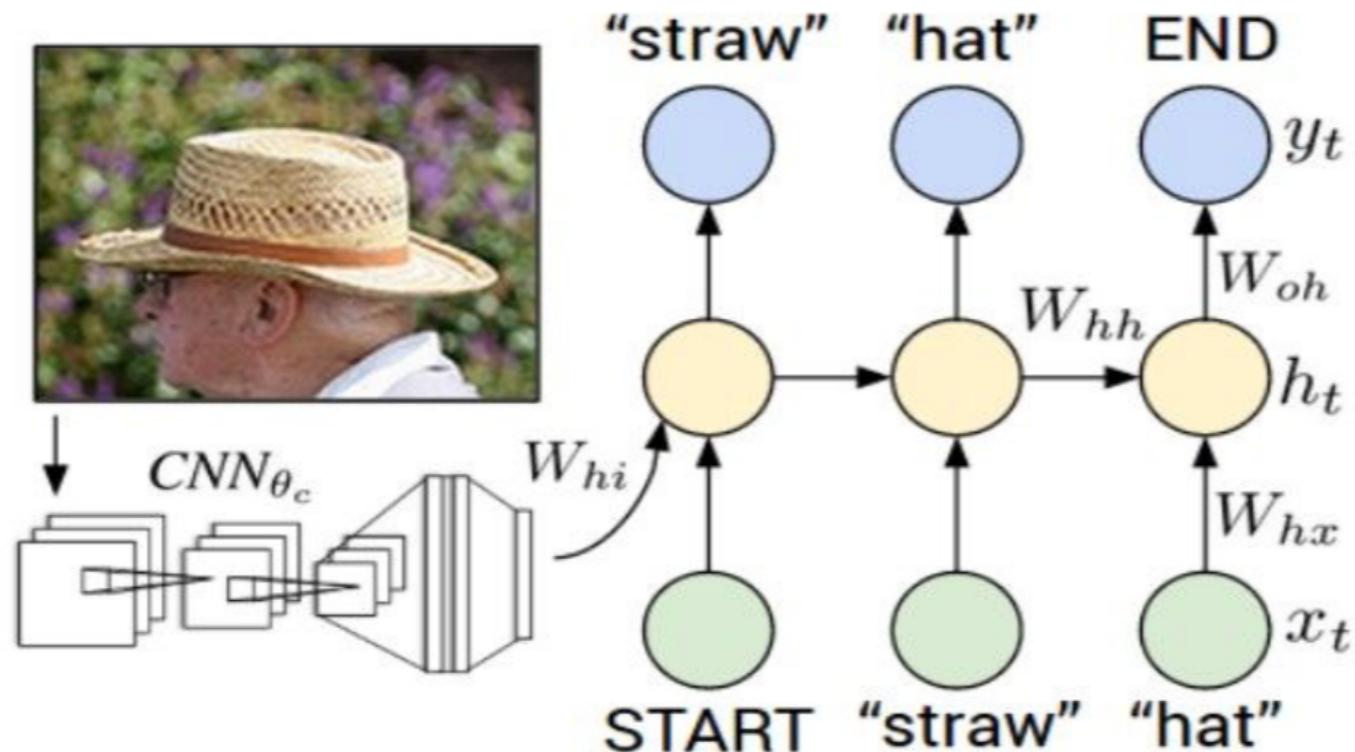
        param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update

```

```
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
```

# 图像说明

## Image Captioning



Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

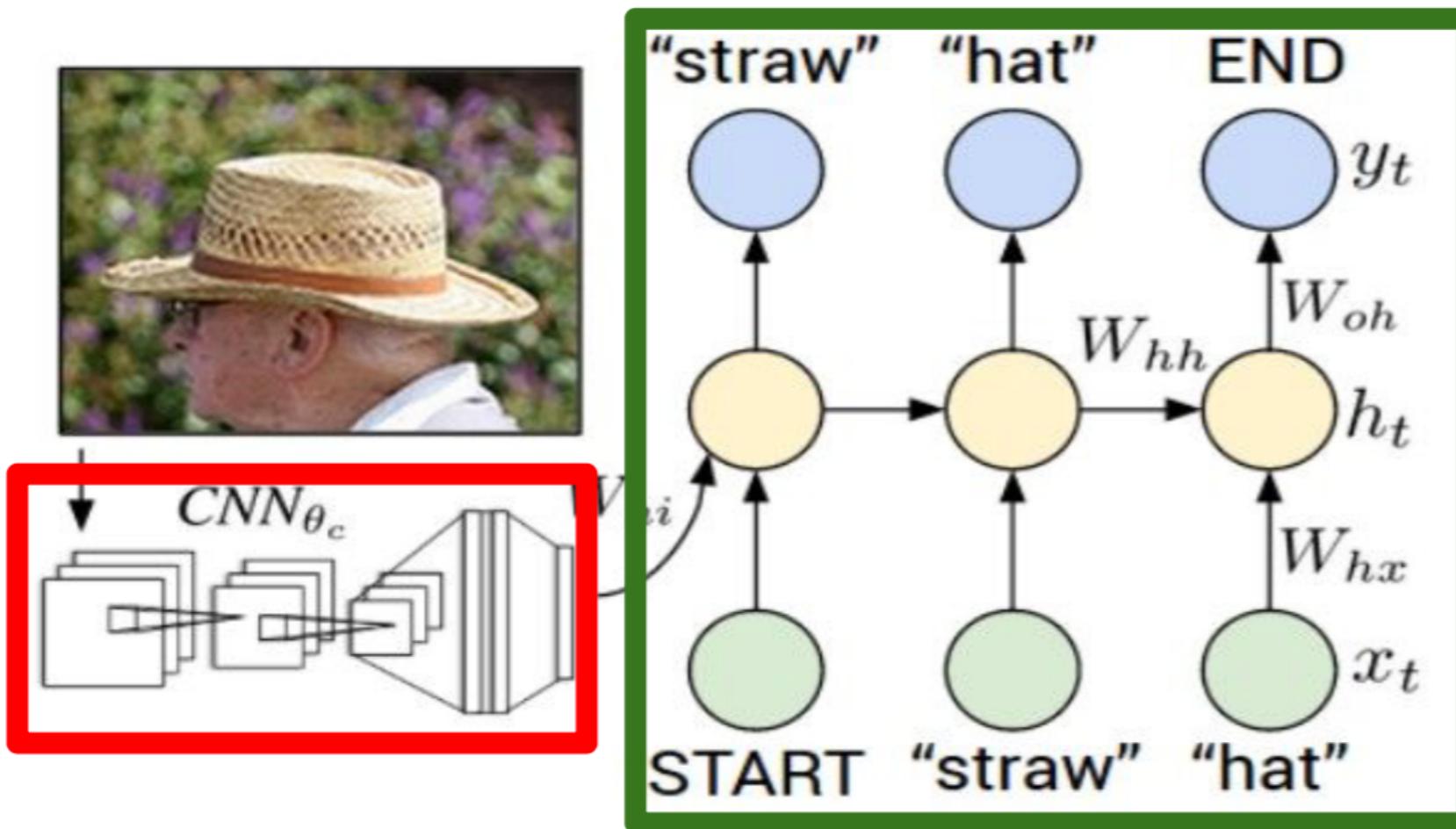
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

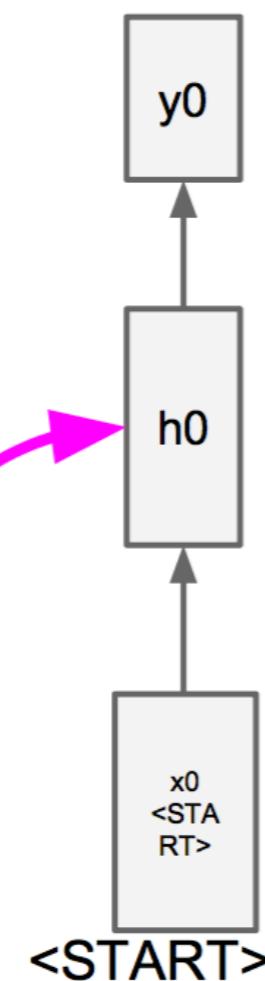
# Recurrent Neural Network



## Convolutional Neural Network



test image



**WiH**

V

**before:**

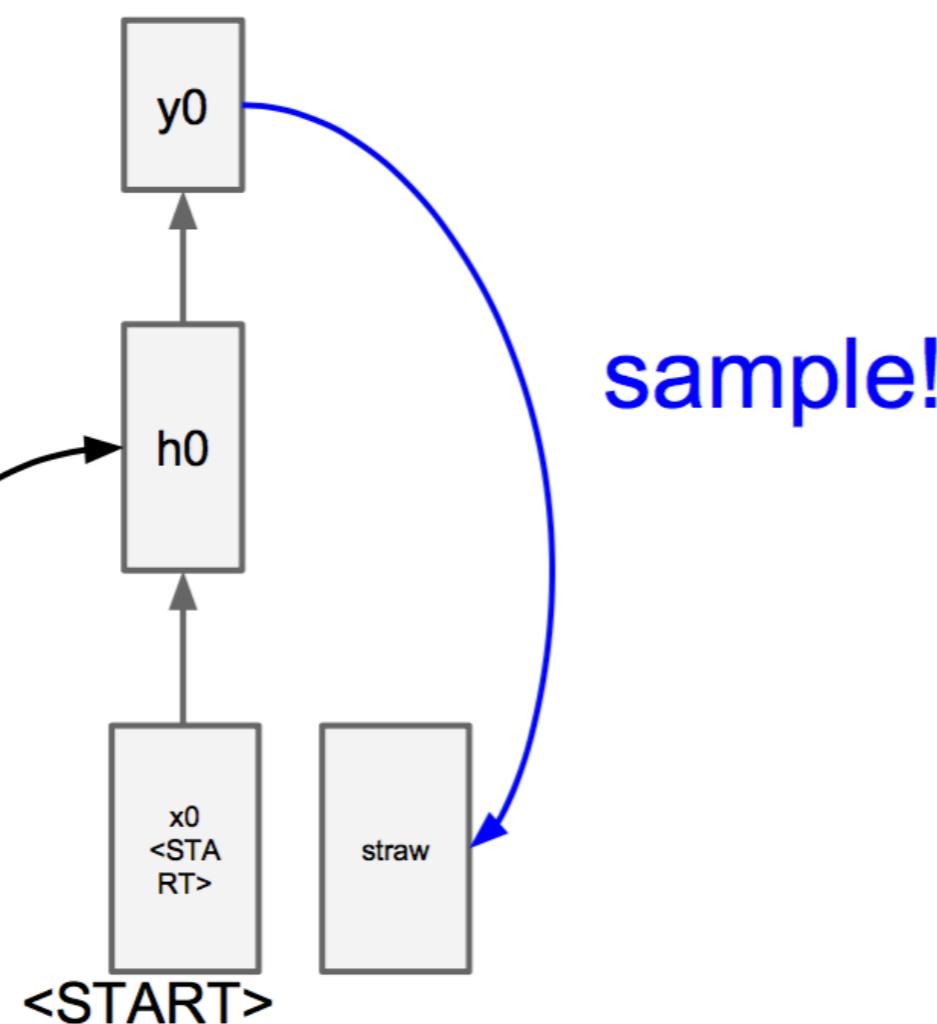
$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

**now:**

$$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$$

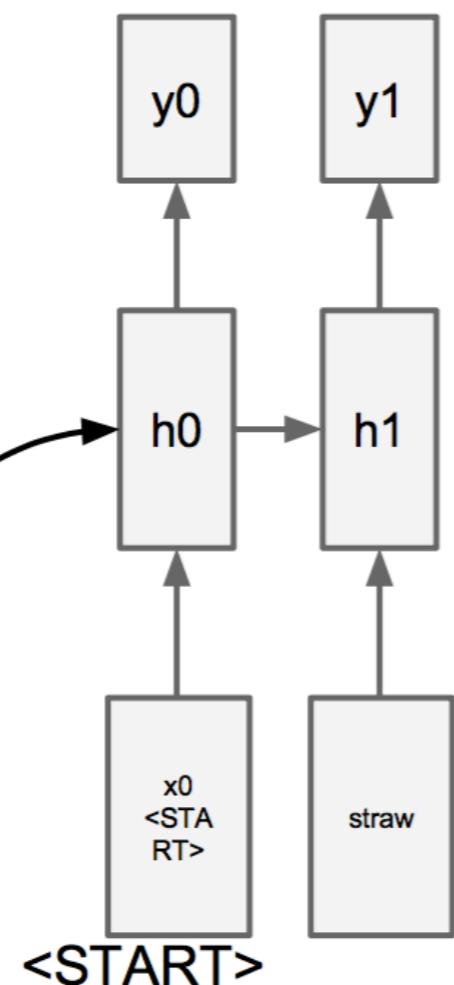


test image



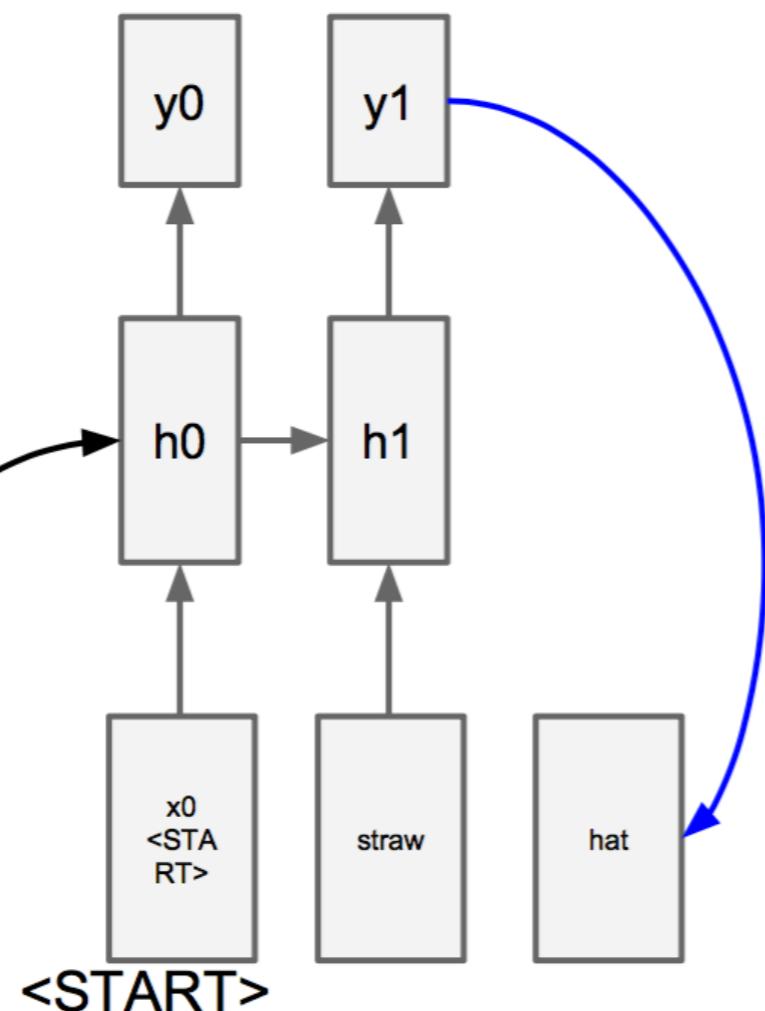


test image





test image



sample!

# Image Sentence Datasets

a man riding a bike on a dirt path through a forest.  
bicyclist raises his fist as he rides on desert dirt trail.  
this dirt bike rider is smiling and raising his fist in triumph.  
a man riding a bicycle while pumping his fist in the air.  
a mountain biker pumps his fist in celebration.



**Microsoft COCO**  
*[Tsung-Yi Lin et al. 2014]*  
[mscoco.org](http://mscoco.org)

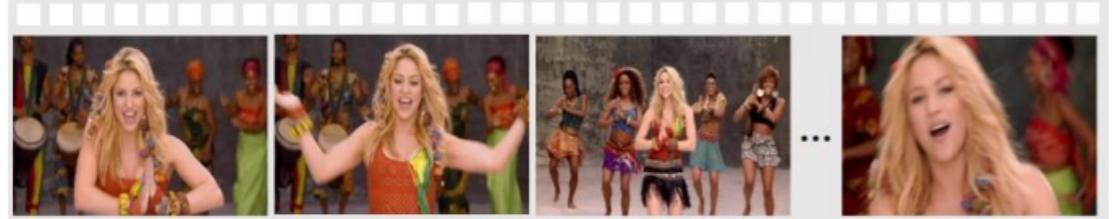
currently:  
~120K images  
~5 sentences each

# 视频描述



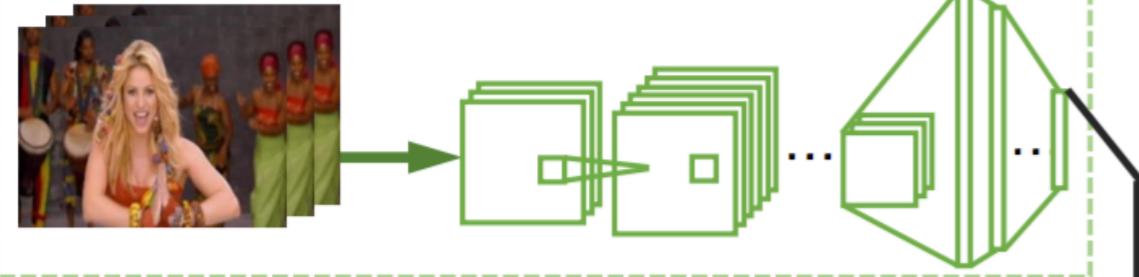
1. A player is putting the basketball into the post from distance.
2. The player makes a three-pointer.
3. People are playing basketball.
4. A 3 point shot by someone in a basketball race.
5. A basketball team is playing in front of speculators.

video



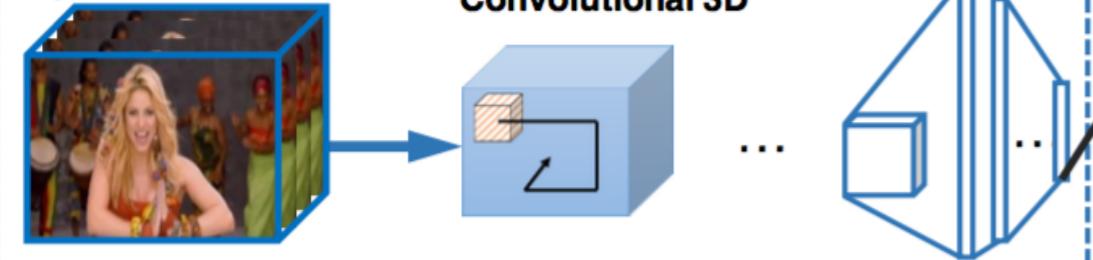
frames

2D CNN (AlexNet, GoogleNet, VGG)

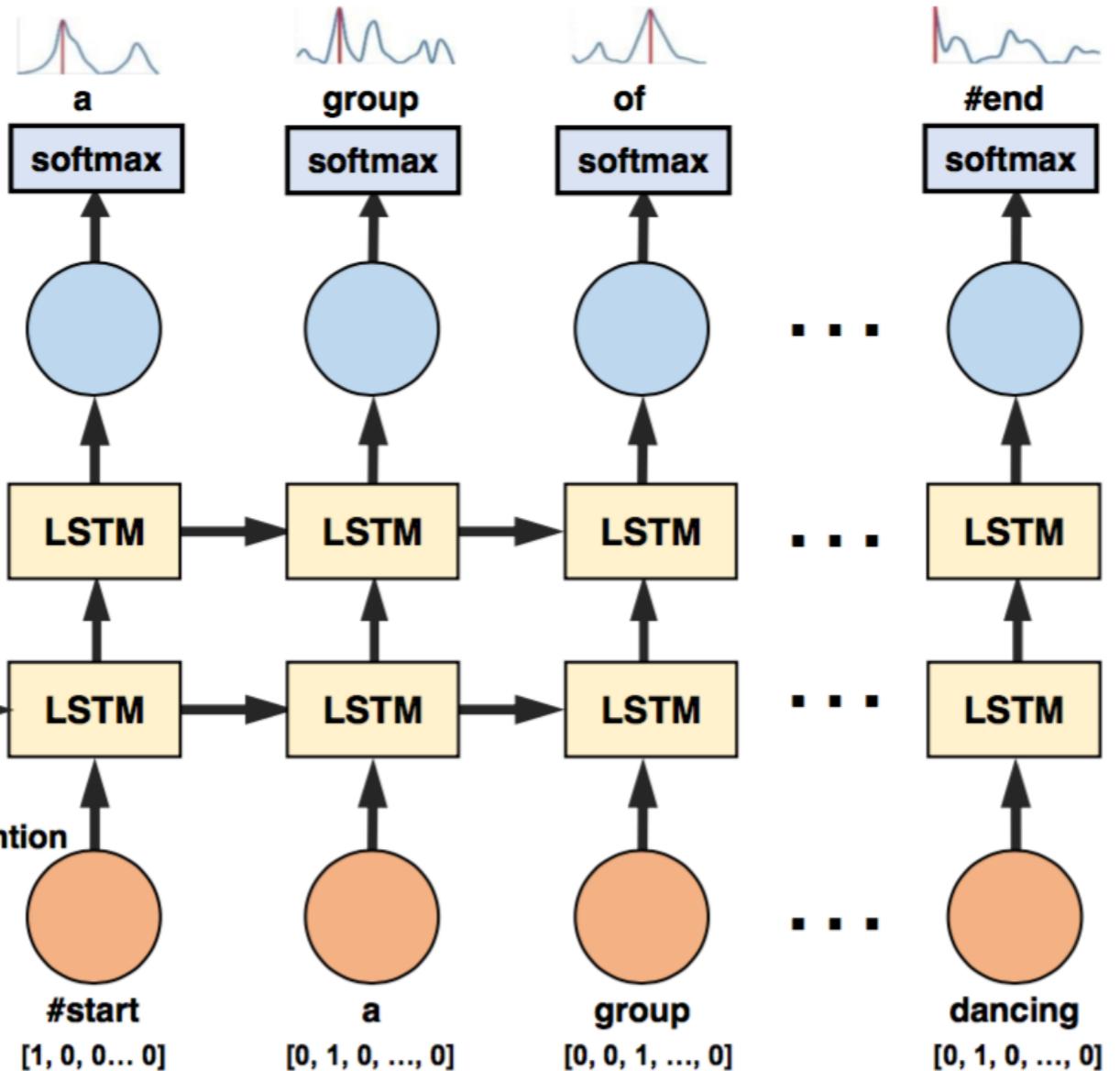


clip

Convolutional 3D



pooling:  
mean or  
soft-attention



<http://research.microsoft.com/apps/pubs/default.aspx?id=264836>

# 手写体生产

<http://www.cs.toronto.edu/~graves/handwriting.html>

<https://github.com/szcom/rnnlib>

Too simple, sometimes naive.

Too simple, sometimes naive

Too simple, sometimes naive