# RNN With Example

Shima Foolad

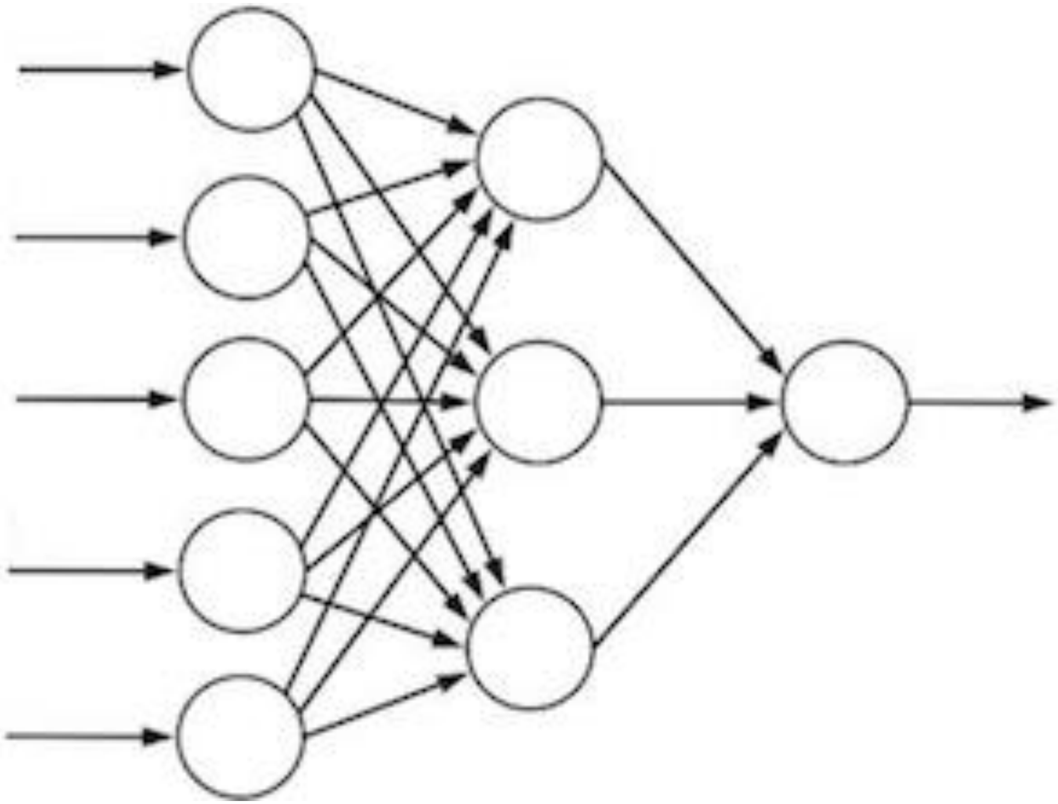
29 Oct. 2017

# 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.



HTTPS://AYEAROFAI.COM/ROHAN-LENNY-3-RECURRENT-NEURAL-NETWORKS-10300100899B

# 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.

HTTPS://AYEAROFAI.COM/ROHAN-LENNY-3-RECURRENT-NEURAL-NETWORKS-10300100899B

# 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.

HTTPS://AYEAROFAI.COM/ROHAN-LENNY-3-RECURRENT-NEURAL-NETWORKS-10300100899B

# EXAMPLE FOR RNN

**Character-level language model example**

2017, STANFORD UNIVERSITY, C321N, LECTURE 10
FEI-FEI LI & ANDREJ KARPATHY & JUSTIN JOHNSON

# [min-char-rnn.py gist: 112 lines of Python](https://gist.github.com/karpathy/d4dee566867f8291f086)

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

```python
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

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

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

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

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, 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
```

([https://gist.github.com/karpathy/d4dee566867f8291f086](https://gist.github.com/karpathy/d4dee566867f8291f086))

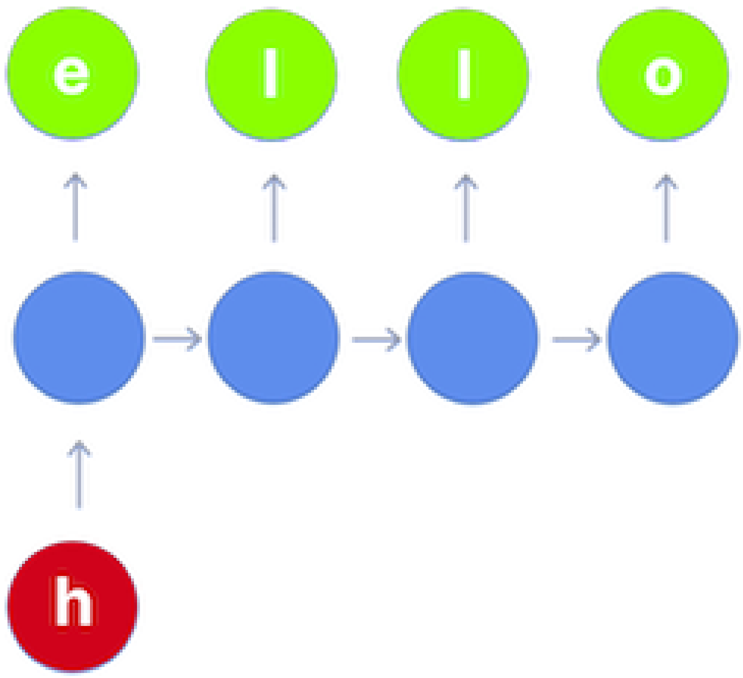
# AN EXAMPLE

**Character-level language model example**

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

Vocabulary:
[h,e,l,o]

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

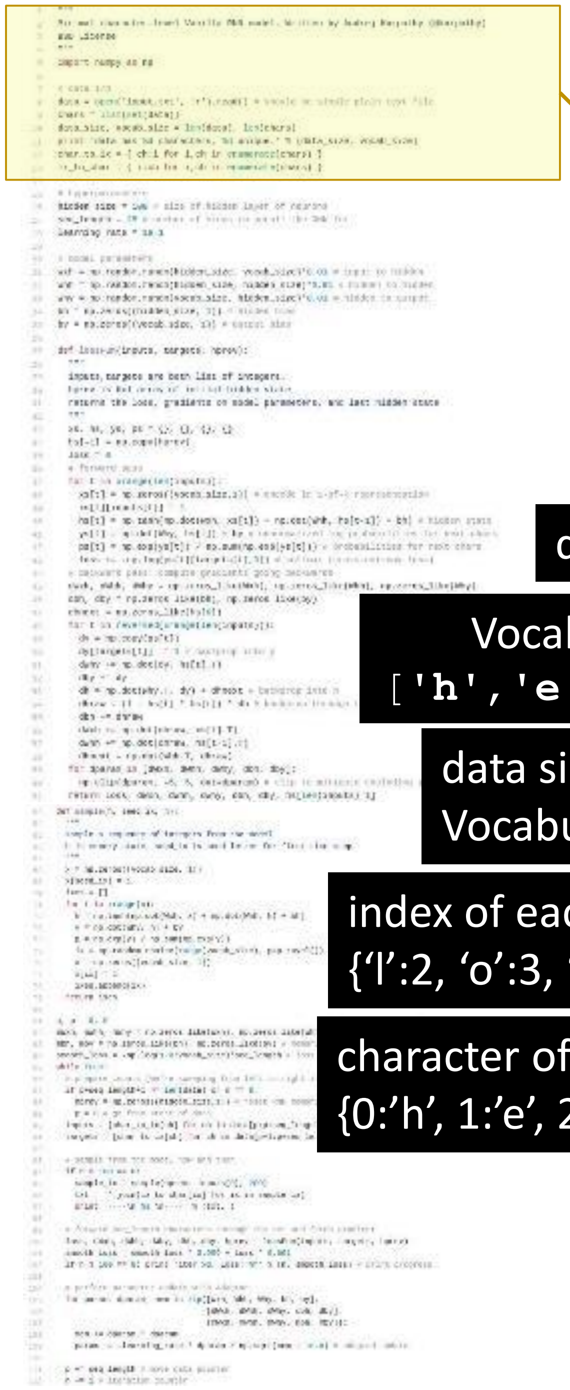



Example training sequence:
**"hello"**

HTTPS://AYEAROFAI.COM/ROHAN-LENNY-3-RECURRENT-NEURAL-NETWORKS-10300100899B

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

# Data I/O



```python
"""
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) }
```

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’}

min-char-rnn.py gist

# Initializations

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

wxh is Weight matrix 3*4 between input layer and output layer

whh is Weight matrix 3*3 between two hidden layers

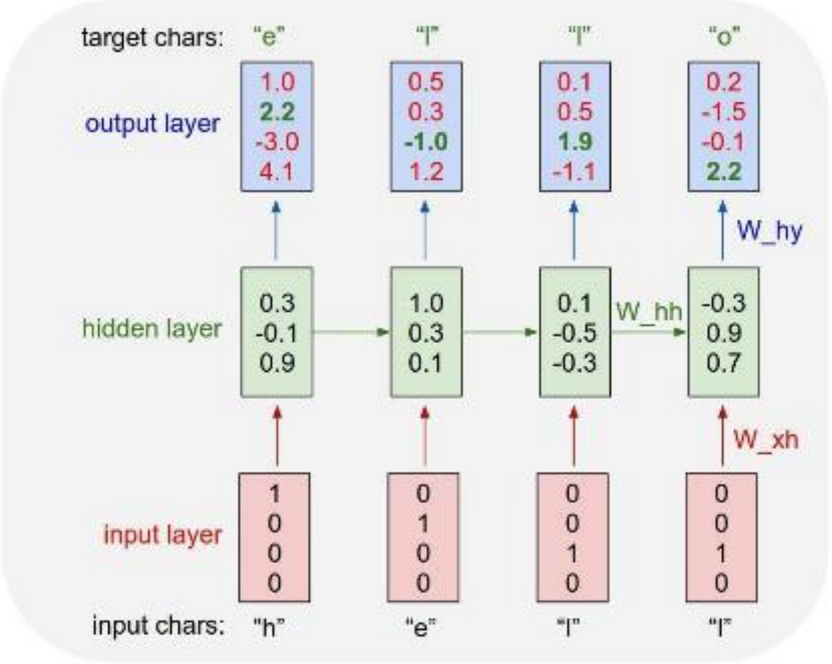
why is Weight matrix 4*3 between hidden layer and output layer

bh is hidden bias

by is output bias

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

recall:



min-char-rnn.py gist

# Initializations

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

wxh is Weight matrix 3*4 between input layer and output layer

whh is Weight matrix 3*3 between two hidden layers

why is Weight matrix 4*3 between hidden layer and output layer

bh is hidden bias

by is output bias

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

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

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

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

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

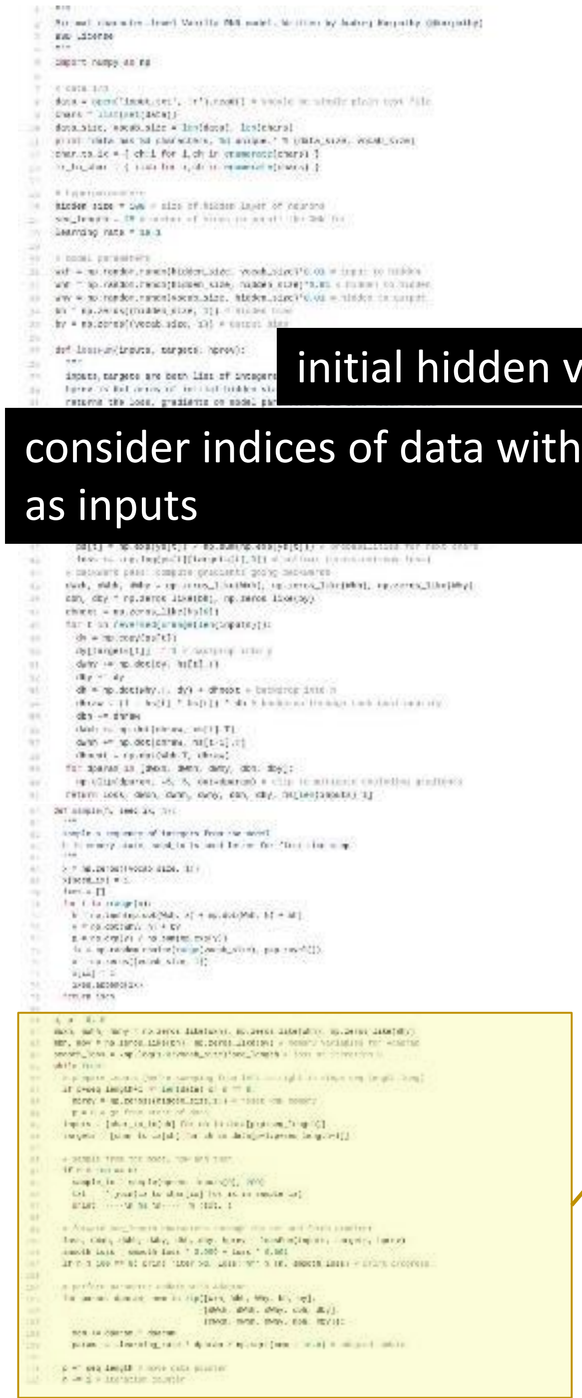
$$b_y = [0 \quad 0 \quad 0 \quad 0]$$

**min-char-rnn.py** gist

# Main loop

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

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

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

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, 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 data pointer
  n += 1 # iteration counter
```

[**min-char-rnn.py** gist]



# Main loop

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

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

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]

```python
t
, hprev)
rint progress
```

```python
  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
```

[min-char-rnn.py gist]

# Main loop

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

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

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]

```python
  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
```

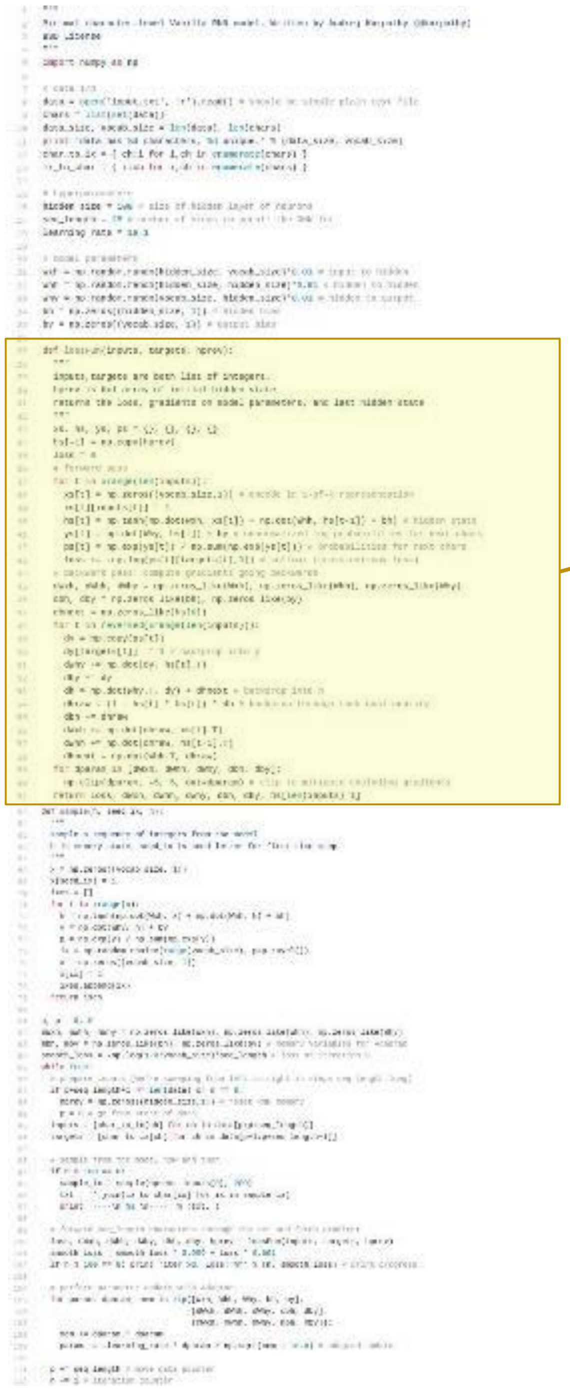
[min-char-rnn.py gist]

# Main loop

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

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

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

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, 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
```

Do a forward pass, backward pass, and get the gradients in a Loss function (lossFun)

[**min-char-rnn.py** gist]



# Loss function

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

Forward pass

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

# min-char-rnn.py gist

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

for each timestep

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

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

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

probability distribution (Softmax) $$p_t = \frac{e^{y_t}}{\sum_j e^{y_j}}$$

Softmax classifier

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

# FORWARD PASS

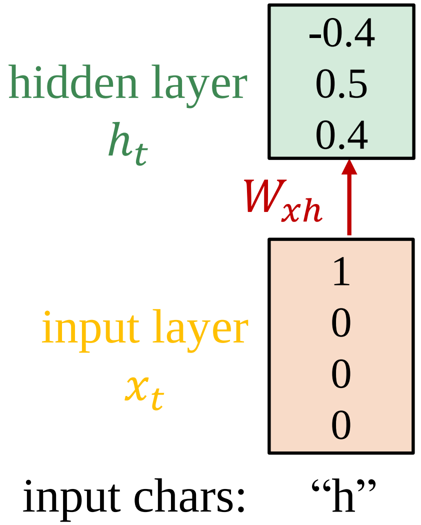
Example training sequence: **“hello”**

timestep 1:
(t=1)

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

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

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




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

$$= tanh\left(\begin{bmatrix} -0.5 \\ 0.5 \\ 0.4 \end{bmatrix}\right) = \begin{bmatrix} -0.4 \\ 0.5 \\ 0.4 \end{bmatrix}$$

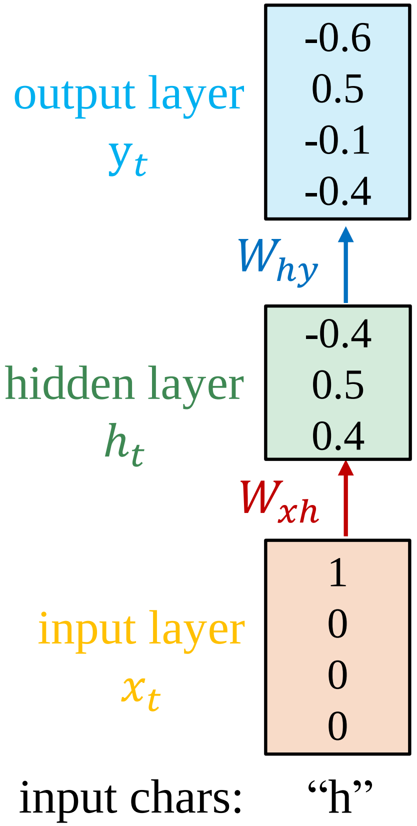
# FORWARD PASS

Example training sequence: **"hello"**

timestep 1:
(t=1)

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

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



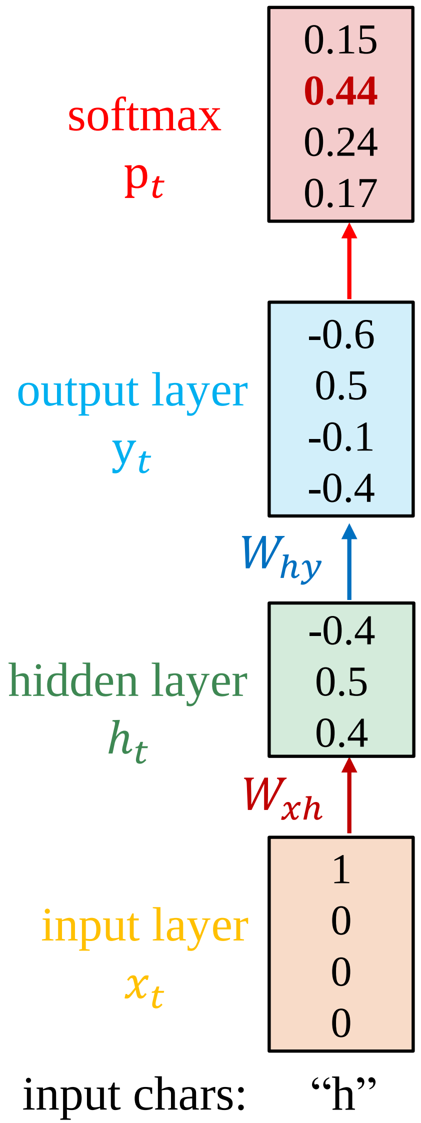

$$y_t = W_{hy} h_t + b_y$$

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

# FORWARD PASS

Example training sequence: **"hello"**

timestep 1:
(t=1)

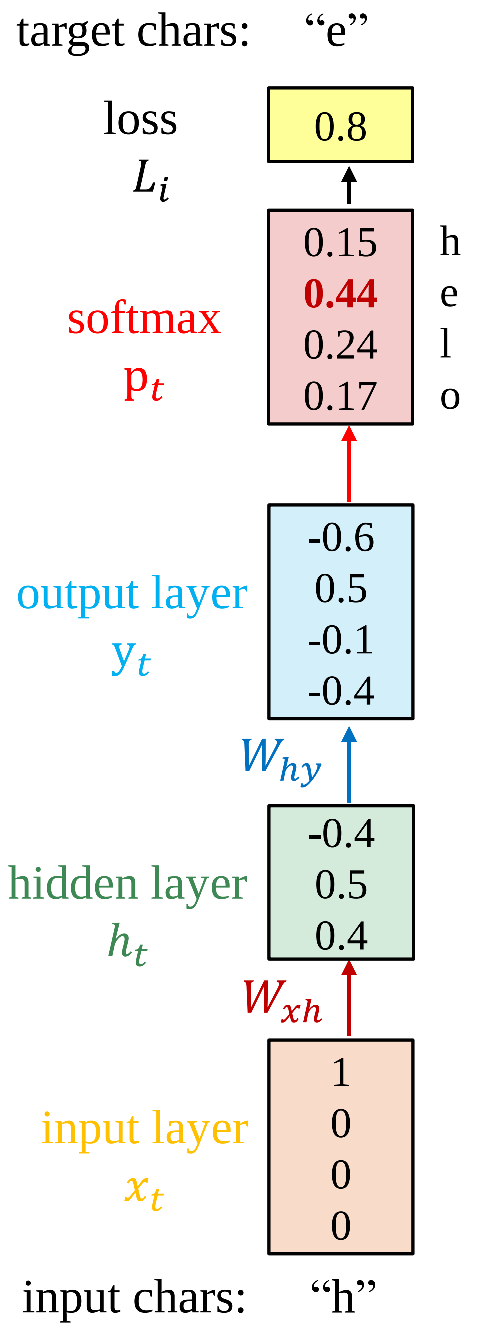



$$p_t = \frac{e^{y_t}}{\sum_j e^{y_j}}$$

# FORWARD PASS

Example training sequence: **“hello”**

timestep 1:
(t=1)



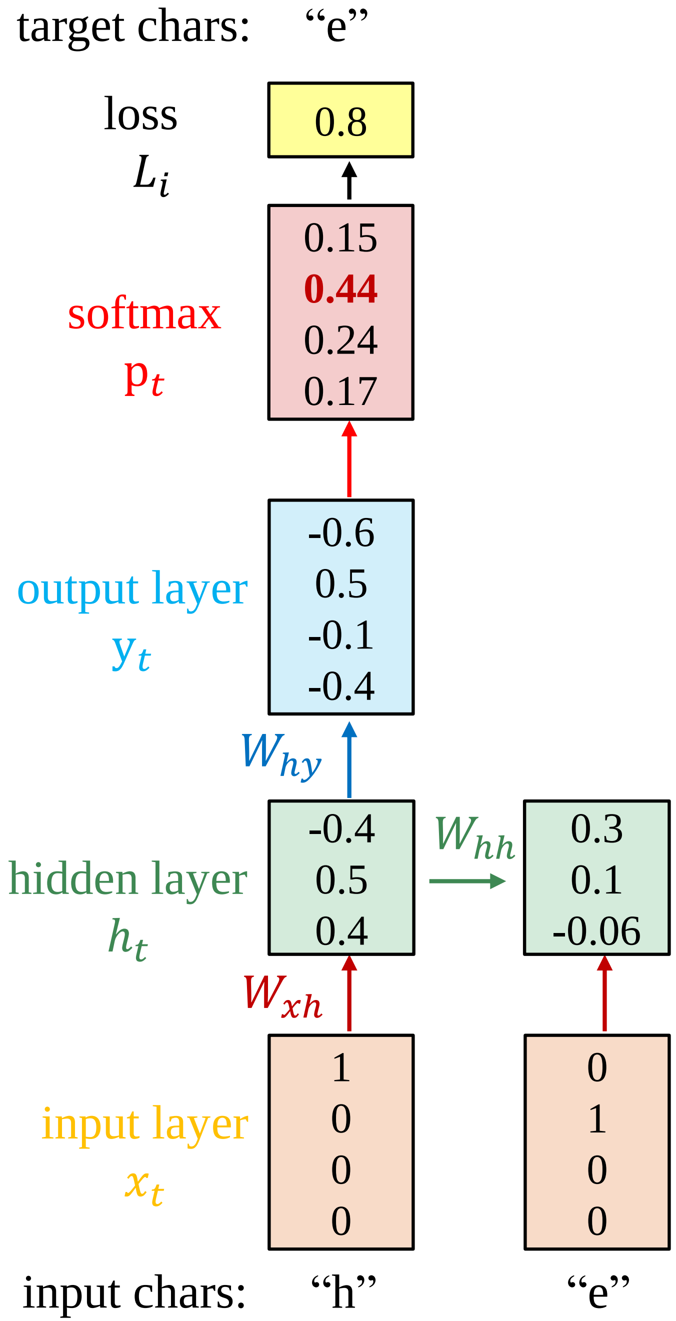

$$L_i = -log(p_{target_i})$$

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

probability distribution of target char (e)

# FORWARD PASS

timestep 2:
(t=2)




Example training sequence: **"hello"**

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

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

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

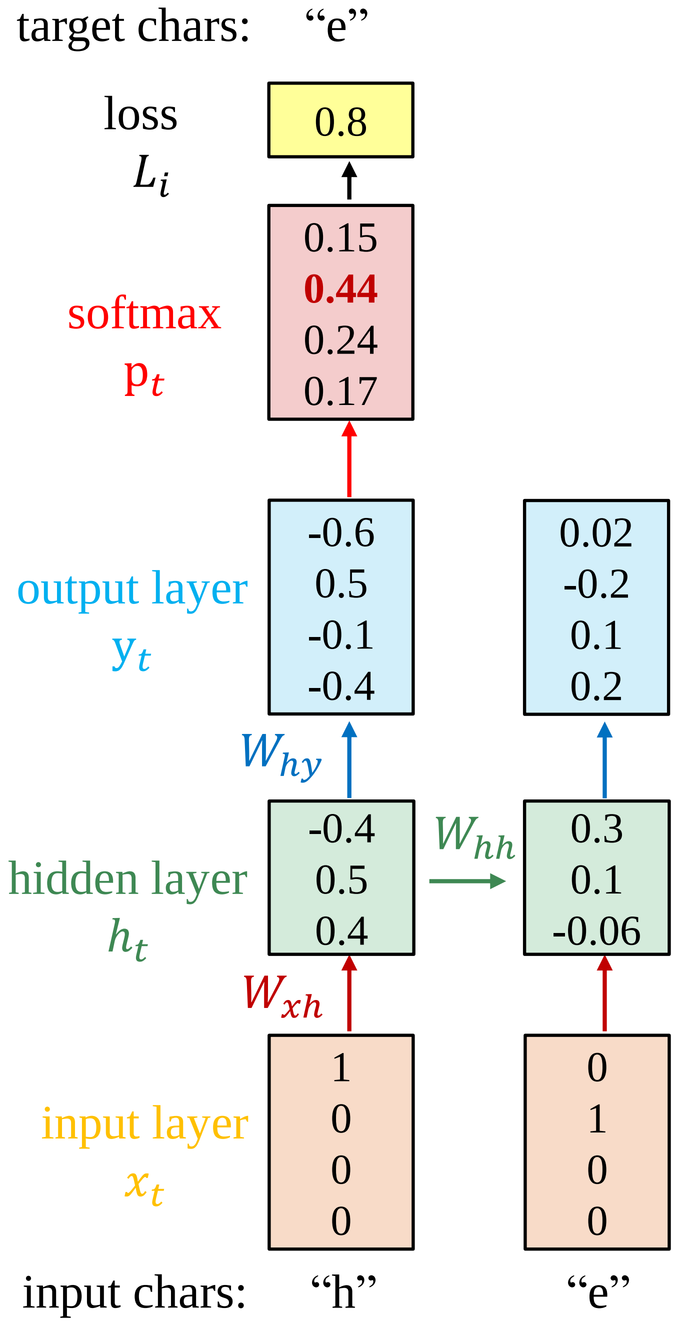
$$h_2 = tanh\left( \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} \begin{bmatrix} -0.4 \\ 0.5 \\ 0.4 \end{bmatrix} + \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0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

$$= \begin{bmatrix} 0.3 \\ 0.1 \\ -0.06 \end{bmatrix}$$

# FORWARD PASS

Example training sequence: **"hello"**

timestep 2:
(t=2)




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

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

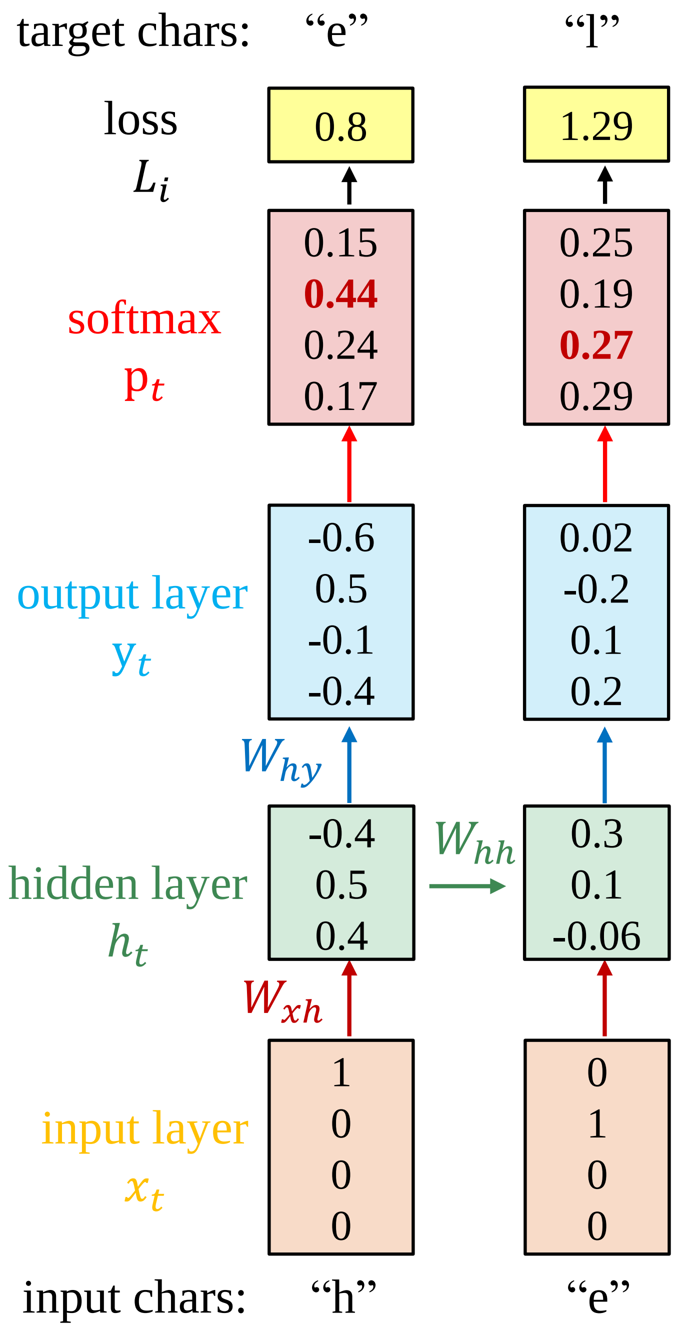
$$y_t = W_{hy} h_t + b_y$$

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

# FORWARD PASS

Example training sequence: **“hello”**

timestep 2:
(t=2)




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

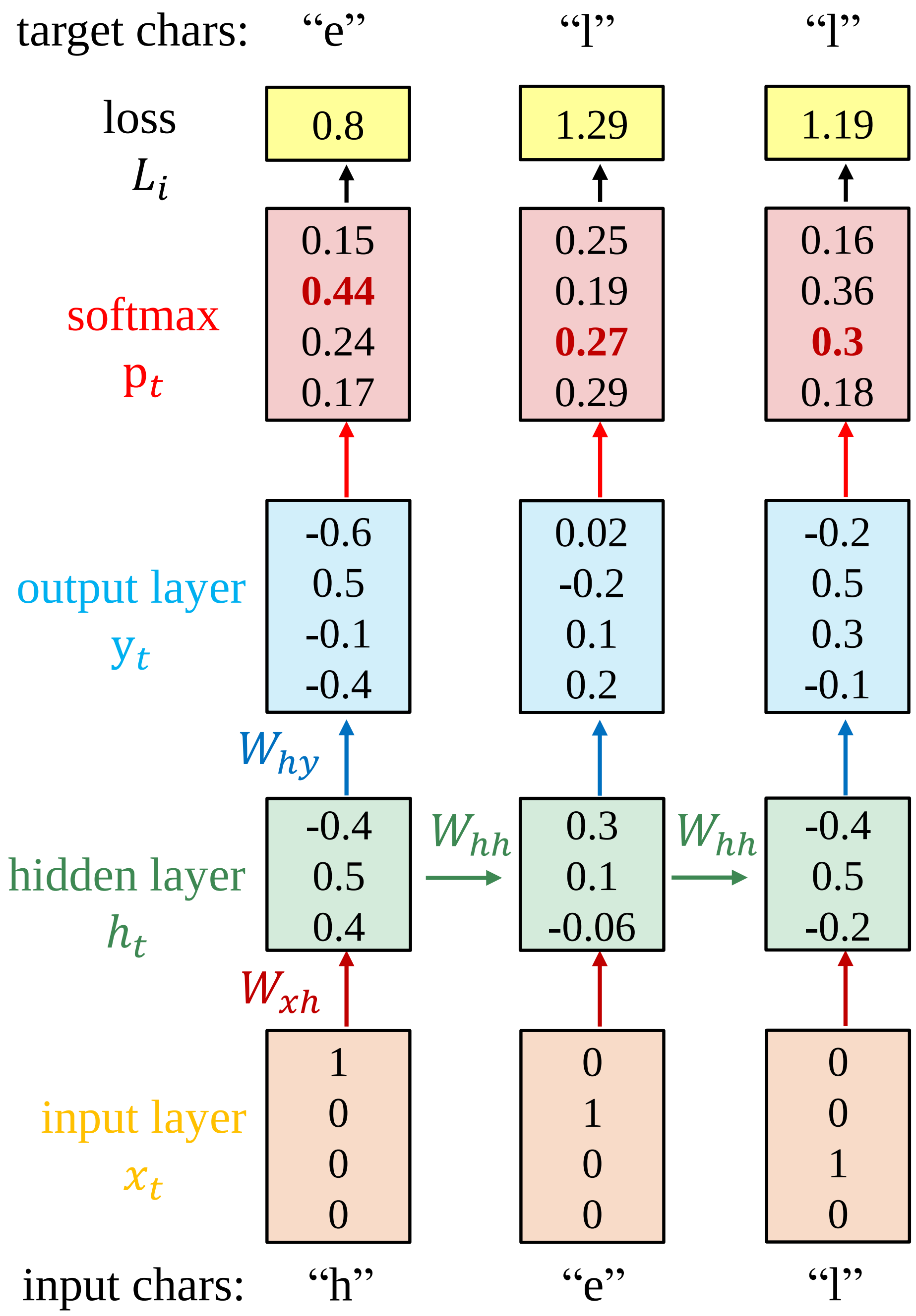
$$L_1 = -log(\underbrace{0.27}) = 1.29$$

probability distribution of target char (l)

# FORWARD PASS

Example training sequence: **"hello"**

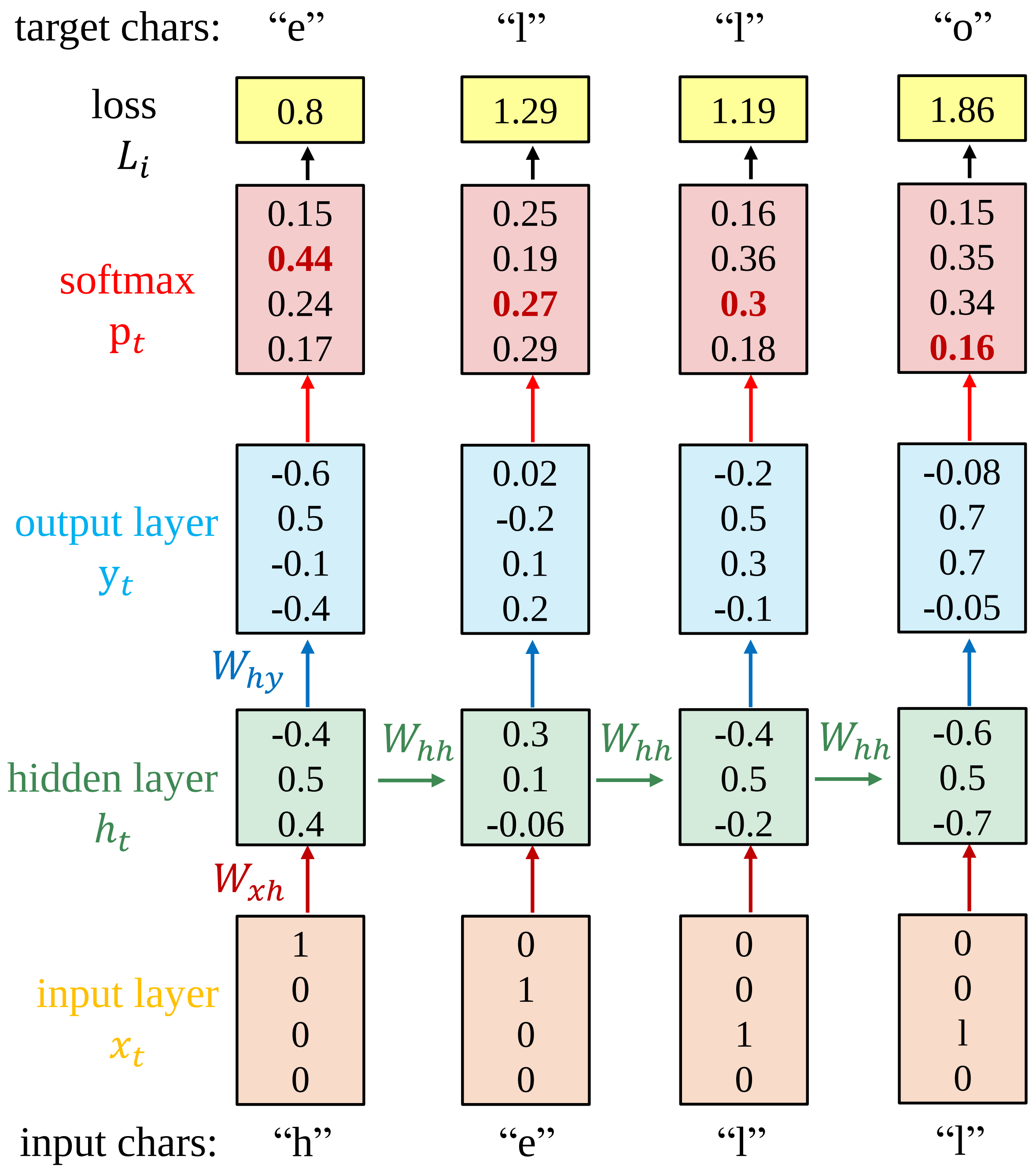
timestep 3:
(t=3)



# FORWARD PASS

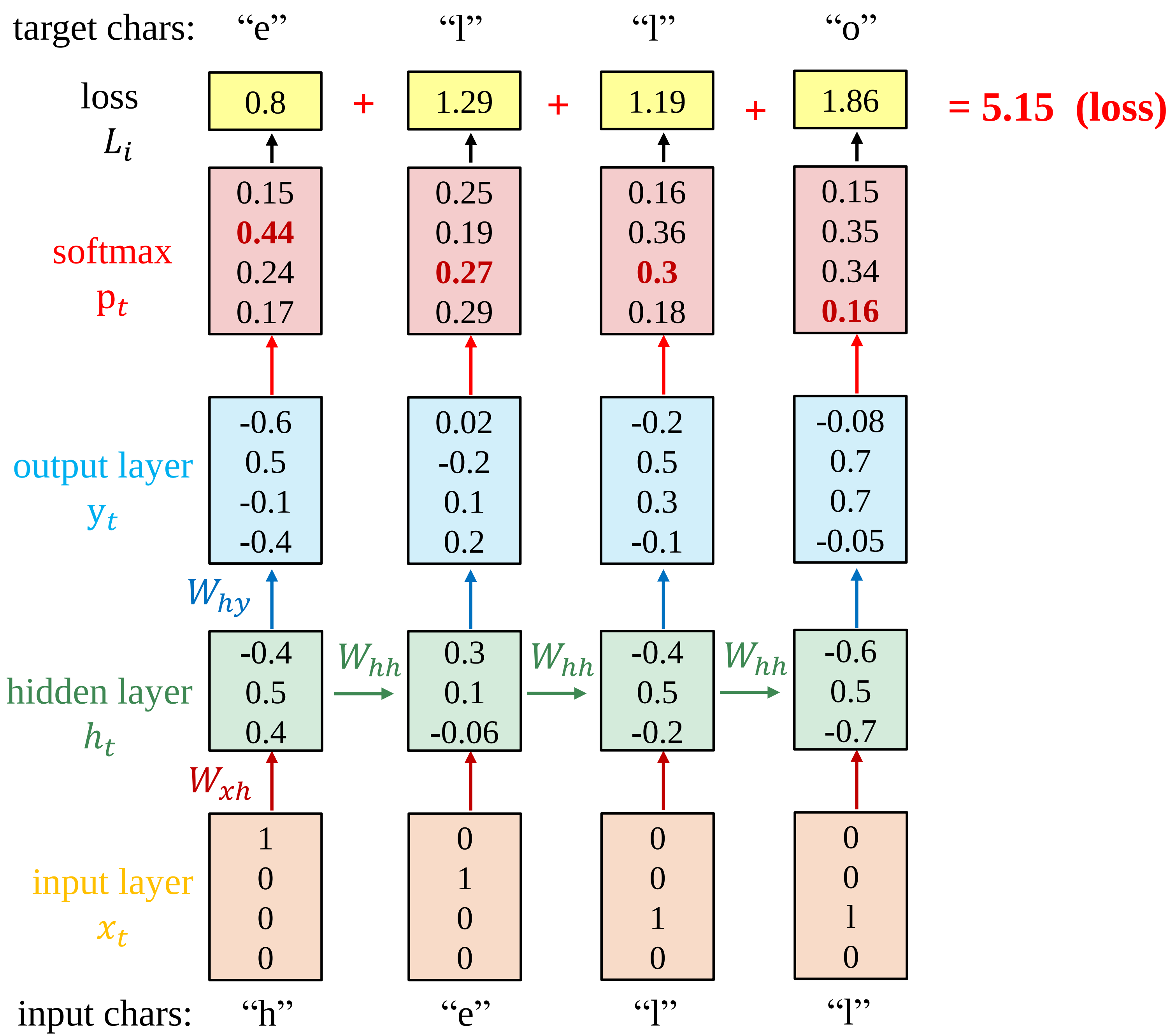
Example training sequence: **“hello”**

timestep 4:
(t=4)



# FORWARD PASS

Example training sequence: **“hello”**



**min-char-rnn.py** gist

# Loss function

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

Backward pass

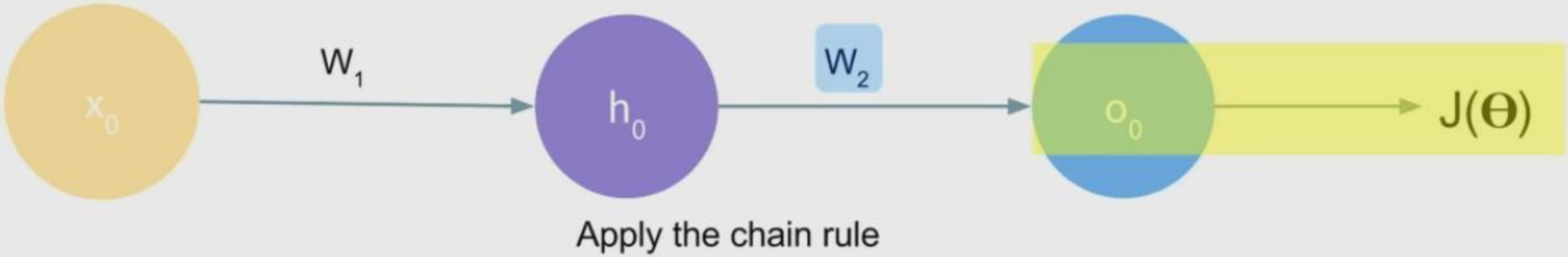
```python
def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward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

# Calculating the Gradient: Backpropagation


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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

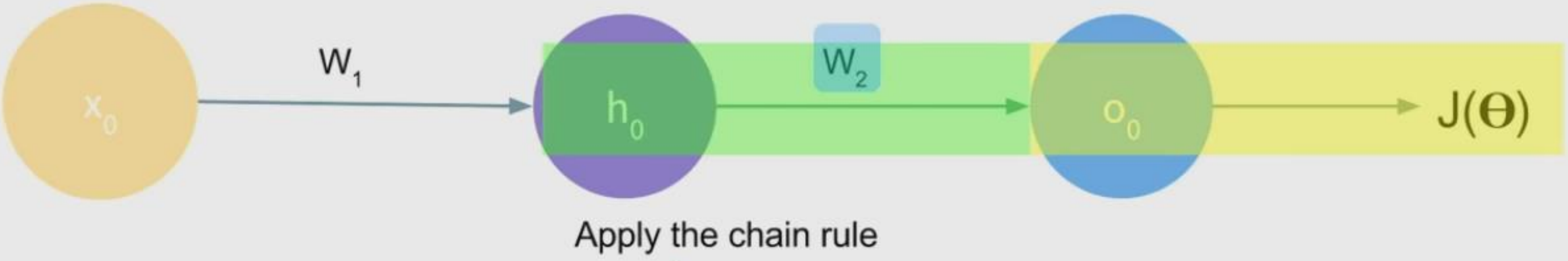
# Calculating the Gradient: Backpropagation




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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

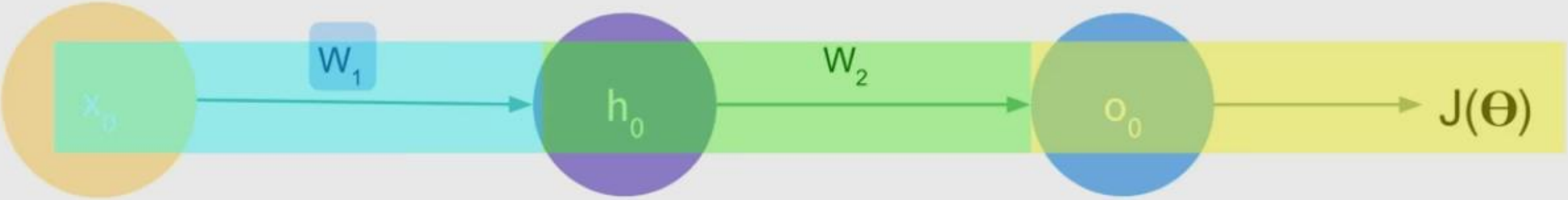
# Calculating the Gradient: Backpropagation




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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

# Calculating the Gradient: Backpropagation




Apply the chain rule

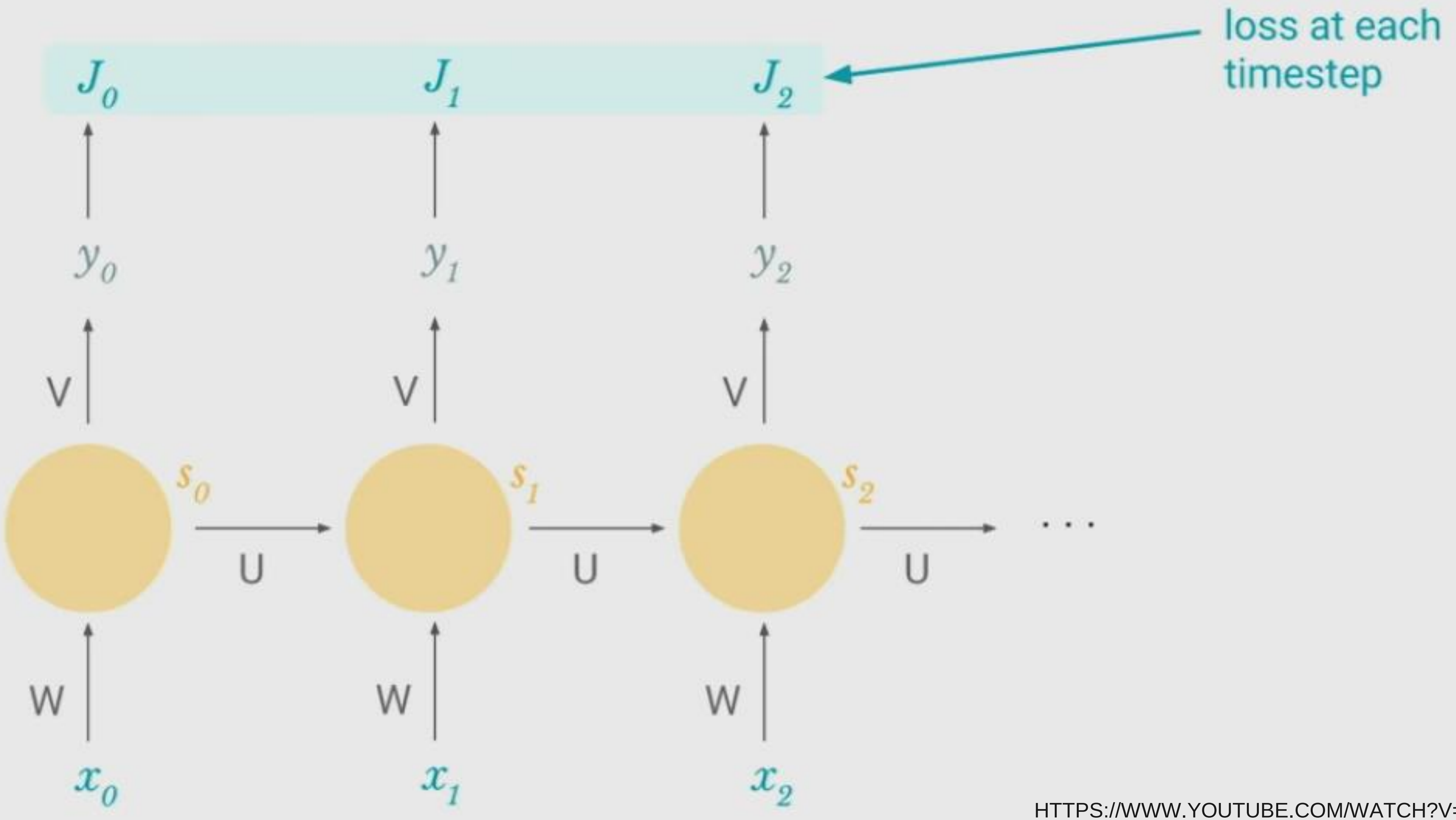
Apply the chain rule

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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

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

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




HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

we **sum the losses** across time:

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

$\mathbf{\Theta}$ = our parameters, like weights

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

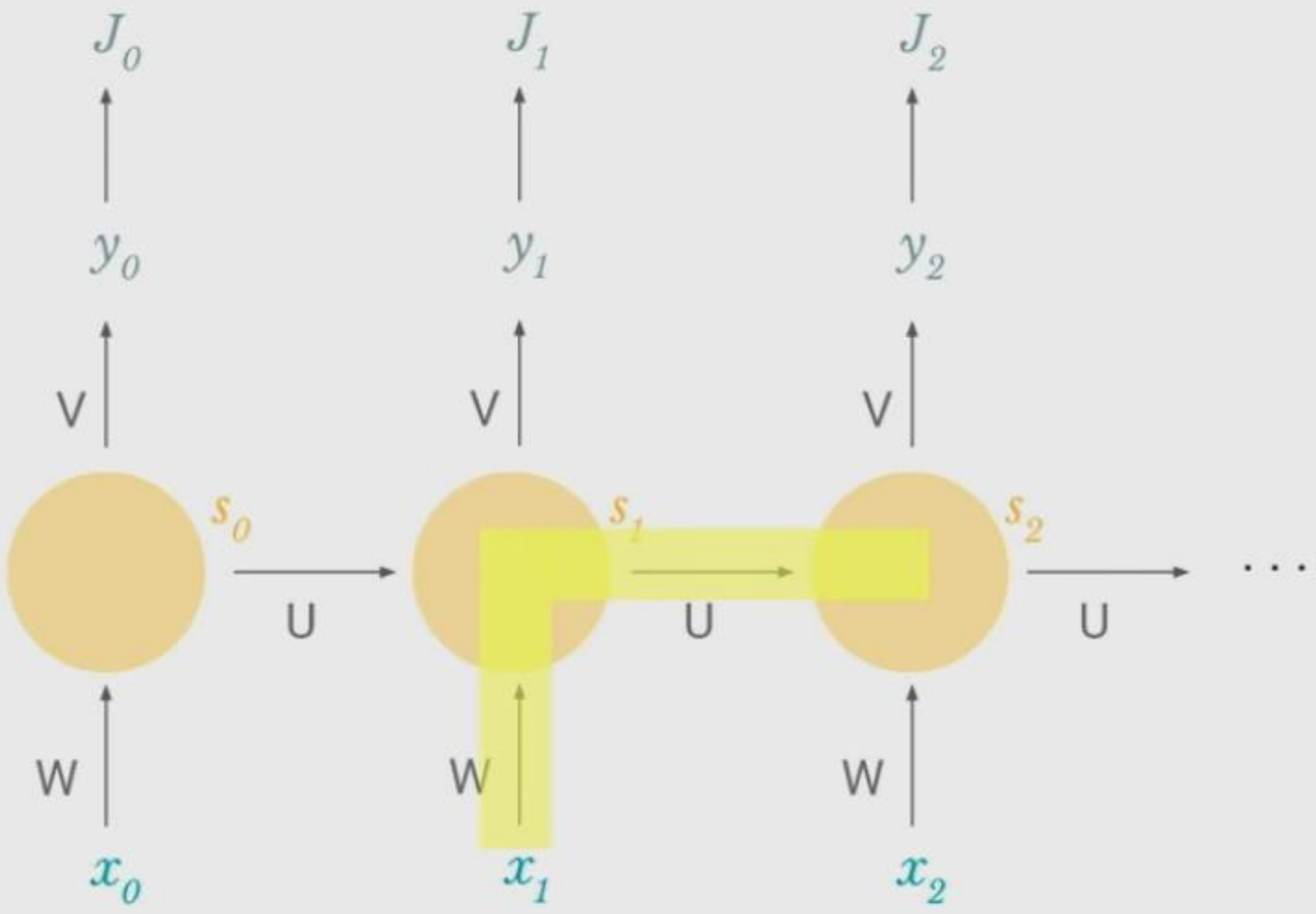
# what are our **gradients**?

we sum gradients across time for each parameter *P*:

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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

# 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...

$$s_2 = tanh(Us_1 + Wx_2)$$

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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

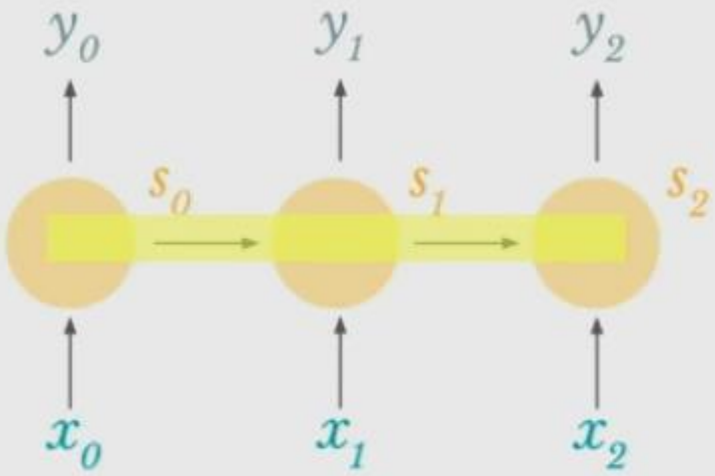
# backpropagation through time:

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

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

# **problem**: vanishing gradient

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

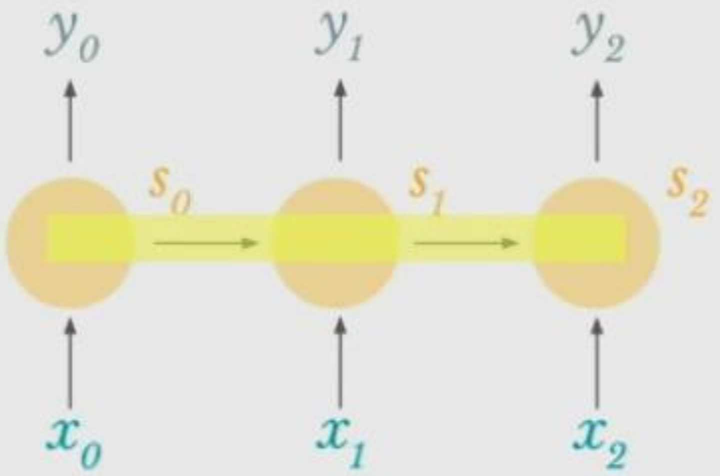



at $k = 0$: $$\boxed{\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}}$$

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

# **problem**: vanishing gradient

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




at $k = 0$: $\boxed{\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}}$

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

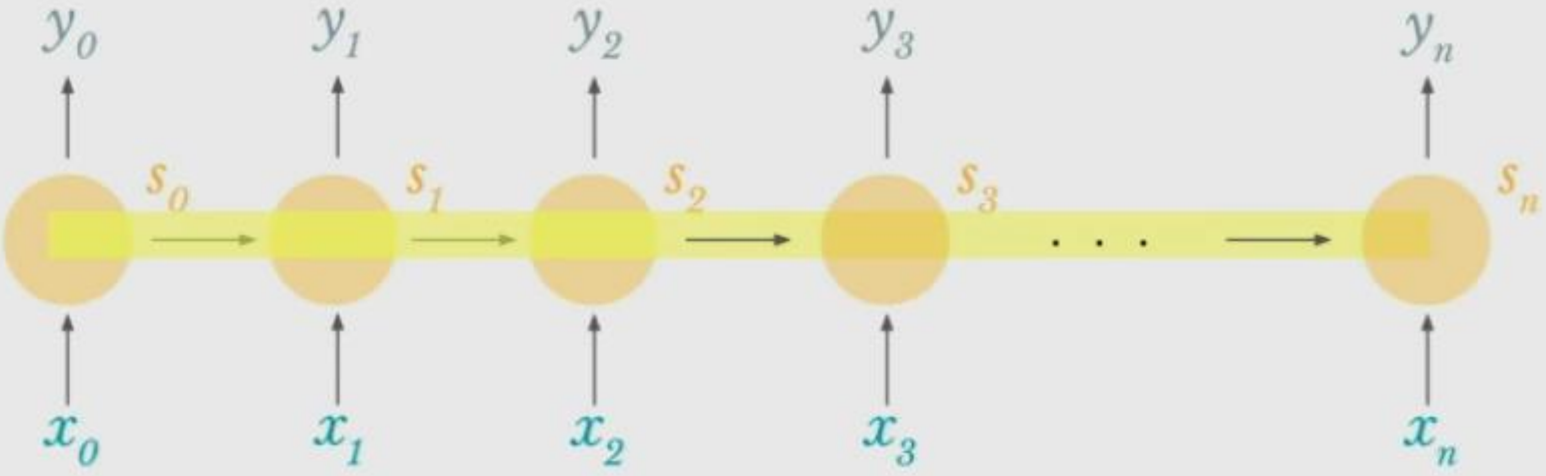
**problem**: vanishing gradient

at k=0

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

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

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




HTTPS://WWW.YOUTUBE.COM/WATCH?V=L4X-KZJL1GS

**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 $W_{hh}$

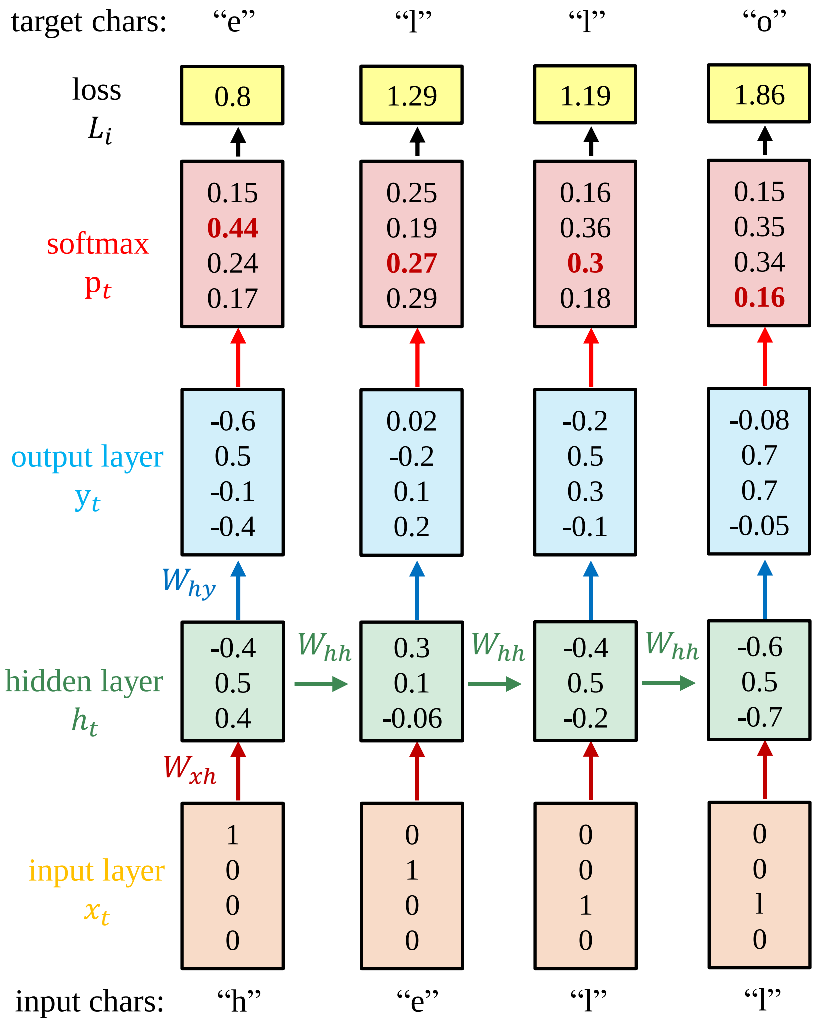
```python
# backward pass: compute gradients going backwards
dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
dbh, dby = np.zeros_like(bh), np.zeros_like(by)
dhnext = np.zeros_like(hs[0])
for t in reversed(xrange(len(inputs))):
  dy = np.copy(ps[t])
  dy[targets[t]] -= 1 # backprop into y
  dWhy += np.dot(dy, hs[t].T)
  dby += dy
  dh = np.dot(Why.T, dy) + dhnext # backprop into h
  dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
  dbh += dhraw
  dWxh += np.dot(dhraw, xs[t].T)
  dWhh += np.dot(dhraw, hs[t-1].T)
  dhnext = np.dot(Whh.T, dhraw)
for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
  np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

# BACKWARD PASS

$$\frac{\partial L}{\partial y} = \sum_t \frac{\partial L_t}{\partial y_t} \qquad \frac{\partial L_t}{\partial y_t} = output - target$$

$$\frac{\partial L_4}{\partial y_4} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ 0.16 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.15 \\ 0.35 \\ 0.34 \\ -0.84 \end{bmatrix} \qquad \frac{\partial L_3}{\partial y_3} = \begin{bmatrix} 0.16 \\ 0.36 \\ 0.3 \\ 0.18 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.16 \\ 0.36 \\ -0.7 \\ 0.18 \end{bmatrix}$$

$$\frac{\partial L_2}{\partial y_2} = \begin{bmatrix} 0.25 \\ 0.19 \\ 0.27 \\ 0.29 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.15 \\ 0.19 \\ -0.73 \\ 0.29 \end{bmatrix} \qquad \frac{\partial L_1}{\partial y_1} = \begin{bmatrix} 0.15 \\ 0.44 \\ 0.24 \\ 0.17 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.15 \\ -0.56 \\ 0.24 \\ 0.17 \end{bmatrix}$$



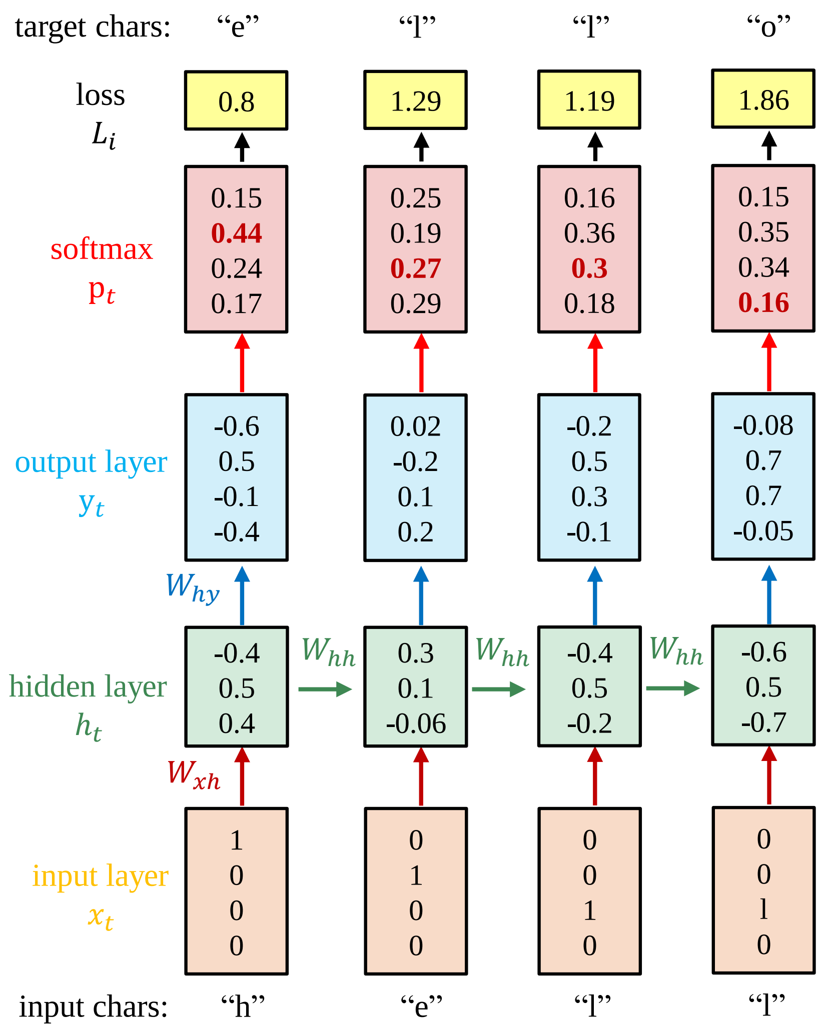
# BACKWARD PASS

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

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

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

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

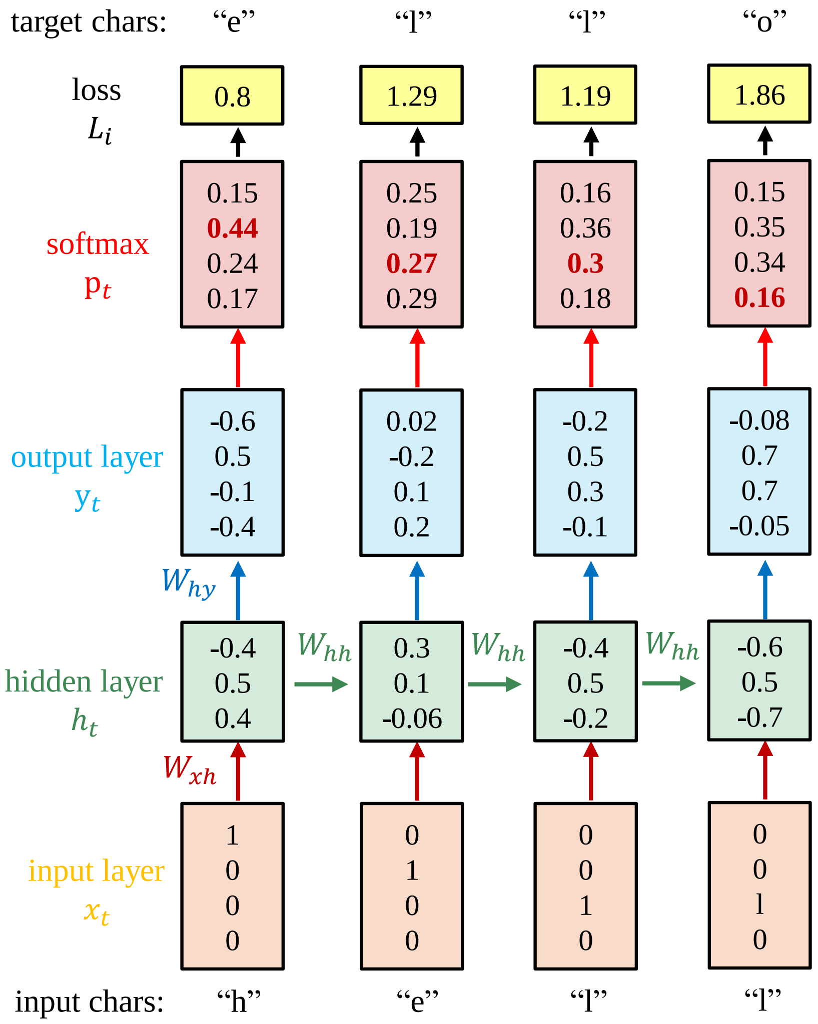


# BACKWARD PASS

$$\frac{\partial L}{\partial b_y} = \sum_t \frac{\partial L}{\partial y_t} \underbrace{\frac{\partial y_t}{\partial b_y}}_{1}$$

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

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

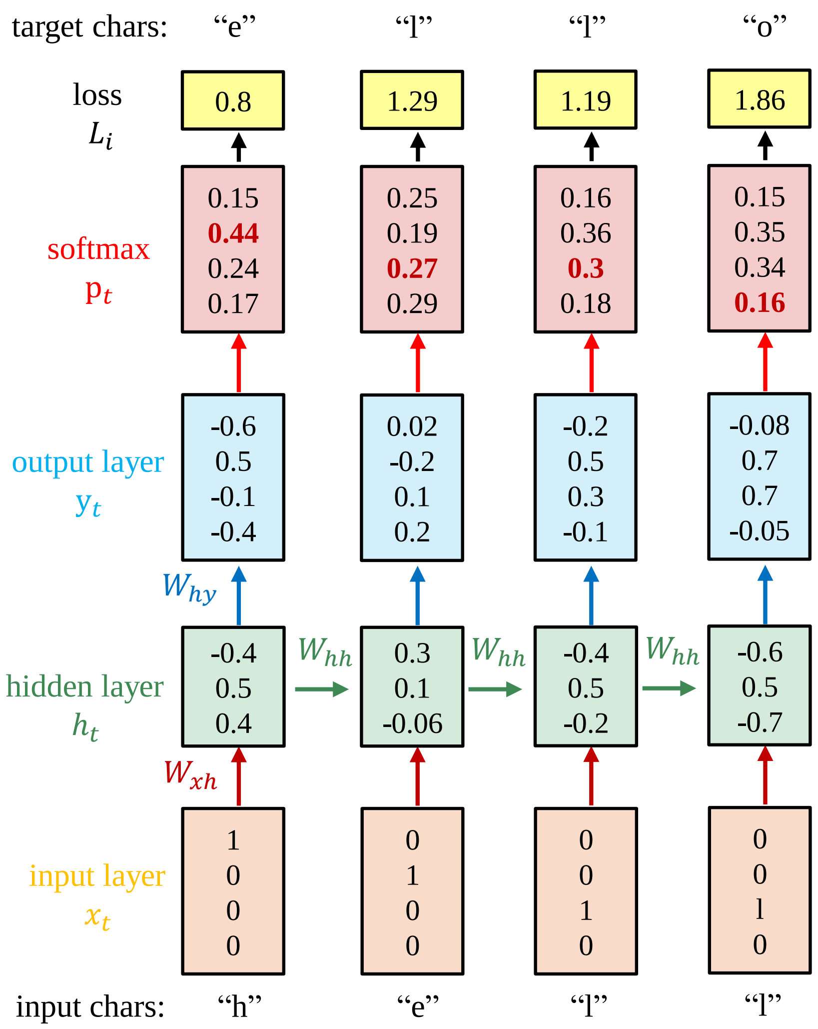


# BACKWARD PASS

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

$$\frac{\partial L}{\partial h_1} = \frac{\partial h_k}{\partial h_{k-1}} \frac{\partial h_{k-1}}{\partial h_{k-2}} \frac{\partial h_{k-2}}{\partial h_{k-3}} \cdots \frac{\partial h_2}{\partial h_1}$$

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



# BACKWARD PASS

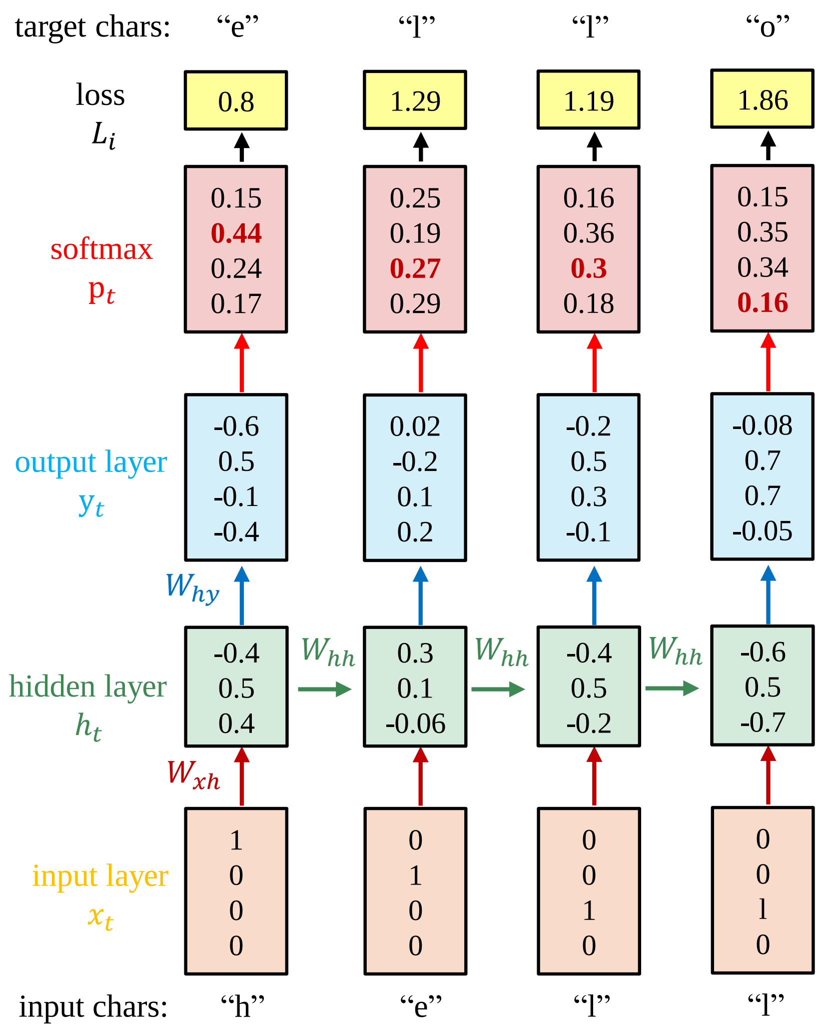
$$\frac{\partial L}{\partial W_{hh}} = \sum_t \frac{\partial L_t}{\partial W_{hh}}$$

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

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

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

...




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

# BACKWARD PASS

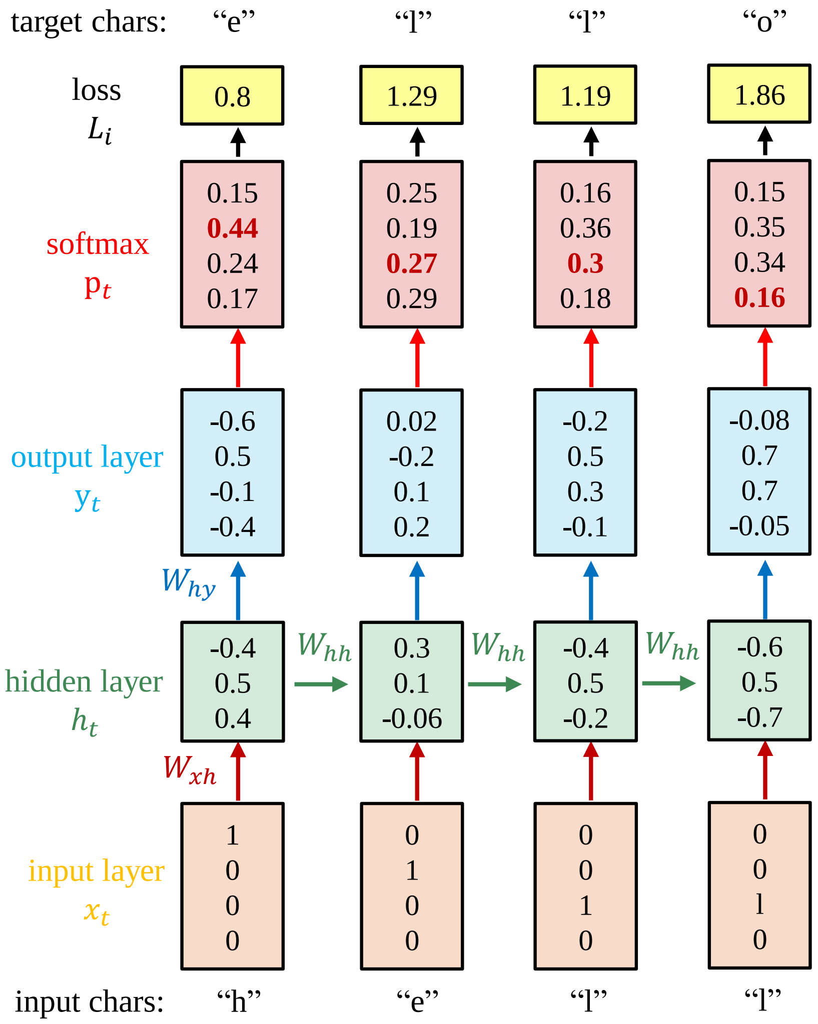
$$\frac{\partial L}{\partial W_{xh}} = \sum_t \frac{\partial L_t}{\partial W_{xh}}$$

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

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

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

…




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

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

# Main loop

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

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

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

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

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, 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
```

# UPDATE PARAMETERS WITH ADAGRAD

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

SGD

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

Adagrad

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

$\epsilon$=1e-8

HTTP://RUDER.IO/OPTIMIZING-GRADIENT-DESCENT/INDEX.HTML#ADAGRAD

# UPDATE PARAMETERS WITH ADAGRAD

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

$\epsilon$=1e-8

$\alpha$=0.1

for iteration 1:

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

$$mW_{xh} \mathrel{+}= \overbrace{\begin{bmatrix} 0.2 & 0.002 & 0.2 & 0 \\ 0.07 & 0.1 & 0.01 & 0 \\ 0.1 & 0.03 & 0.01 & 0 \end{bmatrix}}^{(W_{xh})^2}$$

$$W_{xh} = \begin{bmatrix} -0.5 & 0.4 & -0.2 & 0.3 \\ 0.5 & -0.3 & 0.6 & -0.8 \\ 0.4 & 0.4 & -0.3 & -0.8 \end{bmatrix} - \frac{0.1}{\sqrt{0.00000008 + mW_{xh}}} \begin{bmatrix} 0.5 & 0.04 & -0.5 & 0 \\ -0.2 & -0.3 & 0.1 & 0 \\ -0.4 & 0.1 & 0.1 & 0 \end{bmatrix} = \begin{bmatrix} -0.6 & 0.3 & -0.1 & 0.3 \\ 0.6 & -0.2 & 0.5 & -0.8 \\ 0.5 & 0.3 & -0.4 & -0.8 \end{bmatrix}$$

# UPDATE PARAMETERS WITH ADAGRAD

for iteration 1:

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

$\epsilon$=1e-8

$\alpha$=0.1

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

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

$$W_{hy} = \begin{bmatrix} 0.1 & -0.6 & -0.5 \\ -0.7 & 0.3 & -0.08 \\ 0.1 & 0.4 & -0.7 \\ 0.5 & 0.05 & -0.4 \end{bmatrix} - \frac{0.1}{\sqrt{0.00000008 + mW_{hy}}} \begin{bmatrix} 0.1 & 0.2 & -0.1 \\ -0.06 & 0.1 & -0.5 \\ -0.2 & -0.1 & 0.05 \\ 0.5 & -0.2 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.7 & -0.4 \\ -0.6 & 0.2 & 0.01 \\ 0.2 & 0.5 & -0.8 \\ 0.4 & 0.1 & -0.5 \end{bmatrix}$$

# UPDATE PARAMETERS WITH ADAGRAD

## for iteration 1:

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

$\epsilon$=1e-8

$\alpha$=0.1

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

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

$$W_{hh} = \begin{bmatrix} -0.1 & -0.8 & 0.7 \\ -0.08 & 0.2 & 0.7 \\ 0.5 & -0.5 & 0.01 \end{bmatrix} - \frac{0.1}{\sqrt{0.00000008 + mW_{hh}}} \begin{bmatrix} 0.09 & -0.1 & 0.1 \\ 0.1 & -0.1 & -0.1 \\ -0.05 & 0.1 & 0.07 \end{bmatrix} = \begin{bmatrix} -0.2 & -0.7 & 0.6 \\ -0.1 & 0.3 & 0.8 \\ 0.6 & -0.6 & -0.08 \end{bmatrix}$$

# 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.

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

$\epsilon$=1e-8

$\alpha$=0.1

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

$$b_h = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \frac{0.1}{\sqrt{0.00000008 + mb_h}} \begin{bmatrix} 0.06 \\ -0.5 \\ -0.1 \end{bmatrix} = \begin{bmatrix} -0.09 \\ 0.09 \\ 0.09 \end{bmatrix}$$

# UPDATE PARAMETERS WITH ADAGRAD

## for iteration 1:

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

$\epsilon$=1e-8

$\alpha$=0.1

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

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

$$b_y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} - \frac{0.1}{\sqrt{0.00000008 + mb_y}} \begin{bmatrix} 0.6 \\ 0.3 \\ -0.8 \\ -0.1 \end{bmatrix} = \begin{bmatrix} -0.09 \\ -0.09 \\ 0.09 \\ 0.09 \end{bmatrix}$$

# min-char-rnn.py gist

## Main loop

Test time

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

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

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

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, 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 data pointer
  n += 1 # iteration counter
```

# min-char-rnn.py gist

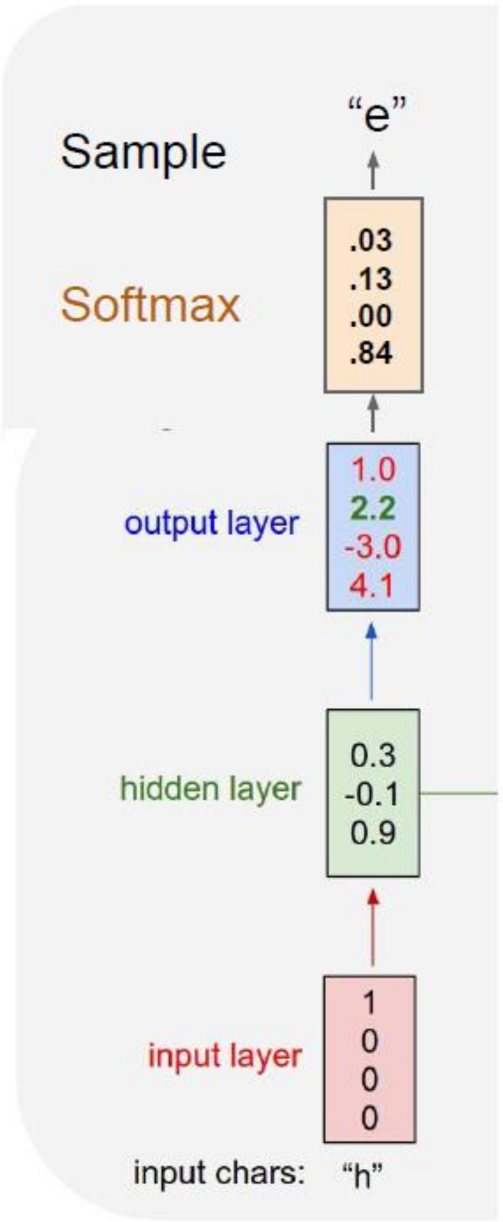
## test-time

```python
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]

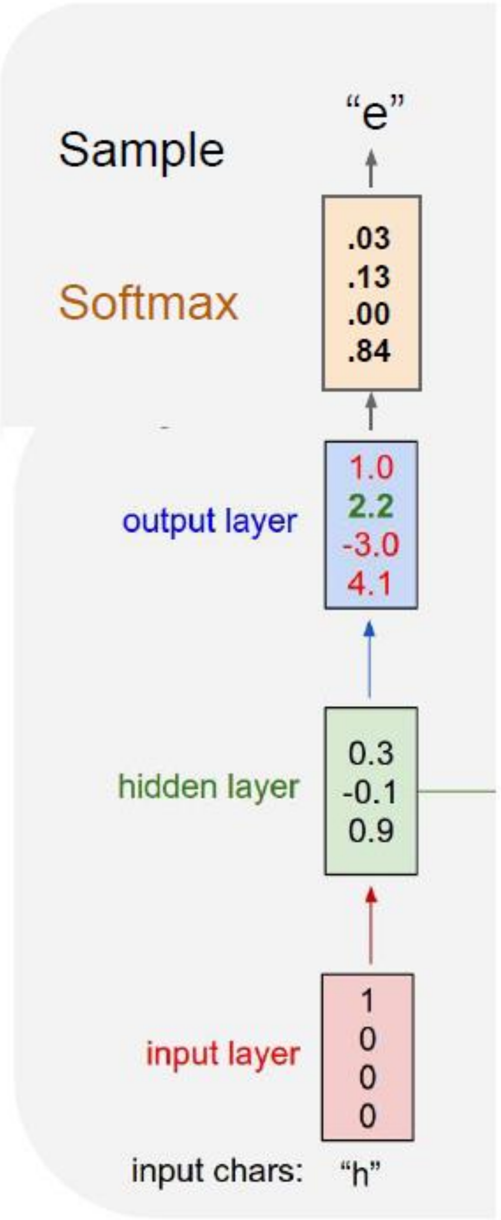
At test-time sample characters one at a time, feed back to model



# Character-level language model example

Vocabulary:
[h,e,l,o]

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



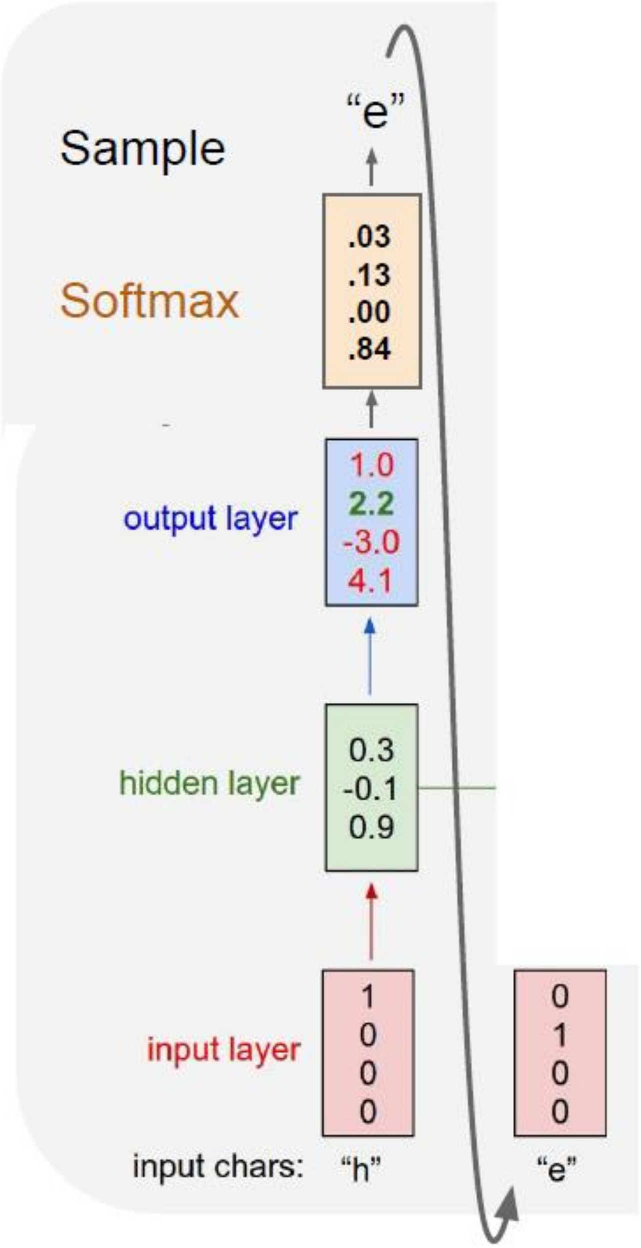

**Why might we sample?**

It lets you get diversity from the model.

# Character-level language model example

Vocabulary:
[h,e,l,o]

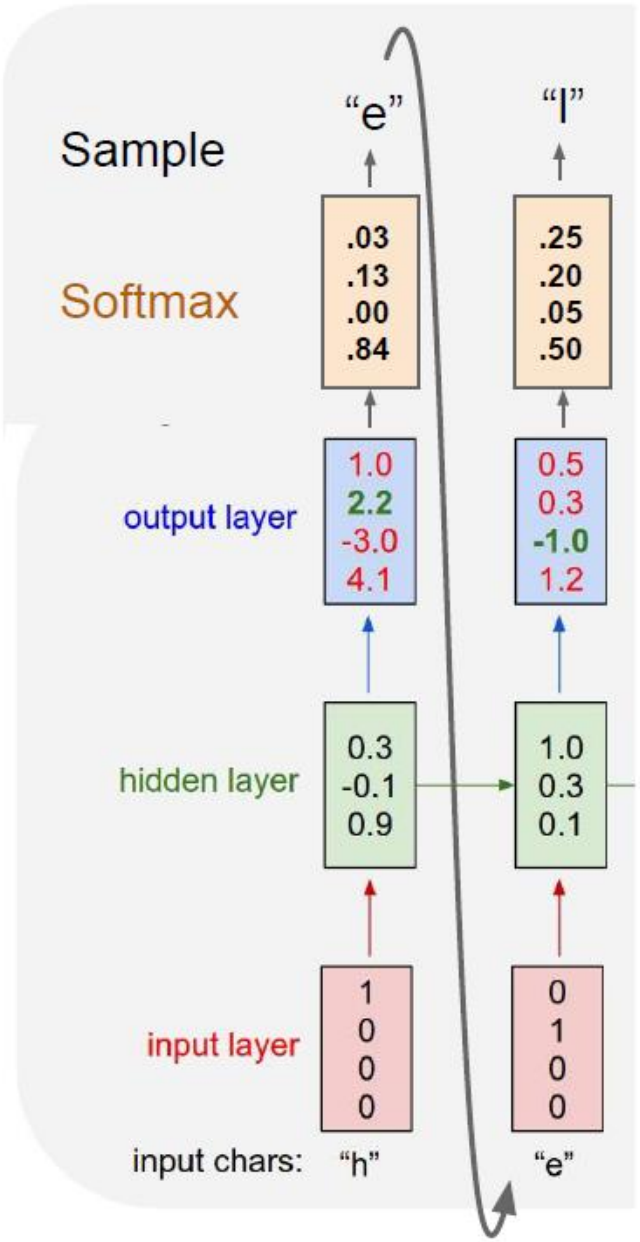
At test-time sample characters one at a time, feed back to model



# Character-level language model example

Vocabulary:
[h,e,l,o]

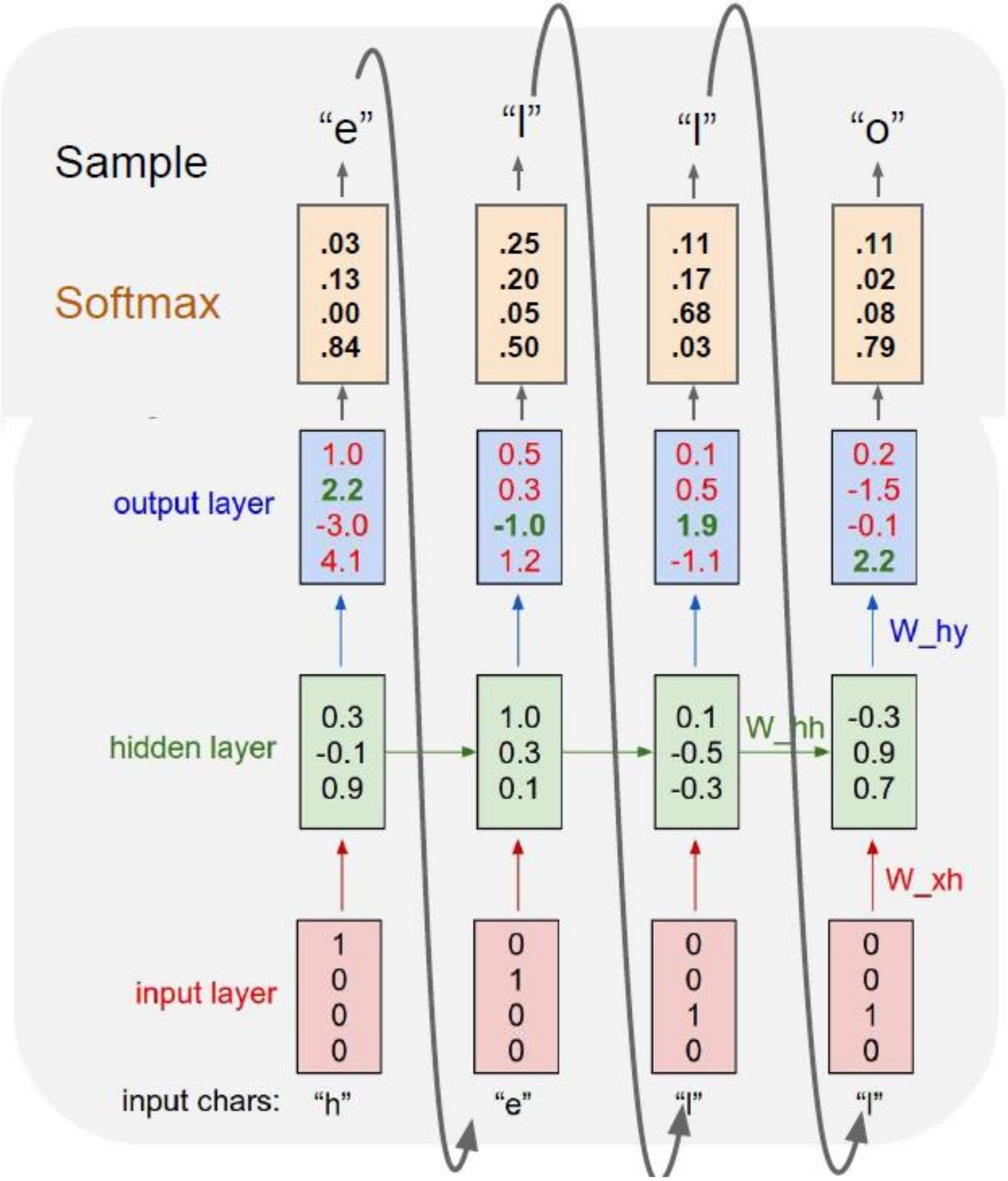
At test-time sample characters one at a time, feed back to model



# Character-level language model example

Vocabulary:
[h,e,l,o]

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



# Thanks