

RNN With Example

Shima Foolad

29 Oct. 2017

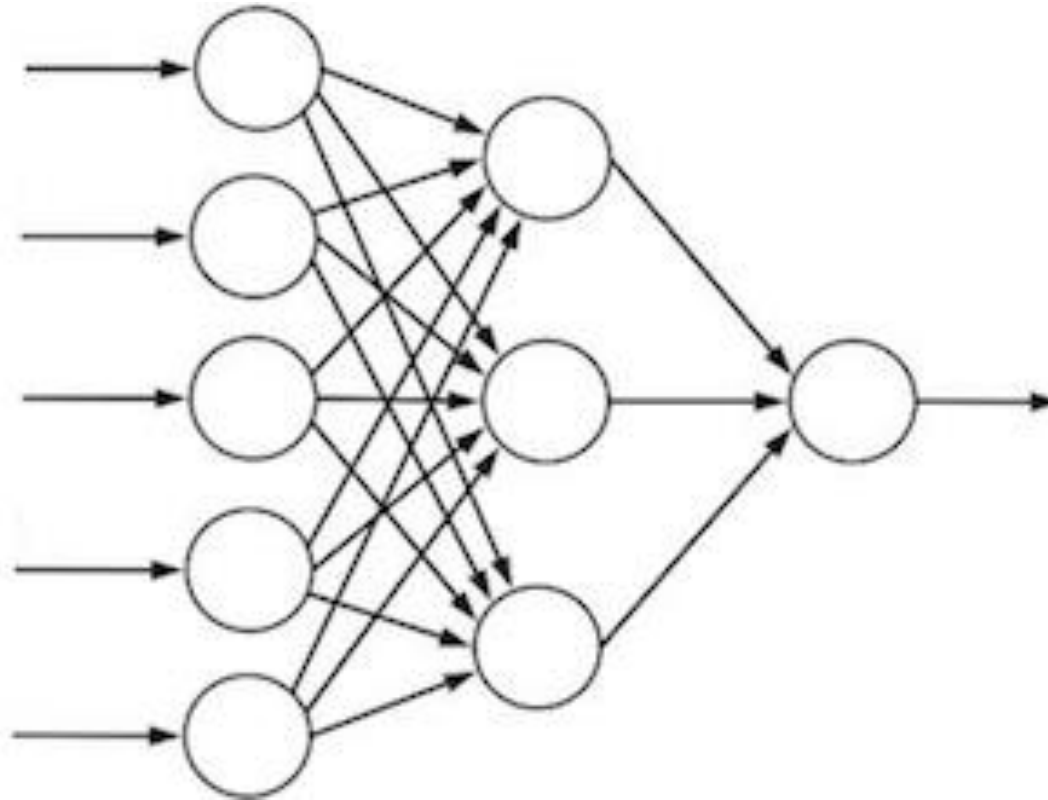
RECURRENT NEURAL NETWORK (RNN)

Rohan & Lenny #3: Recurrent Neural Networks & LSTMs

Fei-Fei Li & Andrej Karpathy & Justin Johnson: C321n, lecture 10

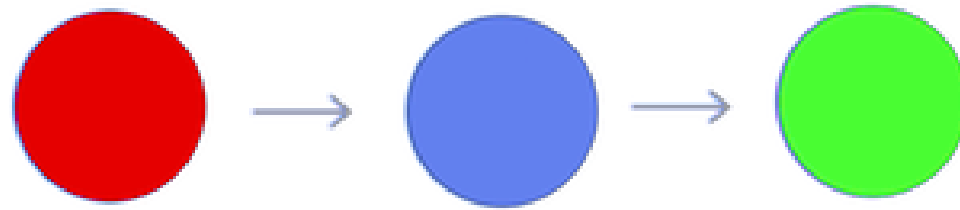
ARTIFICIAL NEURAL NETWORK (ANN)

Each neuron stores a single scalar value. Thus, each layer can be considered a vector.



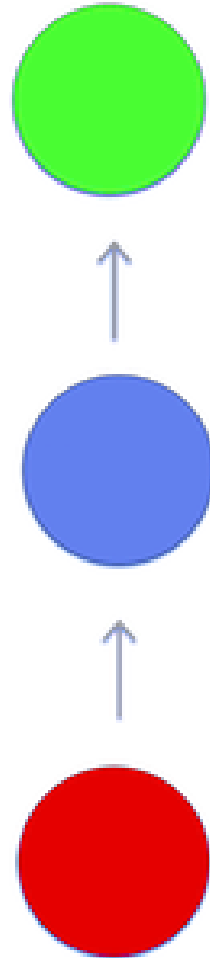
RECURRENT NEURAL NETWORK (RNN)

each neuron (inputs, hidden(s), and outputs) contains a **vector** of information. The term “cell” is also used, and is interchangeable with neuron.



RNN (ONE TO ONE)

This is in fact a type of recurrent neural network—a **one to one** recurrent net, because it maps one input to one output. A one to one recurrent net is equivalent to an artificial neural net.



EXAMPLE FOR RNN

**Character-level language
model example**

[min-char-rnn.py](https://gist.github.com/karpathy/d4dee566867f8291f086) 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     dwh, dwnh, dwhy = np.zeros_like(whh), np.zeros_like(wnh), 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         draw = {} # hs[t] * hs[t] * dh # backprop through tanh nonlinearity
55         dbh += draw
56         draw += np.dot(draw, xs[t-1].T)
57         dwnh += np.dot(draw, hs[t-1].T)
58         dhnext = np.dot(wnh.T, draw)
59     for dparam in [dwh, dwnh, dwhy, dbh, dby]:
60         np.clip(dparam, -1, 1, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwh, dwnh, 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 wnh, whh, why = np.zeros_like(wnh), np.zeros_like(whh), np.zeros_like(why)
83 whh, why = np.zeros_like(bh), np.zeros_like(bh) # memory variables for Adagrad
84 smooth_loss = -np.log(1.8/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 %s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dwh, dwnh, 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                                   [dwh, dwhh, dwhy, dbh, dby],
107                                   [wnh, whh, why, bh, by]):
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>)

AN EXAMPLE

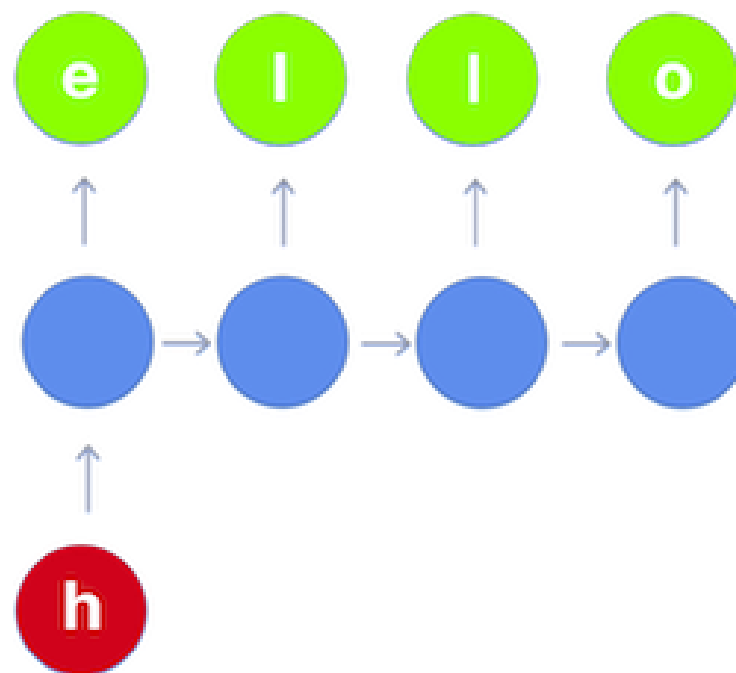
Character-level language model example

Vocabulary:
[h,e,l,o]

Example **training**
sequence:
“hello”

$$\text{"h"} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \text{"e"} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}; \text{"l"} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}; \text{"o"} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

This is what we'd expect with a trained RNN:



min-char-rnn.py gist

Data I/O

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

```
13 # hyperparameters
14 hidden_size = 64 # size of hidden layer of neurons
15 num_layers = 5 # number of layers to put in the RNN
16 learning_rate = 0.01
17
18 # model parameters
19 wif = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
20 wih = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
21 whf = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
22 wih_bias = np.zeros((hidden_size, 1)) # hidden bias
23 bh = np.zeros((hidden_size, 1)) # output bias
```

```
24 def loss_and_grads(targets, prev_h):
25     # inputs, targets are both lists of integers
26     forward_pass_and_loss(targets, prev_h)
27     return the loss, gradients on model parameters, and last hidden state
```

```
28 #
29 # forward pass
30 for i in range(len(targets)):
31     xof = np.zeros((vocab_size, 1)) # model input at time step i
32     hof = np.zeros((hidden_size, 1)) # hidden state
33     yof = np.zeros((1, 1)) # output at time step i
34     # calculate the loss and gradients for this time step
35     loss, grads, hof = rnn_step(xof, hof, targets[i])
36     # update the hidden state for the next time step
37     prev_h = hof
```

```
38 # calculate the total loss and gradients
39 total_loss = 0
40 for i in range(len(targets)):
41     loss, grads, hof = rnn_step(xof, hof, targets[i])
42     total_loss += loss
43     prev_h = hof
44 # return the total loss and gradients
45 return total_loss, grads, prev_h
```

```
46 # sample from the model
47 def sample(model, prev_h):
48     # sample from the model
49     xof = np.zeros((vocab_size, 1))
50     hof = np.zeros((hidden_size, 1))
51     for i in range(10):
52         xof, hof = rnn_step(xof, hof, None)
53         ix = np.argmax(hof[-1, :])
54         xof[-1, ix] = 1
55         hof = hof + grads[ix, :]
```

```
56 # sample from the model
57 def sample(model, prev_h):
58     # sample from the model
59     xof = np.zeros((vocab_size, 1))
60     hof = np.zeros((hidden_size, 1))
61     for i in range(10):
62         xof, hof = rnn_step(xof, hof, None)
63         ix = np.argmax(hof[-1, :])
64         xof[-1, ix] = 1
65         hof = hof + grads[ix, :]
```

```
66 # sample from the model
67 def sample(model, prev_h):
68     # sample from the model
69     xof = np.zeros((vocab_size, 1))
70     hof = np.zeros((hidden_size, 1))
71     for i in range(10):
72         xof, hof = rnn_step(xof, hof, None)
73         ix = np.argmax(hof[-1, :])
74         xof[-1, ix] = 1
75         hof = hof + grads[ix, :]
```

```
76 # sample from the model
77 def sample(model, prev_h):
78     # sample from the model
79     xof = np.zeros((vocab_size, 1))
80     hof = np.zeros((hidden_size, 1))
81     for i in range(10):
82         xof, hof = rnn_step(xof, hof, None)
83         ix = np.argmax(hof[-1, :])
84         xof[-1, ix] = 1
85         hof = hof + grads[ix, :]
```

```
86 # sample from the model
87 def sample(model, prev_h):
88     # sample from the model
89     xof = np.zeros((vocab_size, 1))
90     hof = np.zeros((hidden_size, 1))
91     for i in range(10):
92         xof, hof = rnn_step(xof, hof, None)
93         ix = np.argmax(hof[-1, :])
94         xof[-1, ix] = 1
95         hof = hof + grads[ix, :]
```

data is 'hello'

Vocabulary is
['h', 'e', 'l', 'o']

data size is 5
Vocabulary size is 4

index of each character:
{ 'l':2, 'o':3, 'e':1, 'h':0 }

character of each index :
{ 0:'h', 1:'e', 2:'l', 3:'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

Hidden size is 3

Number of timesteps is 4. it means each character is dependent on four character before it. (chunks of 4 characters in a time)

Learning rate is 0.1

w_{xh} is Weight matrix 3*4 between input layer and output layer

w_{hh} is Weight matrix 3*3 between two hidden layers

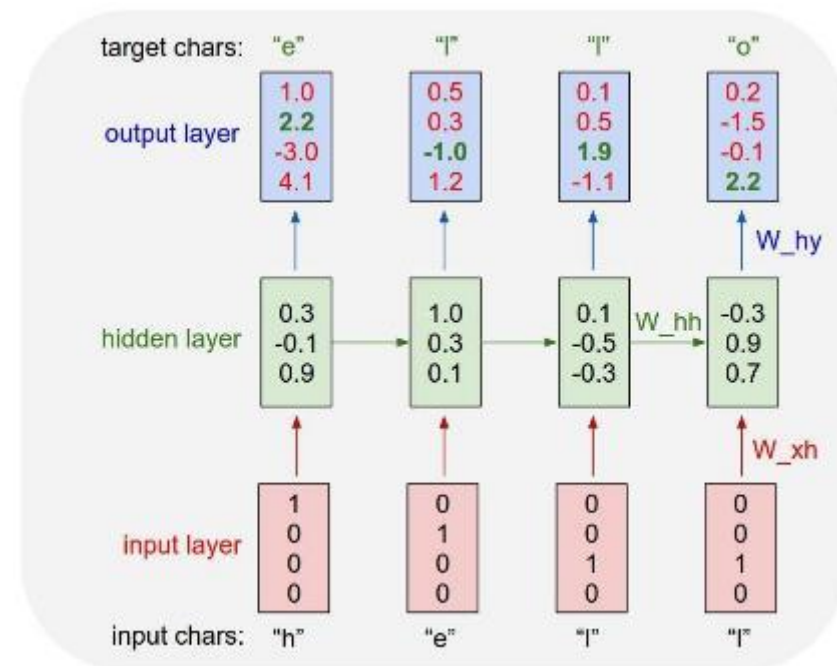
w_{hy} is Weight matrix 4*3 between hidden layer and output layer

b_h is hidden bias

b_y is output bias

```
15 # hyperparameters
16 hidden_size = 3 # size of hidden layer of neurons
17 seq_length = 4 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
```

recall:



min-char-rnn.py gist

Hidden size is 3

Number of timesteps or sequence length is 4. it means each character is dependent on four character before it. (chunks of 4 characters in a time)

Learning rate is 0.1

w_{xh} is Weight matrix 3*4 between input layer and output layer

w_{hh} is Weight matrix 3*3 between two hidden layers

w_{hy} is Weight matrix 4*3 between hidden layer and output layer

b_h is hidden bias

b_y is output bias

Initializations

```
15 # hyperparameters
16 hidden_size = 3 # size of hidden layer of neurons
17 seq_length = 4 # 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
```

$$W_{xh} = \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix}$$

$$W_{hh} = \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix}$$

$$W_{hy} = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix}$$

$$b_h = [0 \quad 0 \quad 0]$$

$$b_y = [0 \quad 0 \quad 0 \quad 0]$$


```

1 # min-char-rnn.py: Train the RNN model. Written by Andrew Senior (@senior1994)
2 # min-char-rnn.py
3 # min-char-rnn.py
4 import numpy as np
5
6 # data size
7 data_size, vocab_size = 100000, 10000
8 data = np.zeros((data_size, vocab_size))
9 char_to_ix = {ch: i for i, ch in enumerate(sorted(vocab))}
10 ix_to_char = {i: ch for i, ch in enumerate(sorted(vocab))}
11
12 # hyperparameters
13 hidden_size = 50 # size of hidden layer of neurons
14 num_layers = 10 # number of layers to use in the RNN
15 learning_rate = 0.01
16
17 # model parameters
18 w = np.random.randn(hidden_size, vocab_size) * 0.01 # input to hidden
19 wh = np.random.randn(hidden_size, hidden_size) * 0.01 # hidden to hidden
20 wb = np.random.randn(hidden_size, hidden_size) * 0.01 # hidden to output
21 b = np.zeros((vocab_size, 1)) # output bias
22 by = np.zeros((vocab_size, 1)) # output bias
23
24 def lossFun(inputs, targets, hprev):
25     # inputs, targets are both lists of integers
26     # forward pass
27     return loss, dWxh, dWhh, dWhy, dbh, dby, hprev
28
29 # sample from the model now and then
30 if n % 100 == 0:
31     sample_ix = sample(hprev, inputs[0], 200)
32     txt = ''.join(ix_to_char[ix] for ix in sample_ix)
33     print '----\n%s \n----' % (txt, )
34
35 # forward seq_length characters through the net and fetch gradient
36 loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
37 smooth_loss = smooth_loss * 0.999 + loss * 0.001
38 if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
39
40 # perform parameter update with Adagrad
41 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
42                               [dWxh, dWhh, dWhy, dbh, dby],
43                               [mWxh, mWhh, mWhy, mbh, mby]):
44     mem += dparam * dparam
45     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
46
47 p += seq_length # move data pointer
48 n += 1 # iteration counter

```

```

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

```

initial hidden vector 3*1 to zero

consider indices of data with sequence length as inputs

in our example, data is 'hello' and consider indices of 4 characters (sequence length is 4) of data as inputs ('hell'). then inputs are [0, 1, 2, 2]

```

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
98         # print progress
99         print('... %s' % txt, end='')
100         hprev = np.zeros((hidden_size,1)) # reset RNN memory
101
102     for param, dparam, mem in zip([Wxh, Wwh, Why, bh, by],
103                                   [dWxh, dWwh, dWhy, dbh, dby],
104                                   [mWxh, mWwh, mWhy, mbh, mby]):
105
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

```


initial hidden vector 3*1 to zero

consider indices of data with sequence length as inputs

the next characters of each character inputs are targets

in our example, input is 'hell', target is 'ello' and consider its indices .
then inputs is [1, 2, 2, 3]

```

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
98         # print progress
99         print(' %s' % txt, end='', flush=True)
100
101     # for param, dparam, mem in zip([Wxh, Wwh, Why, bh, by],
102                                     [dWxh, dWwh, dWhy, dbh, dby],
103                                     [mWxh, mWwh, mWhy, mbh, mby]):
104
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

```

[illegible][illegible]

```

like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
np.zeros_like(by) # memory variables for Adagrad
ab_size)*seq_length # loss at iteration 0

eping from left to right in steps seq_length long
ta) or n == 0:
size,1)) # reset RNN memory
data
r ch in data[p:p+seq_length]]
or ch in data[p+1:p+seq_length+1]]

and then

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

ters through the net and fetch gradient
, dby, hprev = lossFun(inputs, targets, hprev)
0.999 + loss * 0.001
r %d, loss: %f' % (n, smooth_loss) # print progres

with Adagrad
ip([Wxh, Whh, Why, bh, by],
[dwxh, dwhh, dwhy, dbh, dby],
[mwxh, mwhh, mwhy, mbh, mby]):

dparam / np.sqrt(mem + 1e-8) # adagrad update

pointer

```

```
# perform parameter update with Adagrad
for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                               [dWxh, dWhh, dWhy, dbh, dby],
                               [mWxh, mWhh, 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
```


Loss function

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

```
1 # min-char-rnn.py: Simple RNN model, written by Andrew Senior (asr@fb.com)
2 # min-char-rnn
3 # min
4 import numpy as np
5
6 # data size
7 data_size, vocab_size = 10000, 10000
8 data = np.zeros((data_size, vocab_size))
9
10 # load data
11 # data is a list of characters, so we need to convert it to integers
12 # data_size = 10000, vocab_size = 10000
13 # data_size = 10000, vocab_size = 10000
14
15 # hyperparameters
16 hidden_size = 50 # size of hidden layer of neurons
17 num_layers = 10 # number of layers in the RNN
18 learning_rate = 0.01
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(hidden_size, hidden_size) * 0.01 # hidden to output
24 Wbh = np.random.randn(hidden_size, 1) * 0.01 # bias to hidden
25 bby = np.zeros((vocab_size, 1)) # output bias
```

```
26 def lossFun(inputs, targets, hprev):
27     """
28     inputs, targets are both list of integers.
29     hprev is Hx1 array of initial hidden state
30     returns the loss, gradients on model parameters, and last hidden state
31     """
32     xs, hs, ys, ps = {}, {}, {}, {}
33     hs[-1] = np.copy(hprev)
34     loss = 0
35
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]) + Wbh) # hidden state
41         ys[t] = np.dot(Why, hs[t]) + bby # 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     dwsx, dwsh, dwy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
47     dbh, dbby = np.zeros_like(Wbh), np.zeros_like(bby)
48     dhnext = np.zeros_like(hs[0])
49
50     for t in reversed(xrange(len(inputs))):
51         dy = np.copy(ps[t])
52         dy[targets[t]] -= 1 # backprop into y
53         dwhy += np.dot(dy, hs[t].T)
54         dby += dy
55         dh = np.dot(Why.T, dy) + dhnext # backprop into h
56         dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
57         dbh += dhraw
58         dwsx += np.dot(dhraw, xs[t].T)
59         dwsh += np.dot(dhraw, hs[t-1].T)
60         dhnext = np.dot(Whh.T, dhraw)
61
62     for dparam in [dwsx, dwsh, dwy, dbh, dbby]:
63         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
64
65     return loss, dwsx, dwsh, dwy, dbh, dbby, hs[len(inputs)-1]
```

Forward pass

```
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
37     # forward pass
38     for t in xrange(len(inputs)):
39         xs[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + Wbh) # hidden state
42         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
```

```
44 # backward pass: compute gradients going backwards
45 dwsx, dwsh, dwy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46 dbh, dbby = np.zeros_like(Wbh), np.zeros_like(bby)
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     dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55     dbh += dhraw
56     dwsx += np.dot(dhraw, xs[t].T)
57     dwsh += np.dot(dhraw, hs[t-1].T)
58     dhnext = np.dot(Whh.T, dhraw)
```

```
59 for dparam in [dwsx, dwsh, dwy, dbh, dbby]:
60     np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61 return loss, dwsx, dwsh, dwy, dbh, dbby, hs[len(inputs)-1]
```

[illegible]

Convert our input character at this timestep to a one-hot vector

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$
$$p_t = \frac{e^{y_t}}{\sum_j e^{y_j}}$$

Softmax classifier

$$L_i = -\log(p_{target_i})$$

2016, STANFORD UNIVERSITY, C321N, LECTURE 10
FEI-FEI LI & ANDREJ KARPATHY & JUSTIN JOHNSON

FORWARD PASS

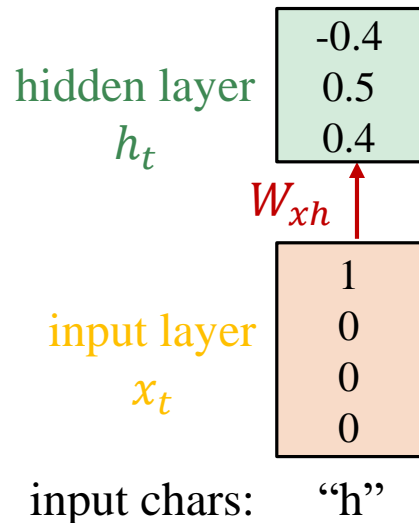
Example **training** sequence: “hello”

timestep 1:
(t=1)

$$W_{hh} = \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} \quad W_{xh} = \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix}$$

$$b_h = [0 \quad 0 \quad 0]$$

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



$$h_1 = \tanh \left(\begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right)$$
$$= \tanh \left(\begin{bmatrix} -0.5 \\ 0.5 \\ 0.4 \end{bmatrix} \right) = \begin{bmatrix} -0.4 \\ 0.5 \\ 0.4 \end{bmatrix}$$

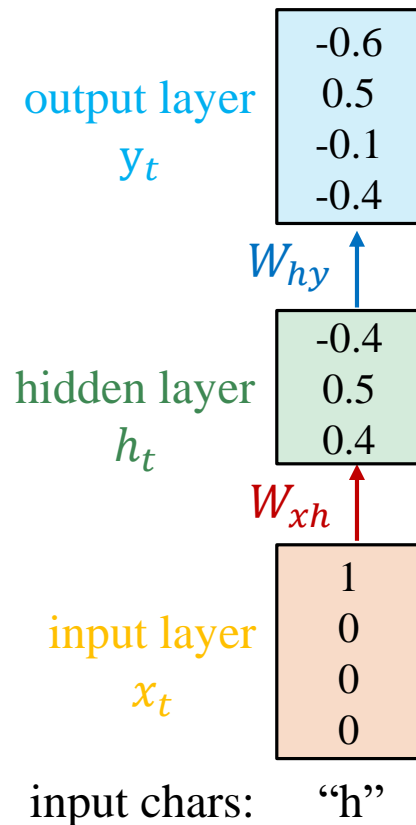
FORWARD PASS

Example **training** sequence: “**hello**”

timestep 1:
(t=1)

$$W_{hy} = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix}$$

$$b_y = [0 \quad 0 \quad 0 \quad 0]$$



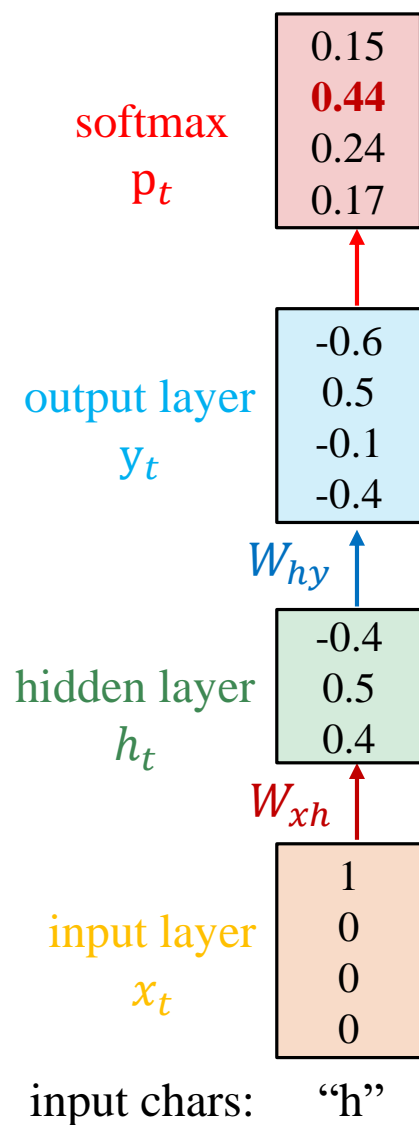
$$y_t = W_{hy} h_t + b_y$$

$$y_1 = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix} \begin{bmatrix} -0.4 \\ 0.5 \\ 0.4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.6 \\ 0.5 \\ -0.1 \\ -0.4 \end{bmatrix}$$

FORWARD PASS

Example **training** sequence: “hello”

timestep 1:
(t=1)

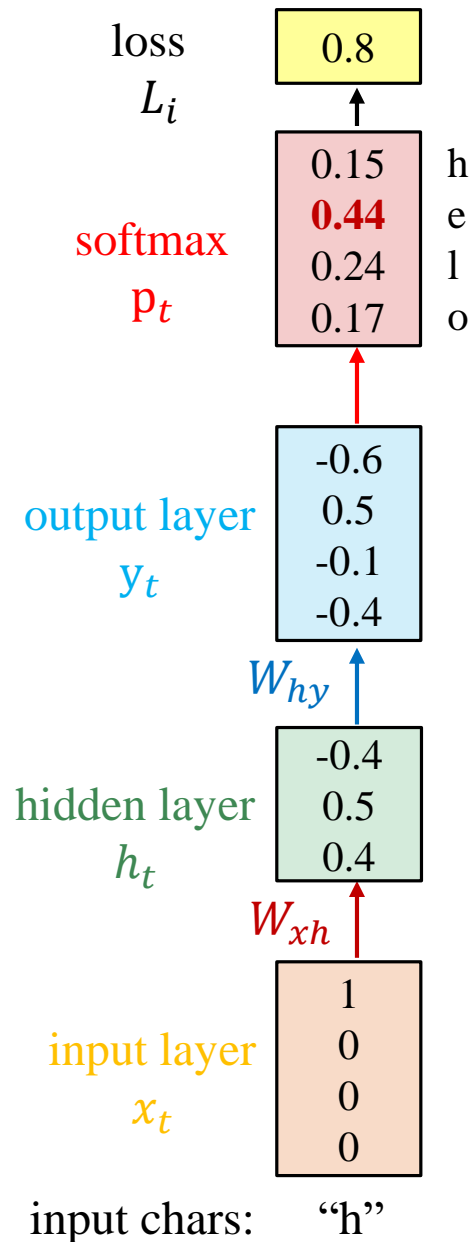


$$p_t = \frac{e^{y_t}}{\sum_j e^{y_j}}$$

FORWARD PASS

target chars: "e"

timestep 1:
(t=1)



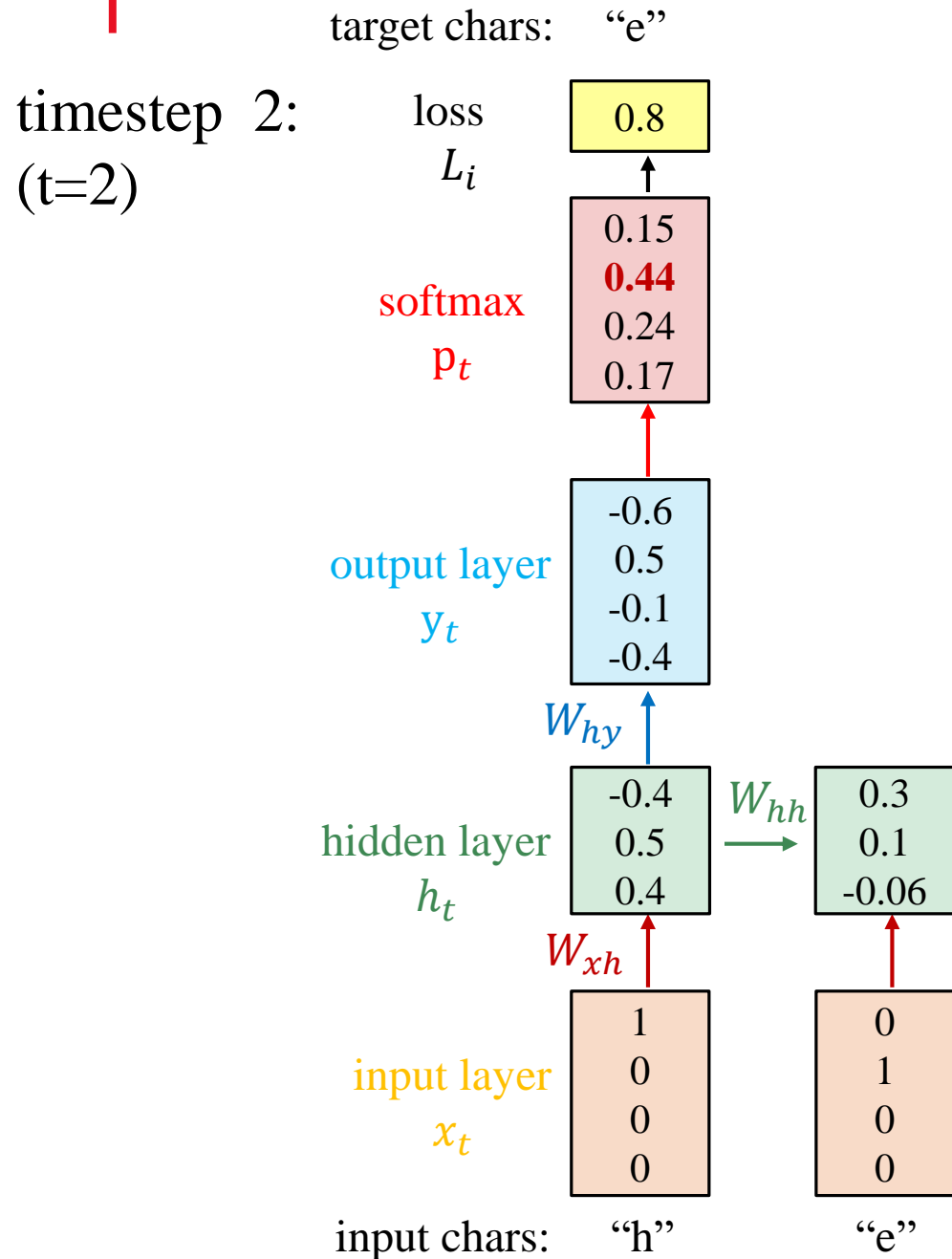
Example **training** sequence: "hello"

$$L_i = -\log(p_{target_i})$$

$$L_1 = -\log(\underbrace{0.44}) = 0.8$$

probability distribution of target char (e)

FORWARD PASS



Example **training** sequence: "hello"

$$W_{hh} = \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix}$$

$$W_{xh} = \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix}$$

$$b_h = [0 \quad 0 \quad 0]$$

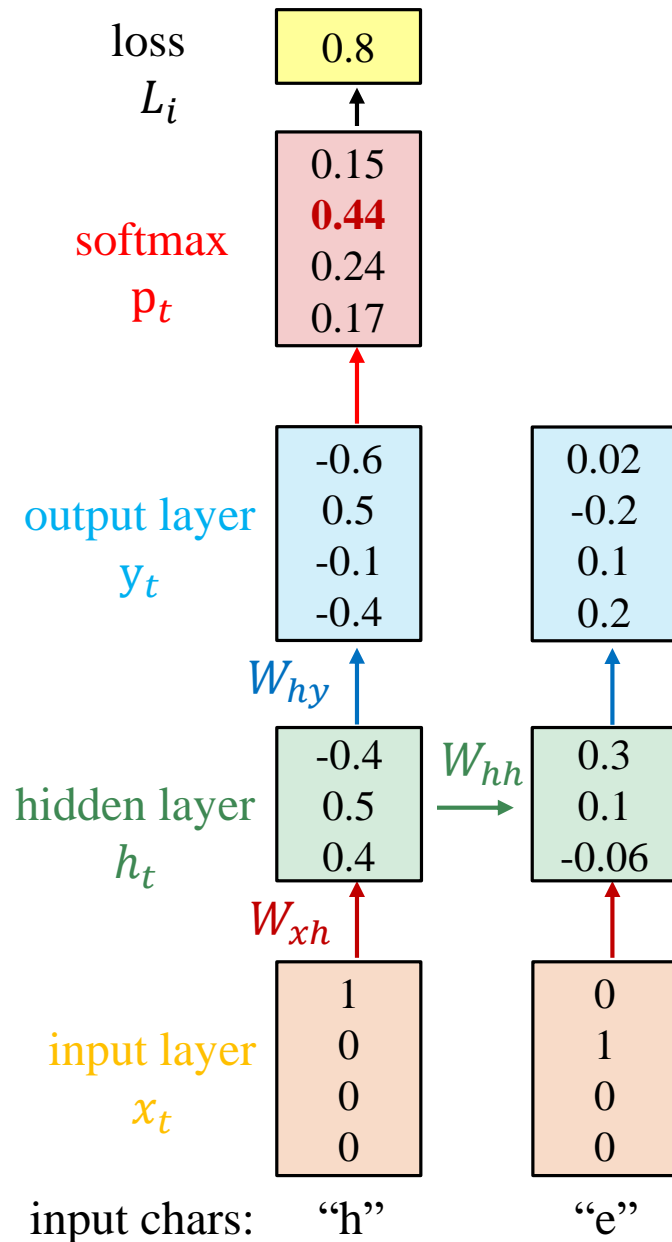
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$h_2 = \tanh \left(\begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} \begin{bmatrix} -0.4 \\ 0.5 \\ 0.4 \end{bmatrix} + \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right)$$
$$= \begin{bmatrix} 0.3 \\ 0.1 \\ -0.06 \end{bmatrix}$$

FORWARD PASS

target chars: "e"

timestep 2:
(t=2)



Example **training** sequence: "hello"

$$W_{hy} = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix}$$

$$b_y = [0 \quad 0 \quad 0 \quad 0]$$

$$y_t = W_{hy}h_t + b_y$$

$$y_2 = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.1 \\ -0.06 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.02 \\ -0.2 \\ 0.1 \\ 0.2 \end{bmatrix}$$

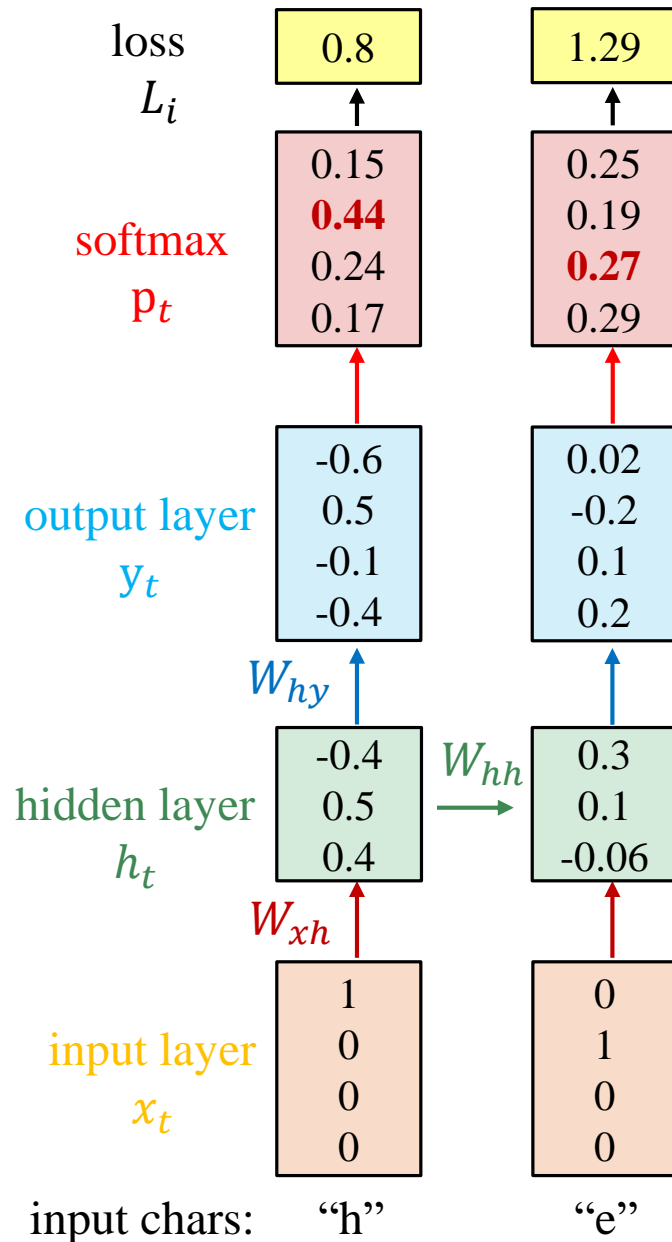
FORWARD PASS

Example **training** sequence: “hello”

timestep 2:
(t=2)

target chars: “e”

“l”



$$L_i = -\log(p_{target_i})$$

$$L_1 = -\log(\underbrace{0.27}) = 1.29$$

probability distribution of target char (l)

FORWARD PASS

Example **training** sequence: "hello"

timestep 3:
(t=3)

target chars: "e"

"l"

"l"

loss

L_i

0.8

1.29

1.19

softmax

p_t

0.15

0.44

0.24

0.17

0.25

0.19

0.27

0.29

0.16

0.36

0.3

0.18

output layer

y_t

-0.6

0.5

-0.1

-0.4

0.02

-0.2

0.1

0.2

-0.2

0.5

0.3

-0.1

W_{hy}

hidden layer

h_t

-0.4

0.5

0.4

W_{hh}

0.3

0.1

-0.06

W_{hh}

-0.4

0.5

-0.2

W_{xh}

input layer

x_t

1

0

0

0

0

1

0

0

0

0

1

0

input chars:

"h"

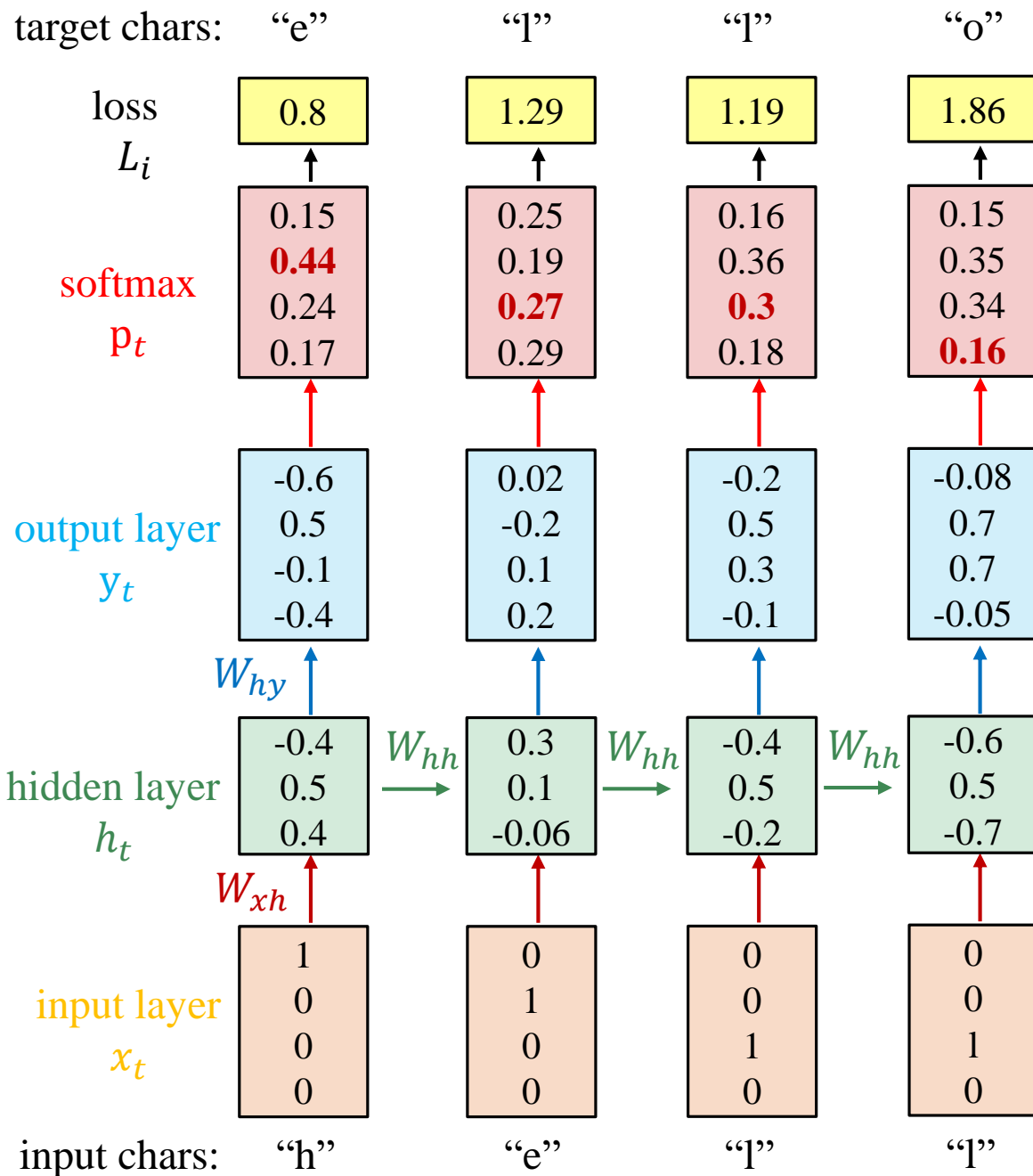
"e"

"l"

FORWARD PASS

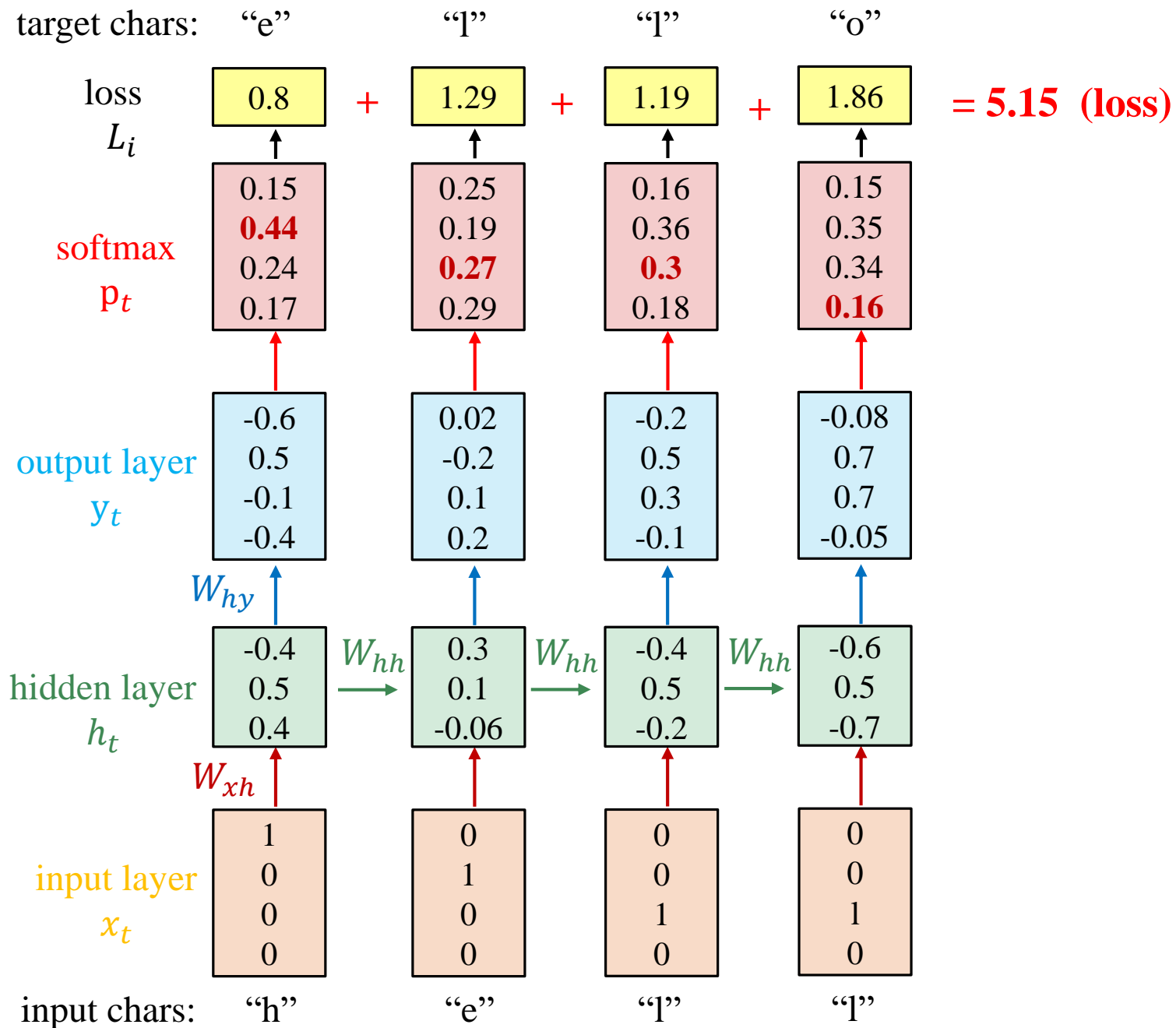
Example **training** sequence: "hello"

timestep 4:
(t=4)



FORWARD PASS

Example **training** sequence: "hello"



Loss function

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

```
1 # min-char-rnn.py: train the RNN model. Modified by Andrew Senior (asr@fb.com)
2 # min-char-rnn.py
3 #
4 # Imports
5 import numpy as np
6
7 # Constants
8 vocab_size = 256 # size of the vocabulary
9 num_embeddings = 256 # number of embeddings
10 num_hidden = 128 # number of hidden units
11 num_layers = 1 # number of layers
12 num_epochs = 10 # number of epochs
13 num_batches = 100 # number of batches per epoch
14 num_workers = 4 # number of workers
15 cuda_device = -1 # cuda device to use (-1 for cpu)
16 cuda_visible_devices = '0,1' # cuda devices to use
17
18 # Parameters
19 Wxh = np.zeros((vocab_size, num_hidden)) # input to hidden
20 Wwh = np.zeros((num_hidden, num_hidden)) # hidden to hidden
21 Why = np.zeros((num_hidden, vocab_size)) # hidden to output
22 Wb = np.zeros((vocab_size, 1)) # bias
23 b = np.zeros((vocab_size, 1)) # bias
24
25 # Loss function
```

```
26 def lossFun(inputs, targets, hprev):
27     """
28     inputs, targets are both list of integers.
29     hprev is Hx1 array of initial hidden state
30     returns the loss, gradients on model parameters, and last hidden state
31     """
32     xs, hs, ys, ps = {}, {}, {}, {}
33     hs[-1] = np.copy(hprev)
34     loss = 0
35
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(Wwh, hs[t-1]) + b) # hidden state
41         ys[t] = np.dot(Why, hs[t]) + b # 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, dWwh, dWhy = np.zeros_like(Wxh), np.zeros_like(Wwh), np.zeros_like(Why)
47     dbh, db = np.zeros_like(b), np.zeros_like(b)
48     dhnext = np.zeros_like(hs[0])
49
50     for t in reversed(xrange(len(inputs))):
51         dy = np.copy(ps[t])
52         dy[targets[t]] -= 1 # backprop into y
53         dWhy += np.dot(dy, hs[t].T)
54         dby += dy
55         dh = np.dot(Why.T, dy) + dhnext # backprop into h
56         dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
57         dbh += dhraw
58         dWxh += np.dot(dhraw, xs[t].T)
59         dWwh += np.dot(dhraw, hs[t-1].T)
60         dhnext = np.dot(Wwh.T, dhraw)
61
62     for dparam in [dWxh, dWwh, dWhy, dbh, db]:
63         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
64
65     return loss, dWxh, dWwh, dWhy, dbh, db, hs[len(inputs)-1]
```

Backward pass

```
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
37     # forward pass
38     for t in xrange(len(inputs)):
39         xs[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Wwh, hs[t-1]) + b) # hidden state
42         ys[t] = np.dot(Why, hs[t]) + b # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
45
46     # backward pass: compute gradients going backwards
47     dWxh, dWwh, dWhy = np.zeros_like(Wxh), np.zeros_like(Wwh), np.zeros_like(Why)
48     dbh, db = np.zeros_like(b), np.zeros_like(b)
49     dhnext = np.zeros_like(hs[0])
50
51     for t in reversed(xrange(len(inputs))):
52         dy = np.copy(ps[t])
53         dy[targets[t]] -= 1 # backprop into y
54         dWhy += np.dot(dy, hs[t].T)
55         dby += dy
56         dh = np.dot(Why.T, dy) + dhnext # backprop into h
57         dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
58         dbh += dhraw
59         dWxh += np.dot(dhraw, xs[t].T)
60         dWwh += np.dot(dhraw, hs[t-1].T)
61         dhnext = np.dot(Wwh.T, dhraw)
62
63     for dparam in [dWxh, dWwh, dWhy, dbh, db]:
64         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
65
66     return loss, dWxh, dWwh, dWhy, dbh, db, hs[len(inputs)-1]
```

Calculating the Gradient: Backpropagation



$$\frac{\partial J(\theta)}{\partial W_2} =$$

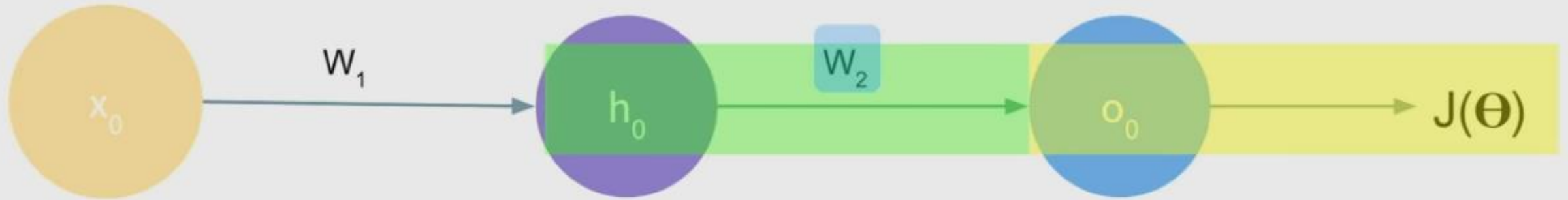
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_2} = \frac{\partial J(\theta)}{\partial o_0}$$

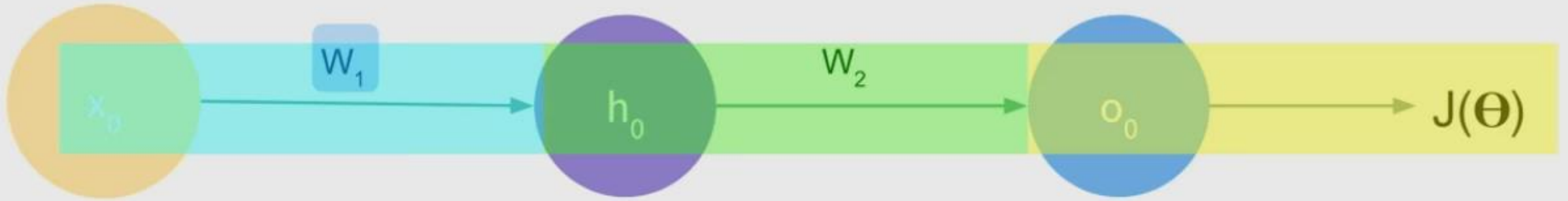
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_2} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial W_2}$$

Calculating the Gradient: Backpropagation



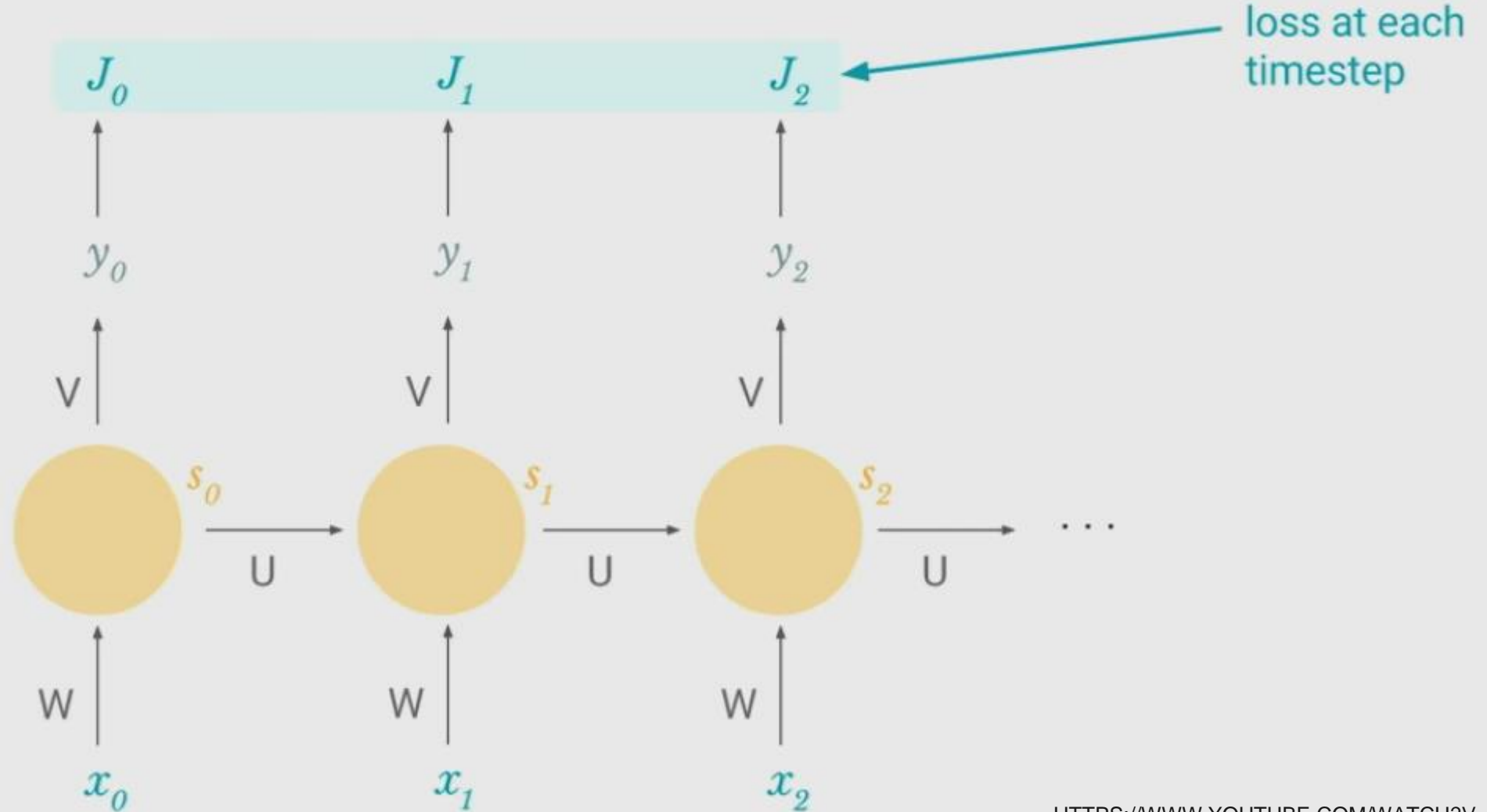
Apply the chain rule

Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_1} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial h_0} * \frac{\partial h_0}{\partial W_1}$$

we have a **loss at each timestep**:

(since we're making a prediction at each timestep)



we **sum the losses** across time:

$$\text{loss at time } t = J_t(\Theta)$$



Θ = our
parameters, like
weights

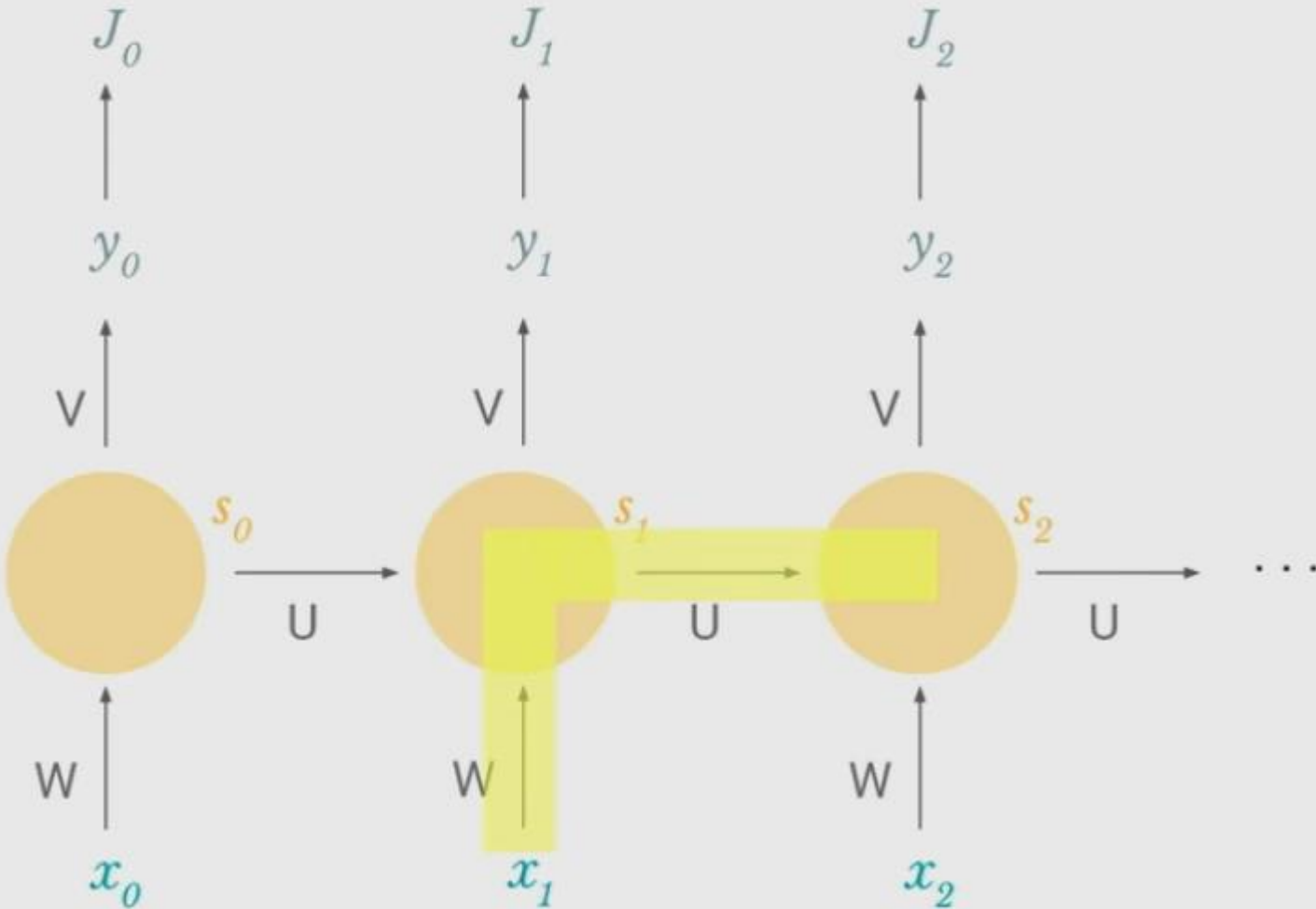
$$\text{total loss} = J(\Theta) = \sum_t J_t(\Theta)$$

what are our **gradients**?

we sum gradients across time for each
parameter P :

$$\frac{\partial J}{\partial P} = \sum_t \frac{\partial J_t}{\partial P}$$

let's try it out for W with the **chain rule**:



$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep t :

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial W}$$

but wait...

$$s_2 = \tanh(U s_1 + W x_2)$$

s_1 also depends on W so we can't just treat $\frac{\partial s_2}{\partial W}$ as a constant!

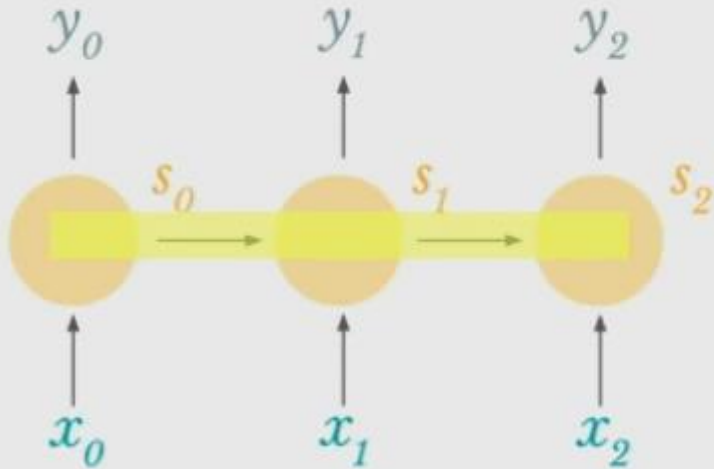
backpropagation through time:

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^2 \underbrace{\frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}}_{\text{Contributions of } W \text{ in previous timesteps to the error at timestep } t}$$

Contributions of W in previous timesteps to the error at timestep t

problem: vanishing gradient

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^2 \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}$$

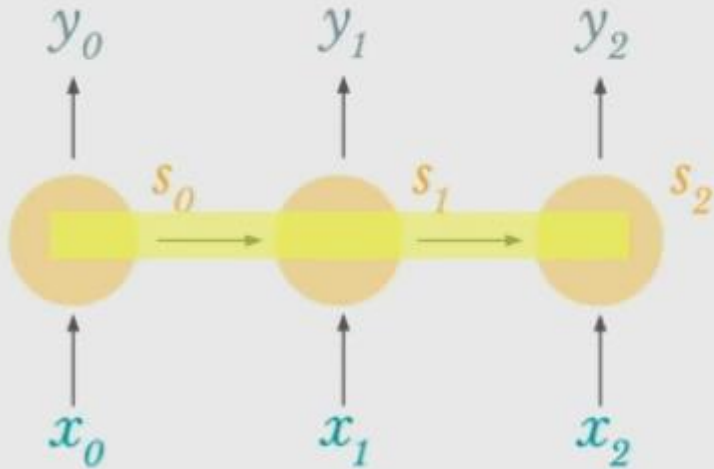


at $k = 0$:

$$\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

problem: vanishing gradient

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^2 \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}$$



at $k = 0$:

$$\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

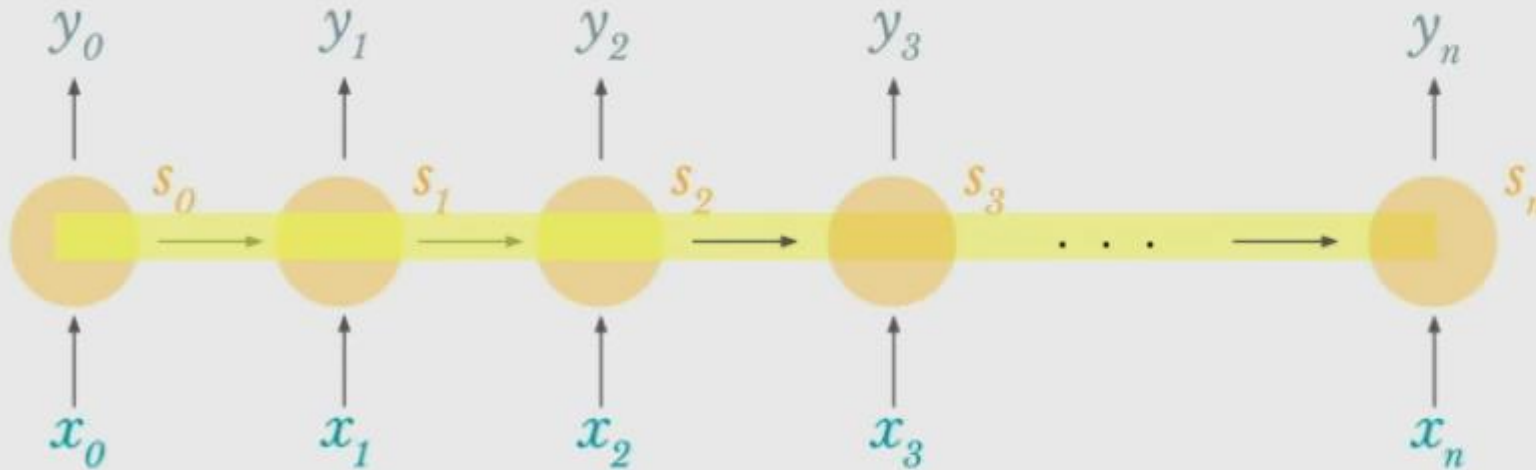
problem: vanishing gradient

at $k=0$

$$\frac{\partial J_n}{\partial W} = \sum_{k=0}^n \frac{\partial J_n}{\partial y_n} \frac{\partial y_n}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

as the gap between timesteps gets bigger, this product gets longer and longer!



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

Go through our sequence in reverse as we back up the gradients

gradient of the output

gradient of the W_{hy}

gradient of the b_y

gradient of the h_t

gradient of the $\tanh(h)_t$

gradient of the b_h

gradient of the W_{xh}

gradient of the W_{hh}

```
44 # backward pass compute gradients going backwards
45 dwdx, 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     dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55     dbh += dhraw
56     dwdx += np.dot(dhraw, xs[t].T)
57     dwhh += np.dot(dhraw, hs[t-1].T)
58     dhnext = np.dot(Whh.T, dhraw)
59     for dparam in [dwdx, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwdx, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

BACKWARD PASS

$$\frac{\partial L}{\partial y} = \sum_t \frac{\partial L_t}{\partial y_t}$$

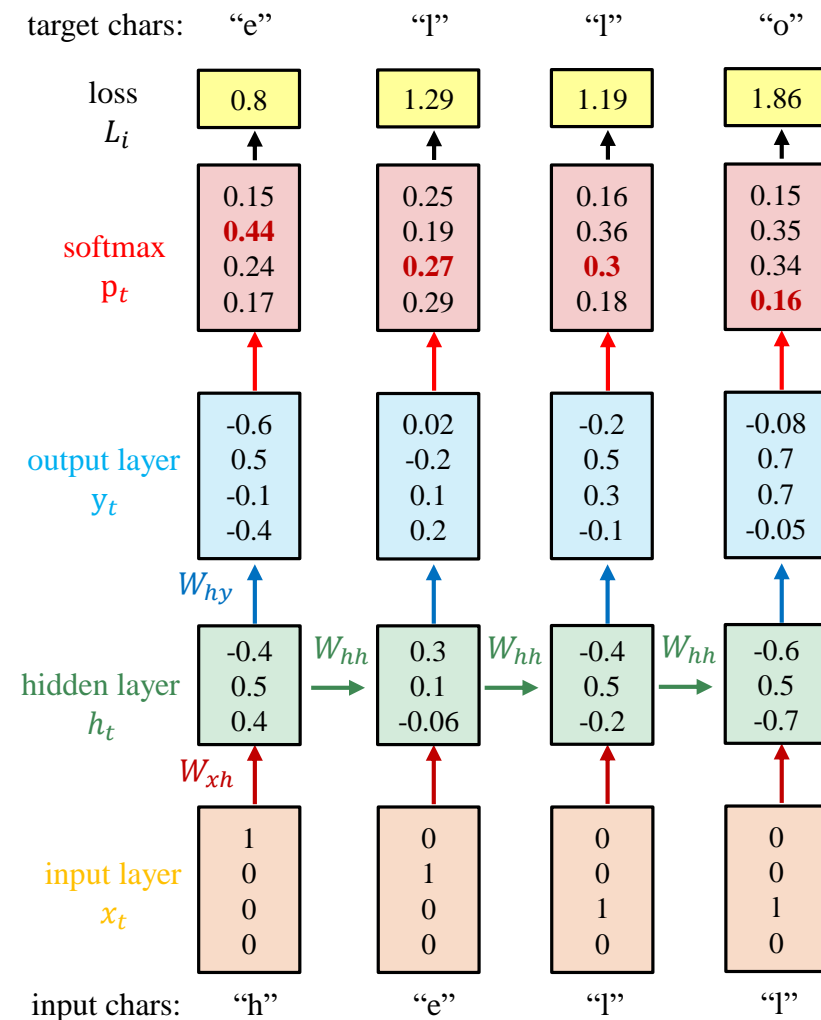
$$\frac{\partial L_t}{\partial y_t} = \text{output} - \text{target}$$

$$\frac{\partial L_4}{\partial y_4} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ 0.16 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ -0.84 \end{bmatrix}$$

$$\frac{\partial L_3}{\partial y_3} = \begin{bmatrix} 0.16 \\ 0.36 \\ 0.3 \\ 0.18 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.16 \\ 0.36 \\ -0.7 \\ 0.18 \end{bmatrix}$$

$$\frac{\partial L_2}{\partial y_2} = \begin{bmatrix} 0.25 \\ 0.19 \\ 0.27 \\ 0.29 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.15 \\ 0.19 \\ -0.73 \\ 0.29 \end{bmatrix}$$

$$\frac{\partial L_1}{\partial y_1} = \begin{bmatrix} 0.15 \\ 0.44 \\ 0.24 \\ 0.17 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.15 \\ -0.56 \\ 0.24 \\ 0.17 \end{bmatrix}$$



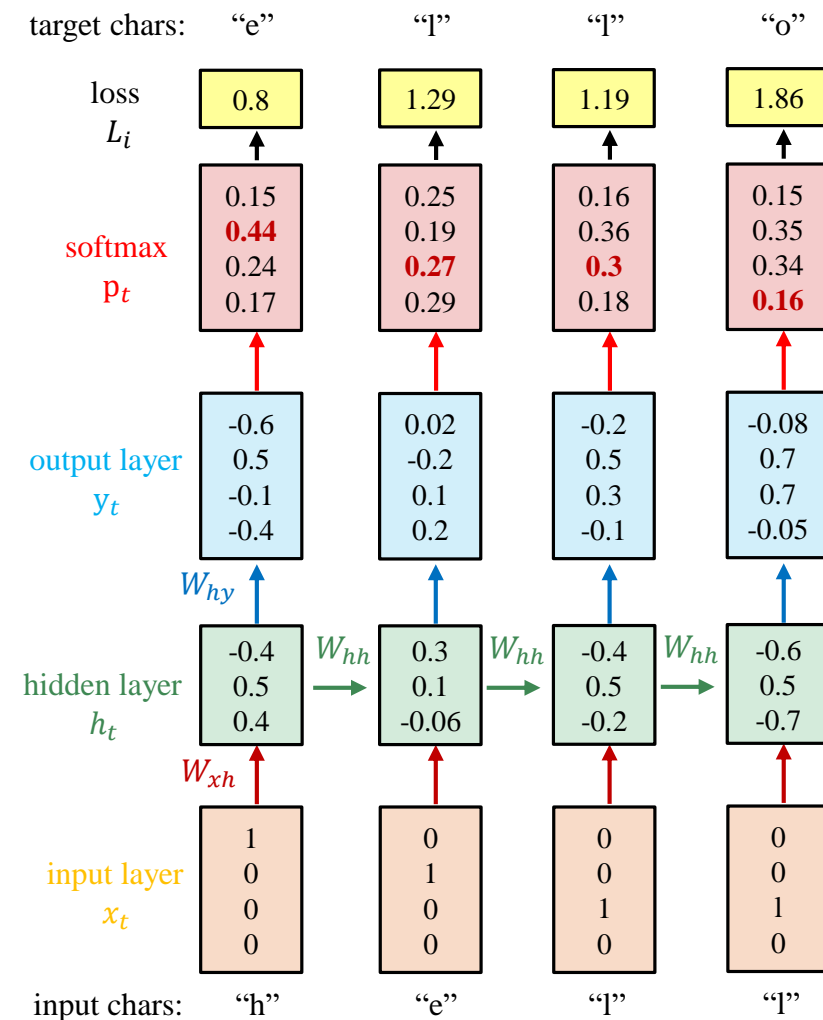
BACKWARD PASS

$$\frac{\partial L}{\partial W_{hy}} = \sum_t \frac{\partial L}{\partial y_t} \underbrace{\frac{\partial y_t}{\partial W_{hy}}}_{h_t}$$

$$\frac{\partial L}{\partial W_{hy}} = \frac{\partial L}{\partial y_4} h_4 + \frac{\partial L}{\partial y_3} h_3 + \frac{\partial L}{\partial y_2} h_2 + \frac{\partial L}{\partial y_1} h_1$$

$$\frac{\partial L}{\partial W_{hy}} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ -0.84 \end{bmatrix} [-0.6 \quad 0.5 \quad -0.7] + \begin{bmatrix} 0.16 \\ 0.36 \\ -0.7 \\ 0.18 \end{bmatrix} [-0.4 \quad 0.5 \quad -0.2]$$

$$+ \begin{bmatrix} 0.15 \\ 0.19 \\ -0.73 \\ 0.29 \end{bmatrix} [0.3 \quad 0.1 \quad -0.06] + \begin{bmatrix} 0.15 \\ -0.56 \\ 0.24 \\ 0.17 \end{bmatrix} [-0.4 \quad 0.5 \quad 0.4] = \begin{bmatrix} 0.1 & 0.2 & -0.1 \\ -0.06 & 0.1 & -0.5 \\ -0.2 & -0.1 & 0.05 \\ 0.5 & -0.2 & 0.6 \end{bmatrix}$$

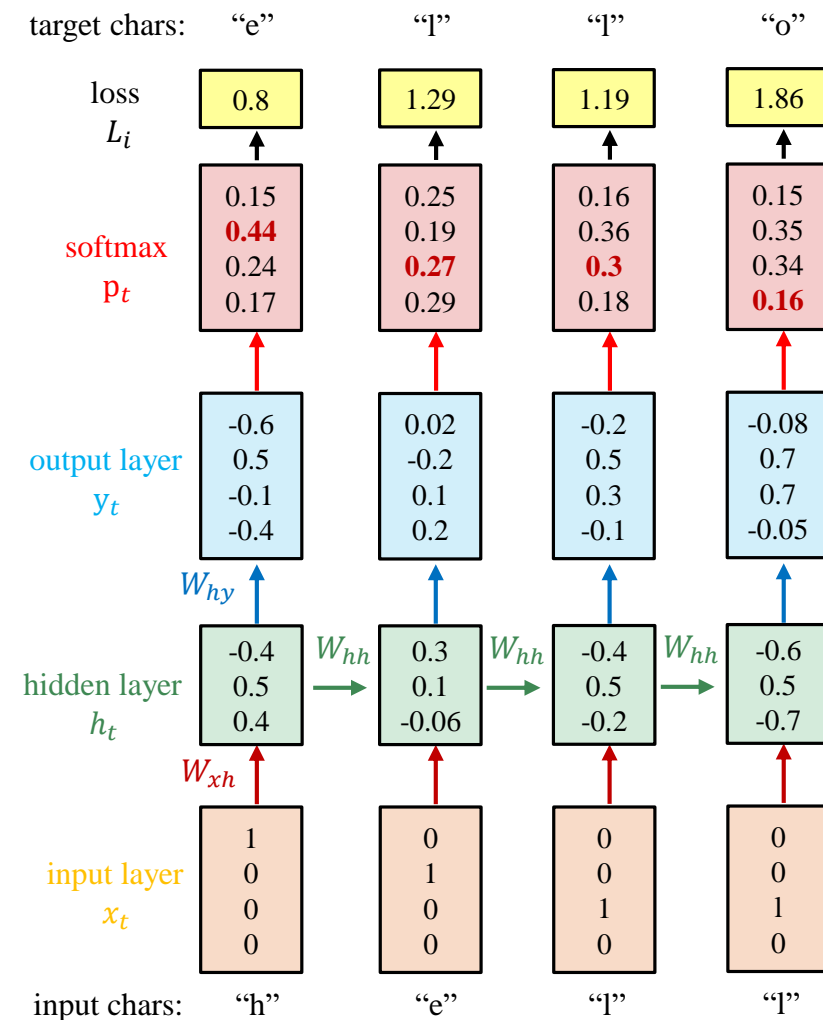


BACKWARD PASS

$$\frac{\partial L}{\partial b_y} = \sum_t \frac{\partial L}{\partial y_t} \underbrace{\frac{\partial y_t}{\partial b_y}}_1$$

$$\frac{\partial L}{\partial b_y} = \frac{\partial L}{\partial y_4} + \frac{\partial L}{\partial y_3} + \frac{\partial L}{\partial y_2} + \frac{\partial L}{\partial y_1}$$

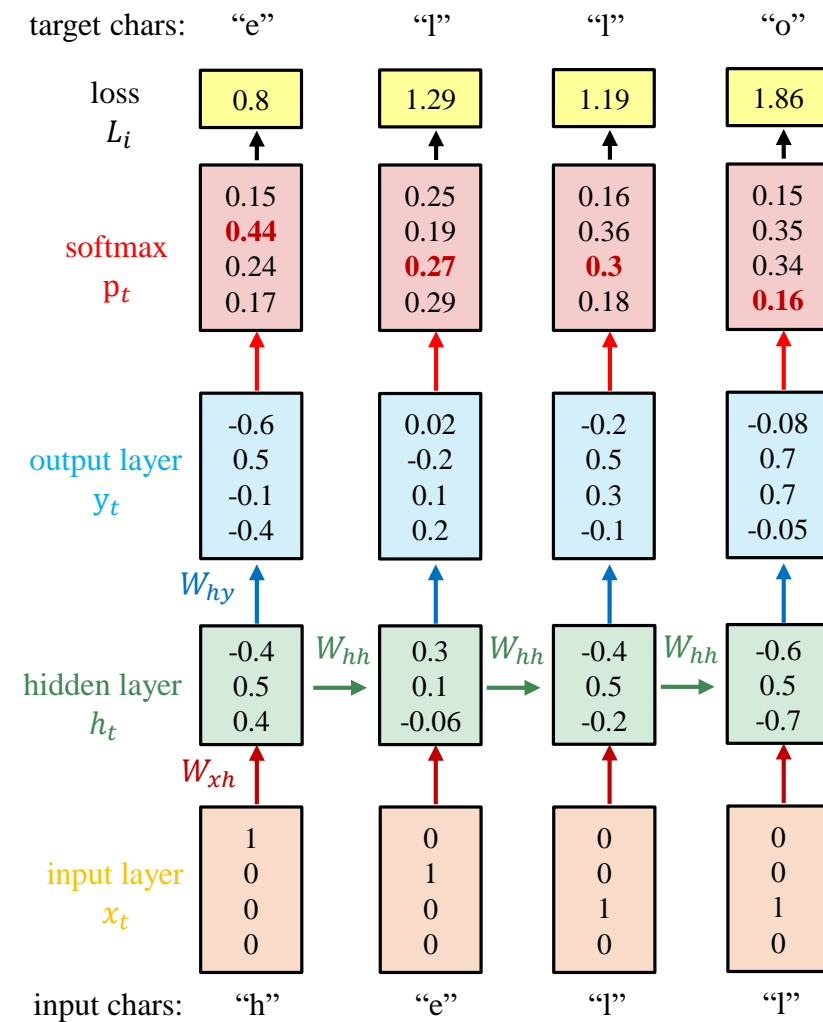
$$\frac{\partial L}{\partial b_y} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ -0.84 \end{bmatrix} + \begin{bmatrix} 0.16 \\ 0.36 \\ -0.7 \\ 0.18 \end{bmatrix} + \begin{bmatrix} 0.15 \\ 0.19 \\ -0.73 \\ 0.29 \end{bmatrix} + \begin{bmatrix} 0.15 \\ -0.56 \\ 0.24 \\ 0.17 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.3 \\ -0.8 \\ -0.1 \end{bmatrix}$$



BACKWARD PASS

$$\frac{\partial L}{\partial b_h} = \sum_t \frac{\partial L}{\partial h_t} \underbrace{\frac{\partial h_t}{\partial b_h}}_{(1 - h_t^2) * 1}$$

$$\frac{\partial L}{\partial b_h} = \begin{bmatrix} 0.06 \\ -0.5 \\ -0.1 \end{bmatrix}$$



BACKWARD PASS

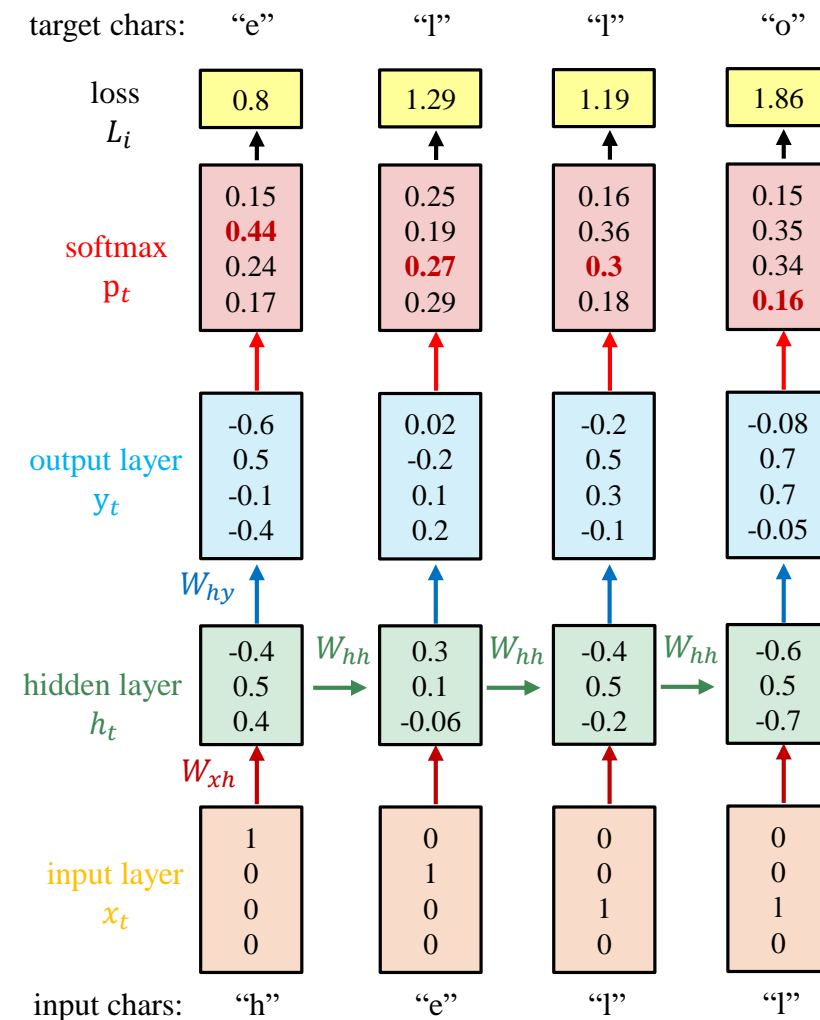
$$\frac{\partial L}{\partial W_{hh}} = \sum_t \frac{\partial L_t}{\partial W_{hh}}$$

$$\frac{\partial L_t}{\partial W_{hh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \underbrace{\frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}}_{(1 - h_t^2) * h_{t-1}}$$

at $k = 1$ $\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_2}{\partial h_1}$

at $k = 2$ $\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_3}{\partial h_2}$

...



$$\frac{\partial L}{\partial W_{hh}} = \begin{bmatrix} 0.09 & -0.1 & 0.1 \\ 0.1 & -0.1 & -0.1 \\ -0.05 & 0.1 & 0.07 \end{bmatrix}$$

BACKWARD PASS

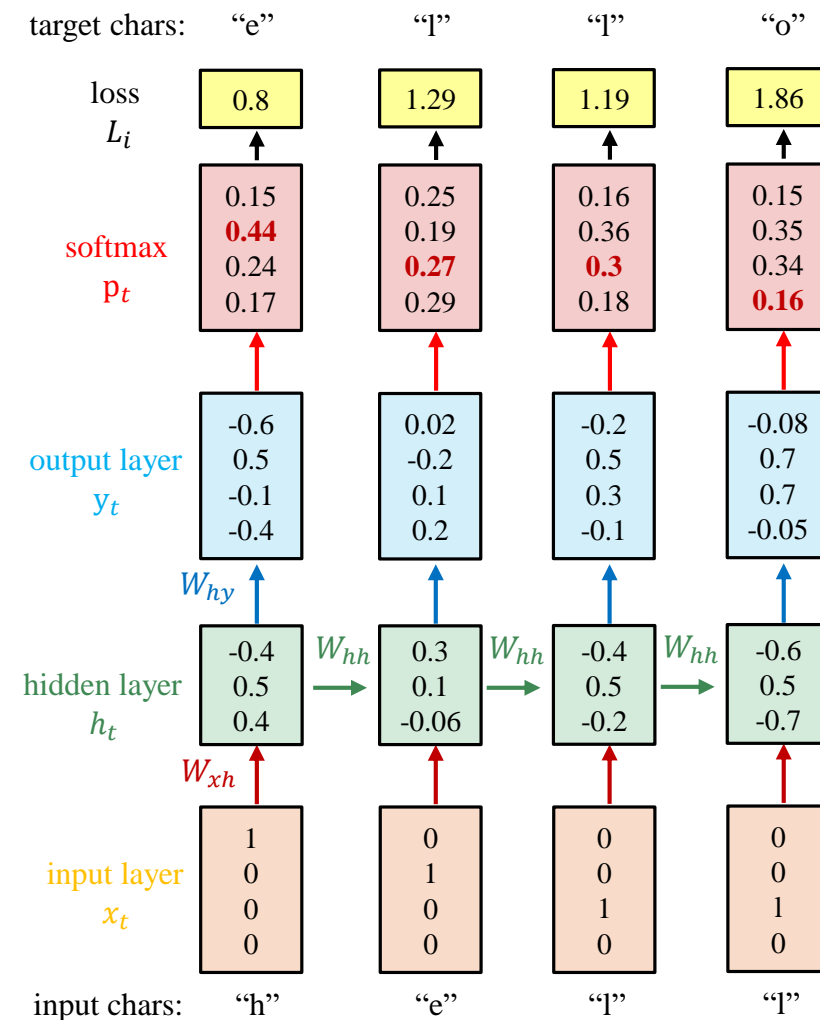
$$\frac{\partial L}{\partial W_{xh}} = \sum_t \frac{\partial L_t}{\partial W_{xh}}$$

$$\frac{\partial L_t}{\partial W_{xh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \underbrace{\frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{xh}}}_{(1 - h_t^2) * x_t}$$

at $k = 1$ $\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_2}{\partial h_1}$

at $k = 2$ $\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_3}{\partial h_2}$

...



$$\frac{\partial L}{\partial W_{xh}} = \begin{bmatrix} 0.5 & 0.04 & -0.5 & 0 \\ -0.2 & -0.3 & 0.1 & 0 \\ -0.4 & 0.1 & 0.1 & 0 \end{bmatrix}$$

```

10 # Set the dimension of the Vanilla NN model. We store by default Empathy (Empathy)
11 # and affect
12 #
13 # Input: happy, sad, etc
14
15 # Data size
16 data = open('train.txt', 'r').read() # words in training sets file
17 words = list(set(data))
18
19 dim_data, vocab_size = len(data), len(words)
20 print('data size: %d, vocab: %d' % (dim_data, vocab_size))
21
22 num_hid_1 = 10 # 1 for 1, 10 for 10, 100 for 100
23 num_hid_2 = 1 # size for 1, 10 for 10, 100 for 100
24
25 # Hyperparameters
26
27 hidden_size = 50 # size of hidden layer of neural
28 num_epochs = 10 # number of times to go over the data for
29 learning_rate = 0.1
30
31 # Model parameters
32 w = np.random.randn(hidden_size, vocab_size) # weights to hidden
33 w0 = np.random.randn(hidden_size, vocab_size) # bias to hidden
34 w1 = np.random.randn(vocab_size, hidden_size) # weights to output
35 w01 = np.random.randn(vocab_size, 1) # bias to output
36
37 def loss_and_grads, targets, words:
38     """
39     Returns the loss, gradients of model parameters, and last hidden state
40     """
41     w, w0, w1, w01 = w, w0, w1, w01
42     hid = np.zeros(hidden)
43     data = 0
44     # Forward pass
45     for i in range(len(words)):
46         w[i] = w.random(vocab_size, 1) # make it 1-d for representation
47         hid = hid + w[i]
48         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # hidden state
49         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
50         hid = hid + w[i]
51         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
52         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
53         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
54         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
55         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
56         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
57         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
58         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
59         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
60         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
61         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
62         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
63         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
64         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
65         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
66         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
67         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
68         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
69         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
70         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
71         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
72         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
73         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
74         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
75         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
76         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
77         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
78         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
79         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
80         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
81         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
82         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
83         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
84         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
85         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
86         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
87         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
88         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
89         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
90         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
91         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
92         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
93         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
94         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
95         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
96         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
97         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
98         w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output
99         w[i] = w.random(vocab_size, 1) # w = np.dot(w, w[i] - w) # output
100        w[i] = w.random(hid, 1) # w = np.dot(w, w[i] - w) # output

```

update parameters (W_{hy} , W_{hh} , W_{xh} , b_y , b_h)
with Adagrad

Main loop

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n %s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dWxh, dWhh, dWhy, dbh, 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

```

UPDATE PARAMETERS WITH ADAGRAD

Adagrad is a more efficient technique where the learning rate (α) are getting smaller during the training.

SGD

$$\theta_{t+1} = \theta_t - \alpha * \frac{\partial L}{\partial \theta_t}$$

Adagrad

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \epsilon=1e-8$$

UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

mW_{xh} , mW_{hh} , mW_{hy} , mb_h and mb_y initial to zero matrix in first iteration.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \begin{array}{l} \epsilon=1e-8 \\ \alpha=0.1 \end{array}$$

$$mW_{xh} += \frac{(W_{xh})^2}{\text{sum of squares}} \begin{bmatrix} 0.2 & 0.002 & 0.2 & 0 \\ 0.07 & 0.1 & 0.01 & 0 \\ 0.1 & 0.03 & 0.01 & 0 \end{bmatrix}$$

$$W_{xh} = \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix} - \frac{0.1}{\sqrt{0.000000008 + mW_{xh}}} \begin{bmatrix} 0.5 & 0.04 & -0.5 & 0 \\ -0.2 & -0.3 & 0.1 & 0 \\ -0.4 & 0.1 & 0.1 & 0 \end{bmatrix} = \begin{bmatrix} -0.6 & 0.3 & -0.1 & 0.3 \\ 0.6 & -0.2 & 0.5 & -0.8 \\ 0.5 & 0.3 & -0.4 & -0.8 \end{bmatrix}$$

UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

mW_{xh} , mW_{hh} , mW_{hy} , mb_h and mb_y initial to zero matrix in first iteration.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \begin{array}{l} \epsilon=1e-8 \\ \alpha=0.1 \end{array}$$

$$mW_{hy} += \overbrace{\begin{bmatrix} 0.01 & 0.3 & 0.2 \\ 0.4 & 0.09 & 0.06 \\ 0.01 & 0.1 & 0.5 \\ 0.2 & 0.002 & 0.1 \end{bmatrix}}^{(W_{hy})^2}$$

$$W_{hy} = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix} - \frac{0.1}{\sqrt{0.000000008 + mW_{hy}}} \begin{bmatrix} 0.1 & 0.2 & -0.1 \\ -0.06 & 0.1 & -0.5 \\ -0.2 & -0.1 & 0.05 \\ 0.5 & -0.2 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.7 & -0.4 \\ -0.6 & 0.2 & 0.01 \\ 0.2 & 0.5 & -0.8 \\ 0.4 & 0.1 & -0.5 \end{bmatrix}$$

UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

mW_{xh} , mW_{hh} , mW_{hy} , mb_h and mb_y initial to zero matrix in first iteration.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \begin{array}{l} \epsilon=1e-8 \\ \alpha=0.1 \end{array}$$

$$mW_{hh} += \overbrace{\begin{bmatrix} 0.08 & 0.03 & 0.01 \\ 0.01 & 0.01 & 0.02 \\ 0.002 & 0.01 & 0.005 \end{bmatrix}}^{(W_{hh})^2}$$

$$W_{hh} = \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} - \frac{0.1}{\sqrt{0.000000008 + mW_{hh}}} \begin{bmatrix} 0.09 & -0.1 & 0.1 \\ 0.1 & -0.1 & -0.1 \\ -0.05 & 0.1 & 0.07 \end{bmatrix} = \begin{bmatrix} -0.2 & -0.7 & 0.6 \\ -0.1 & 0.3 & 0.8 \\ 0.6 & -0.6 & -0.08 \end{bmatrix}$$

UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

mW_{xh} , mW_{hh} , mW_{hy} , mb_h and mb_y initial to zero matrix in first iteration.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \begin{array}{l} \epsilon=1\text{e-}8 \\ \alpha=0.1 \end{array}$$

$$mb_h \overset{(b_h)^2}{+=} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$b_h = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \frac{0.1}{\sqrt{0.000000008 + mb_h}} \begin{bmatrix} 0.06 \\ -0.5 \\ -0.1 \end{bmatrix} = \begin{bmatrix} -0.09 \\ 0.09 \\ 0.09 \end{bmatrix}$$

UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

mW_{xh} , mW_{hh} , mW_{hy} , mb_h and mb_y initial to zero matrix in first iteration.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum_j \left(\frac{\partial L}{\partial \theta_j} \right)^2}} * \frac{\partial L}{\partial \theta_t} \quad \begin{array}{l} \epsilon=1e-8 \\ \alpha=0.1 \end{array}$$

$$mb_y += \overbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}^{(b_y)^2}$$

$$b_y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} - \frac{0.1}{\sqrt{0.000000008 + mb_y}} \begin{bmatrix} 0.6 \\ 0.3 \\ -0.8 \\ -0.1 \end{bmatrix} = \begin{bmatrix} -0.09 \\ -0.09 \\ 0.09 \\ 0.09 \end{bmatrix}$$

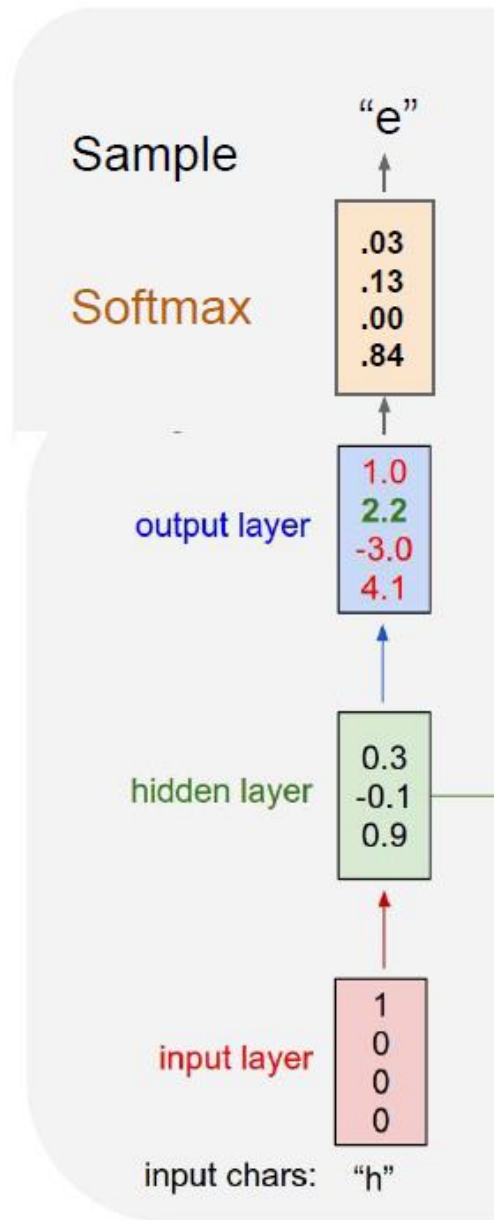
[illegible]

```
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
    """
    x = np.zeros((vocab_size, 1))
    x[seed_ix] = 1
    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
```

Character-level language model example

Vocabulary:
[h,e,l,o]

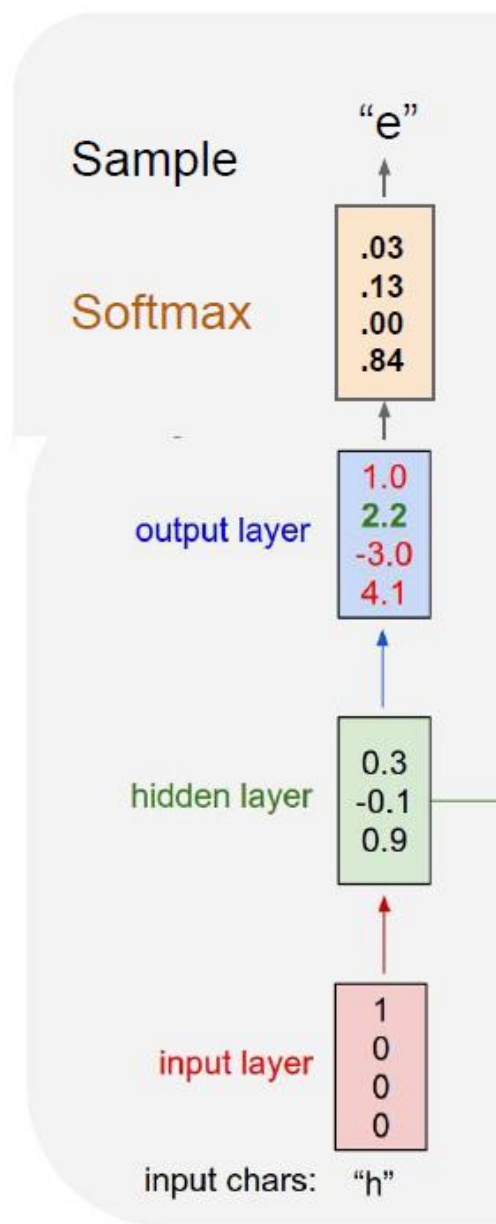
At **test-time** sample characters one at a time, feed back to model



Character-level language model example

Vocabulary:
[h,e,l,o]

At **test-time** sample characters one at a time, feed back to model



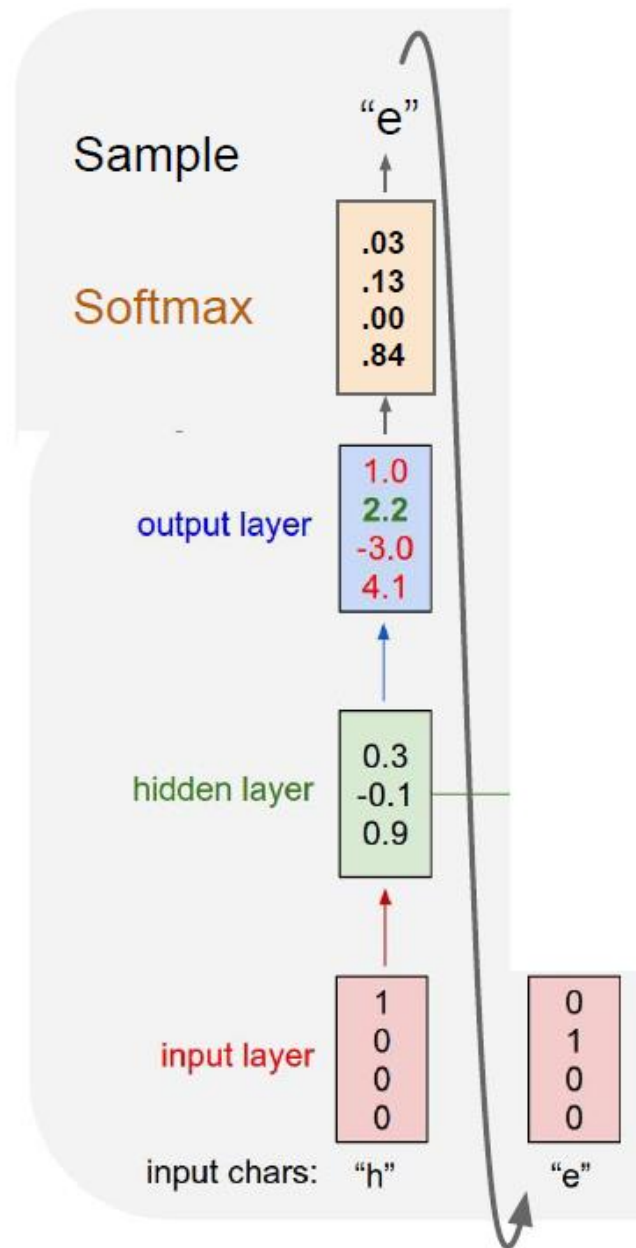
Why might we sample?

It lets you get **diversity** from the model.

Character-level language model example

Vocabulary:
[h,e,l,o]

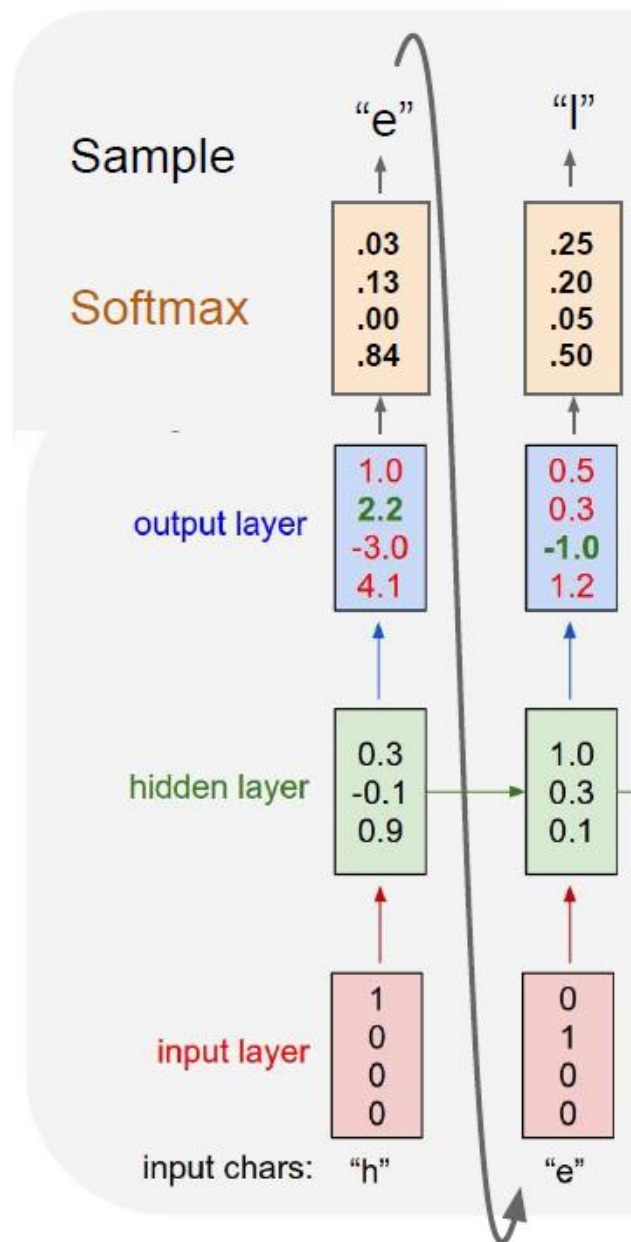
At **test-time** sample characters one at a time, feed back to model



Character-level language model example

Vocabulary:
[h,e,l,o]

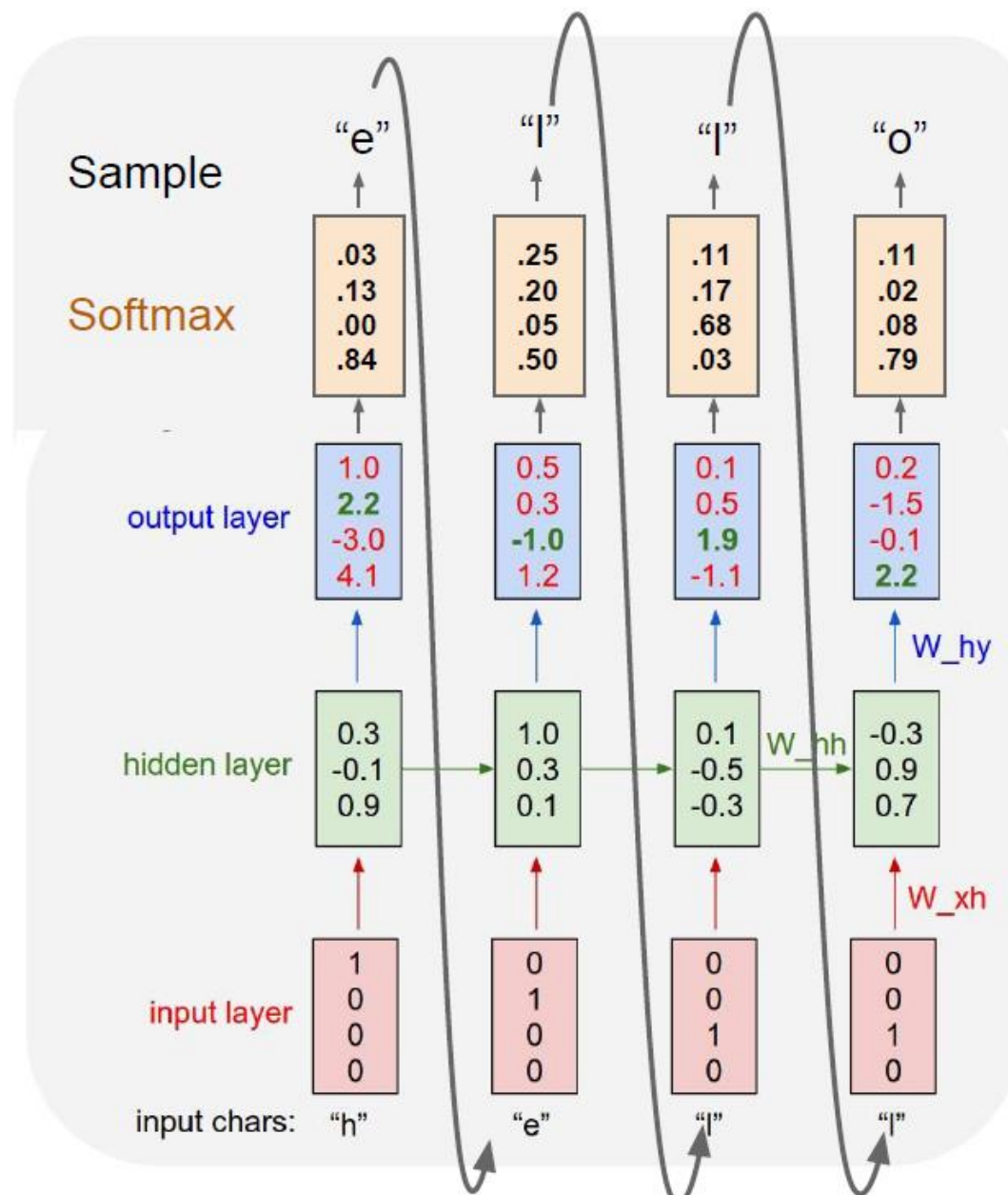
At **test-time** sample characters one at a time, feed back to model



Character-level language model example

Vocabulary:
[h,e,l,o]

At **test-time** sample characters one at a time, feed back to model



Thanks