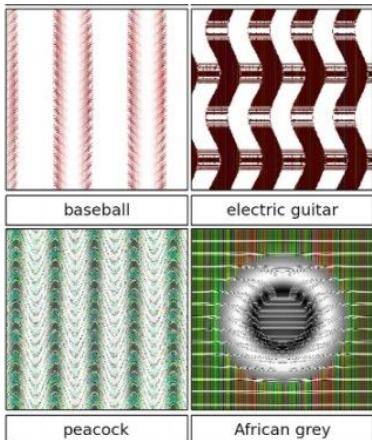
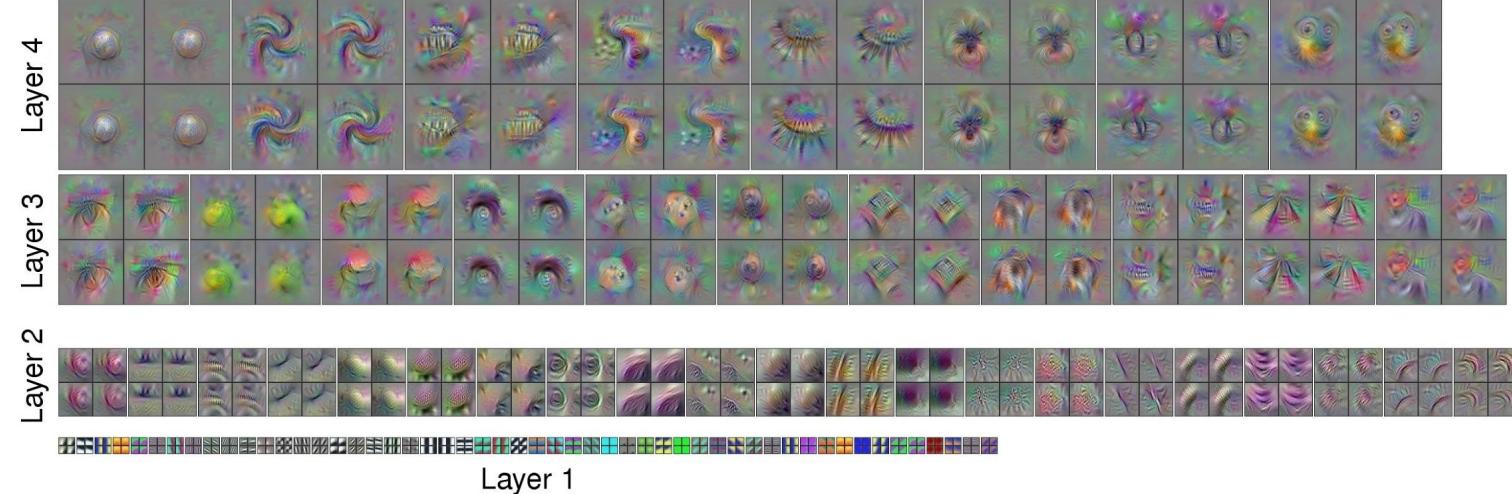
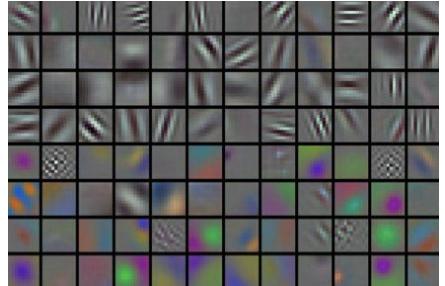


Lecture 10:

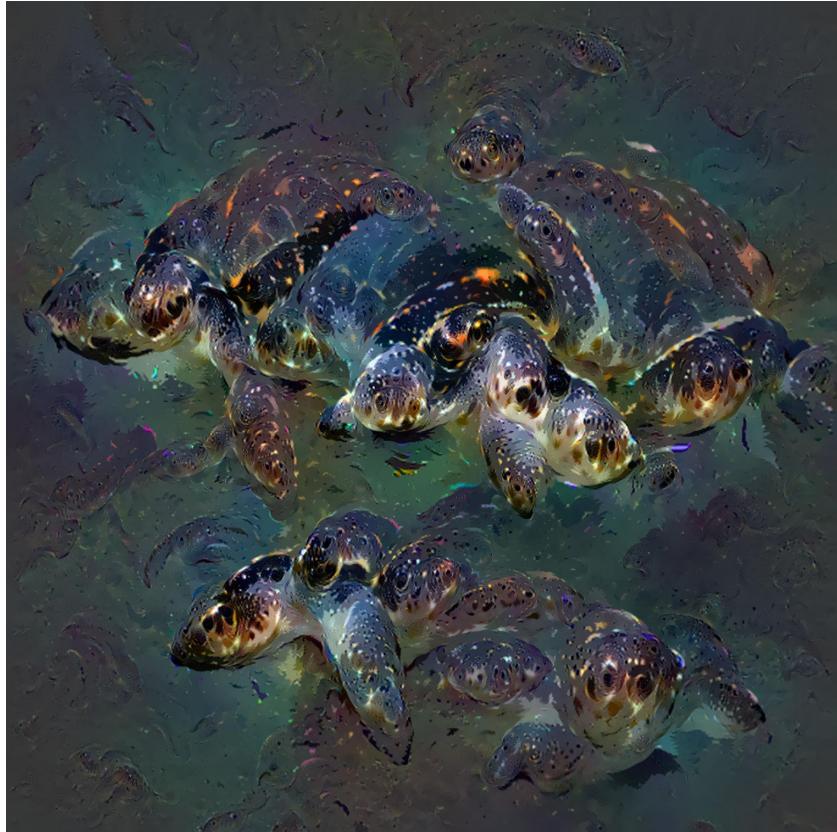
Recurrent Neural Networks

Administrative

- Midterm this Wednesday! woohoo!
- A3 will be out ~Wednesday



<http://mtyka.github.io/deepdream/2016/02/05/bilateral-class-vis.html>

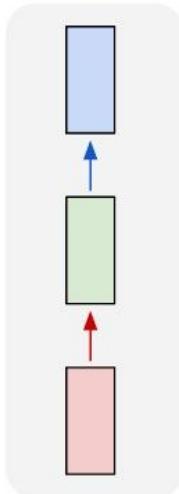


<http://mtyka.github.io/deepdream/2016/02/05/bilateral-class-vis.html>

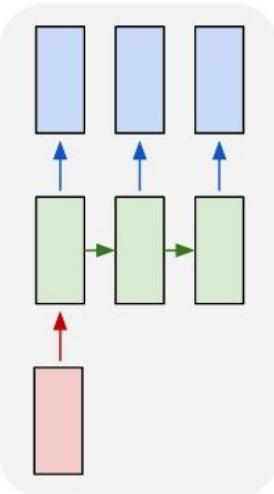


Recurrent Networks offer a lot of flexibility:

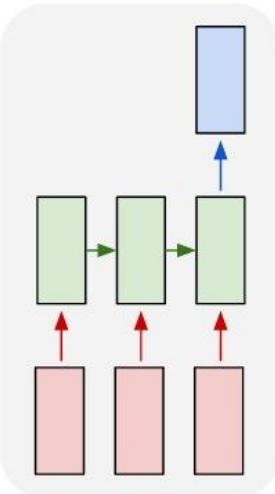
one to one



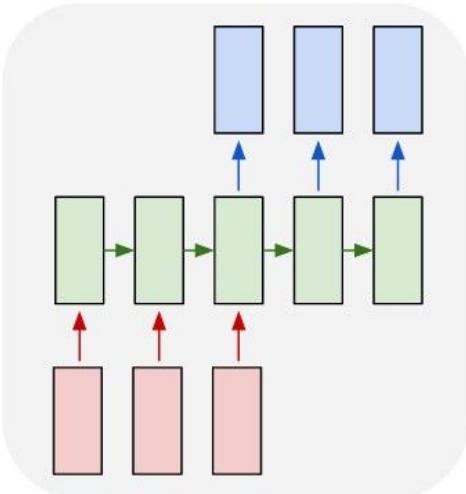
one to many



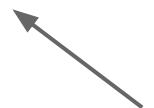
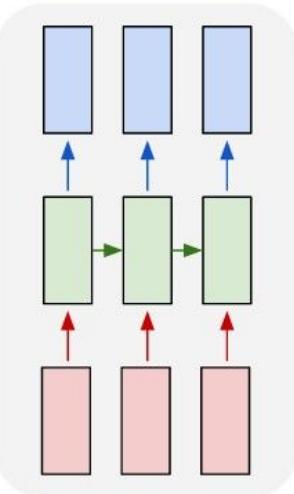
many to one



many to many



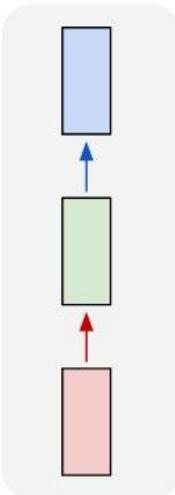
many to many



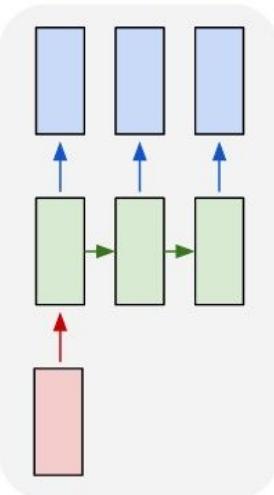
Vanilla Neural Networks

Recurrent Networks offer a lot of flexibility:

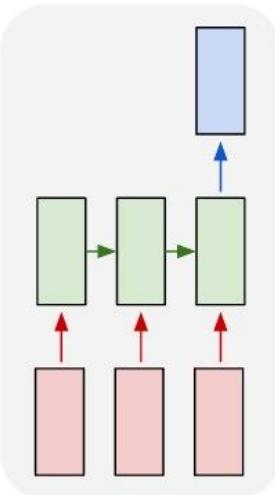
one to one



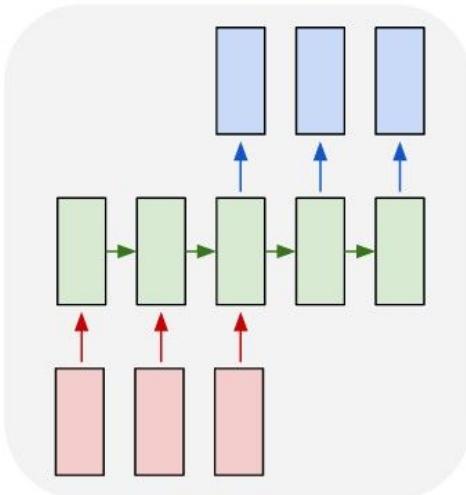
one to many



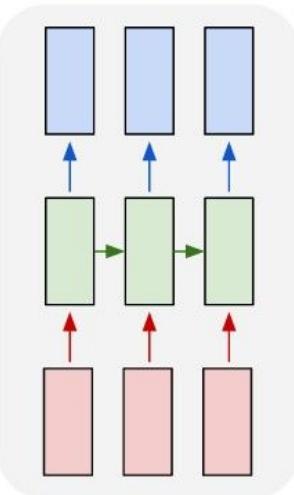
many to one



many to many



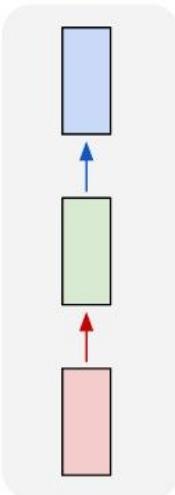
many to many



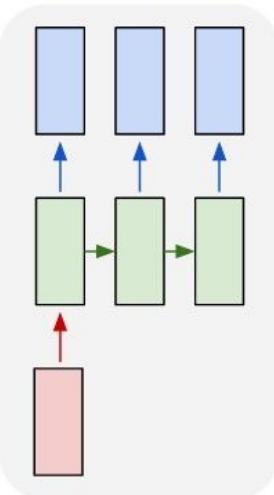
→ e.g. **Image Captioning**
image -> sequence of words

Recurrent Networks offer a lot of flexibility:

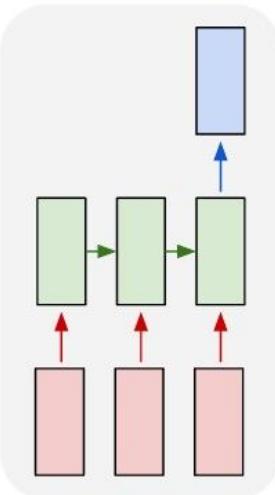
one to one



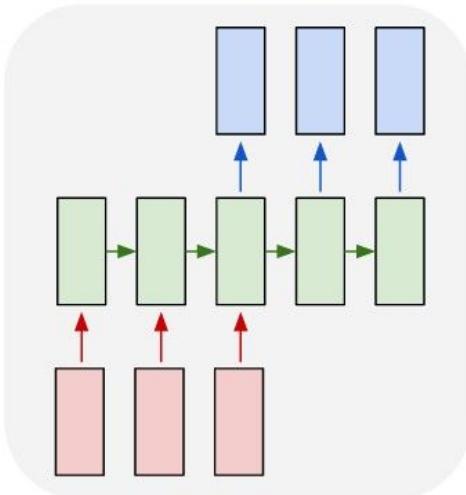
one to many



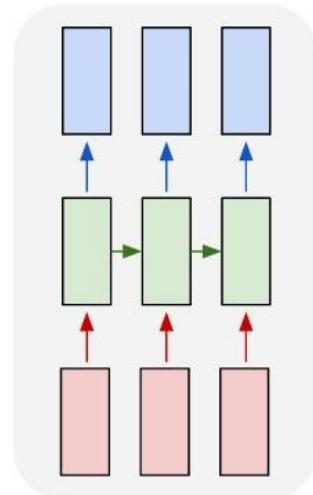
many to one



many to many



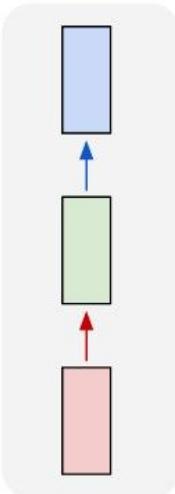
many to many



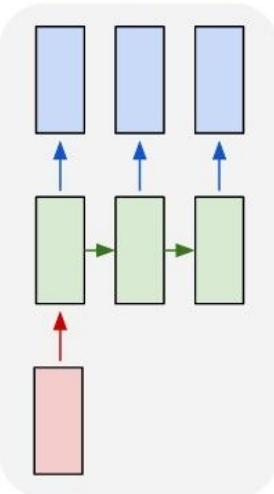
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

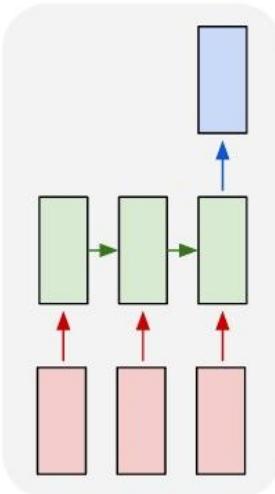
one to one



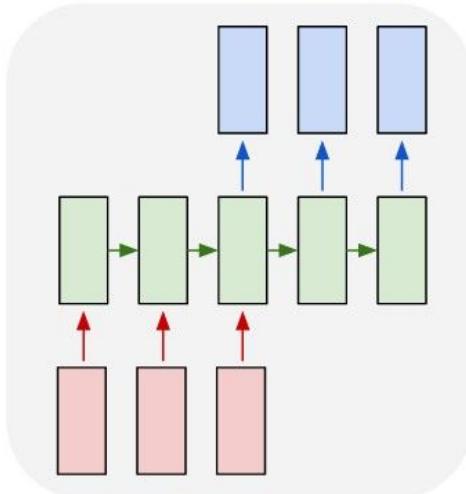
one to many



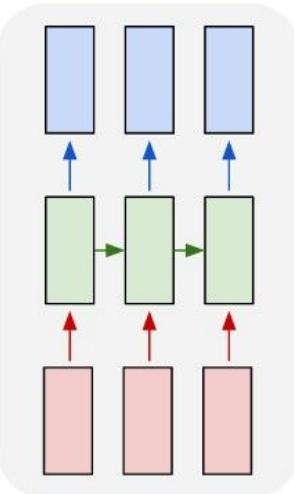
many to one



many to many



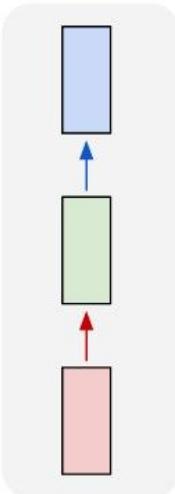
many to many



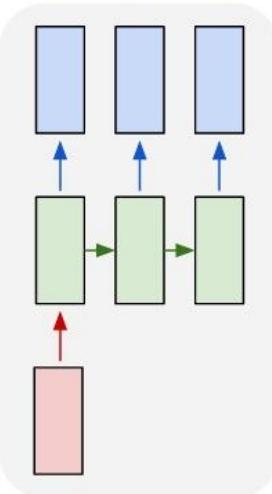
↑
e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Networks offer a lot of flexibility:

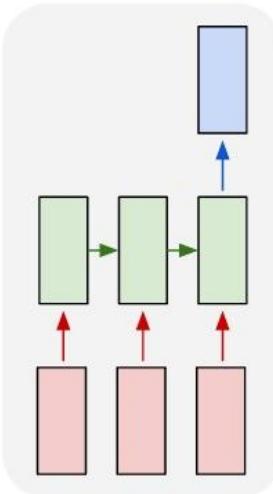
one to one



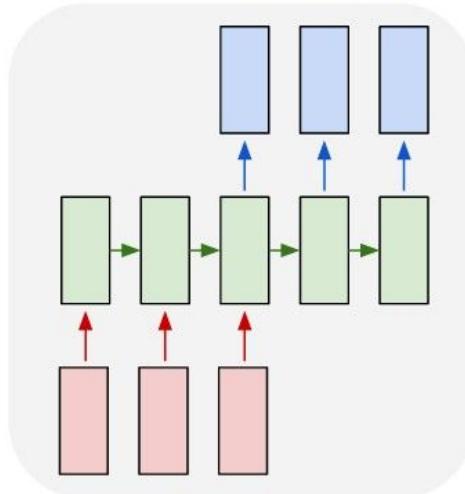
one to many



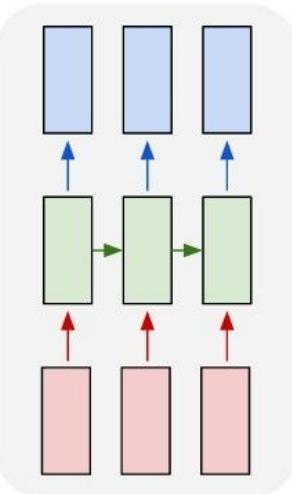
many to one



many to many



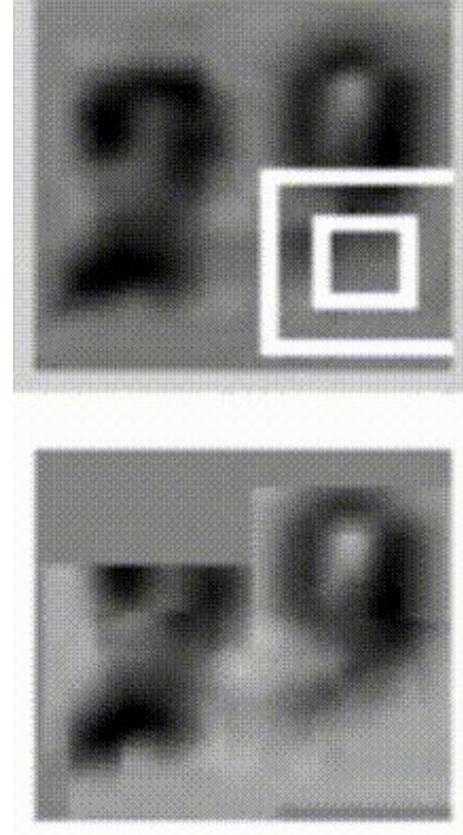
many to many



e.g. Video classification on frame level

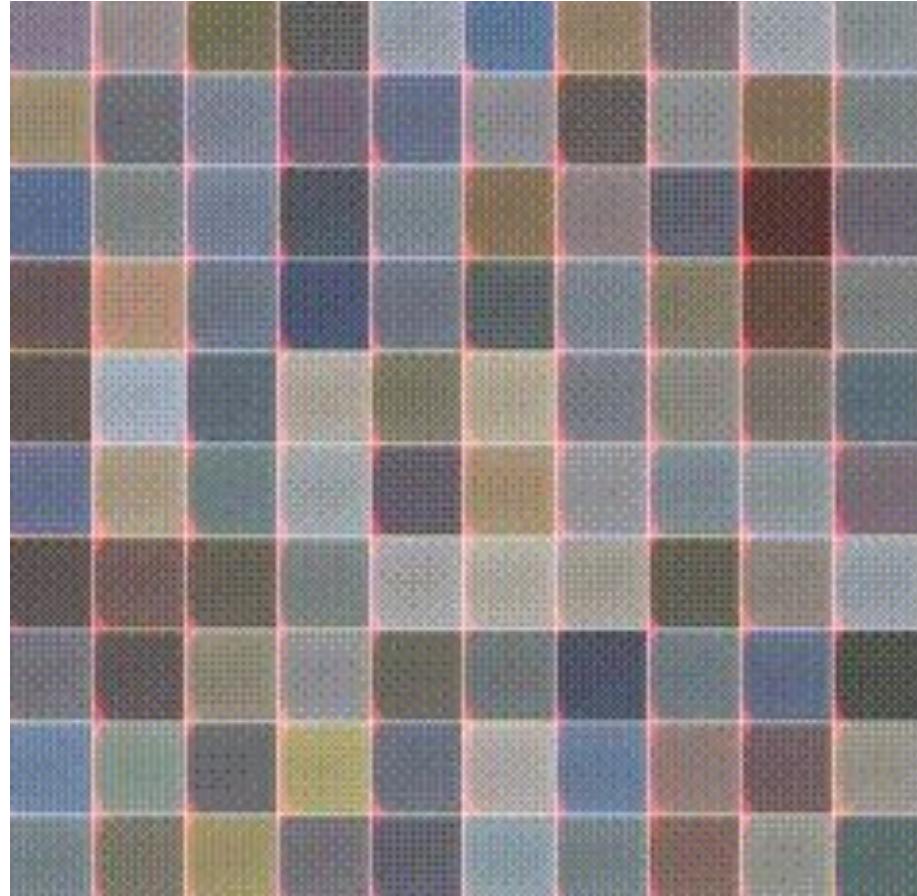
Sequential Processing of fixed inputs

Multiple Object Recognition with
Visual Attention, Ba et al.

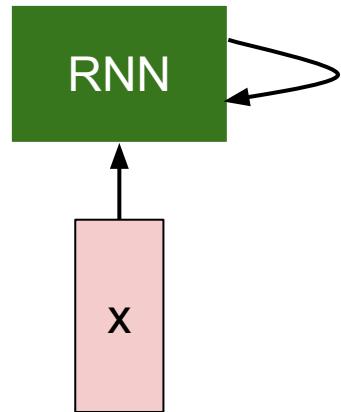


Sequential Processing of fixed outputs

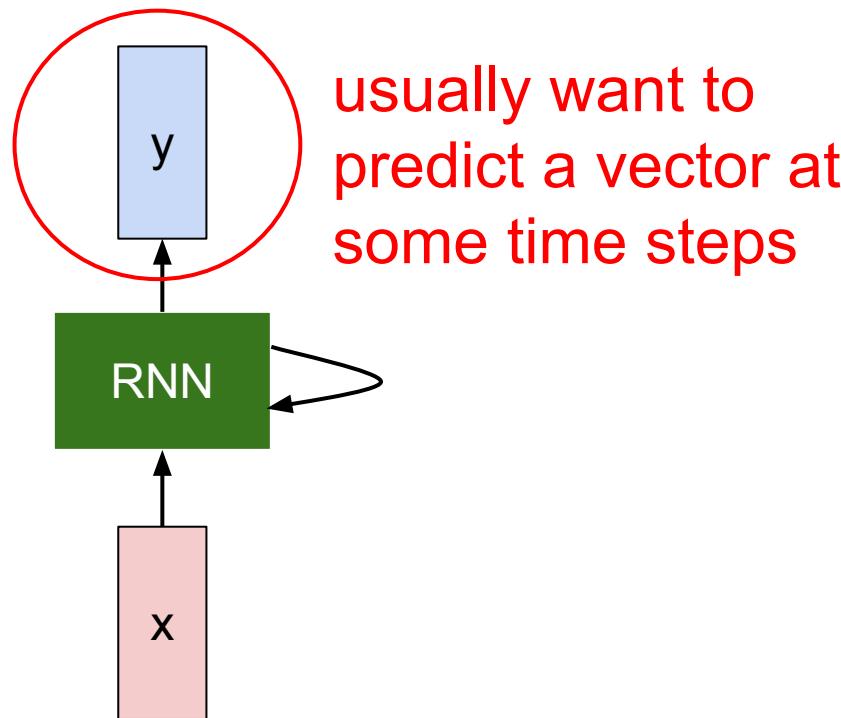
DRAW: A Recurrent
Neural Network For
Image Generation,
Gregor et al.



Recurrent Neural Network



Recurrent Neural Network

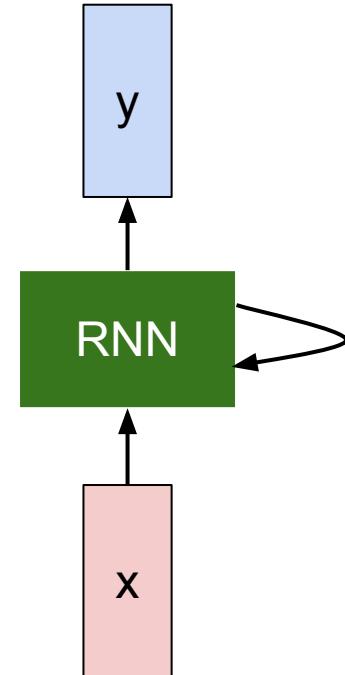


Recurrent Neural Network

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

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

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

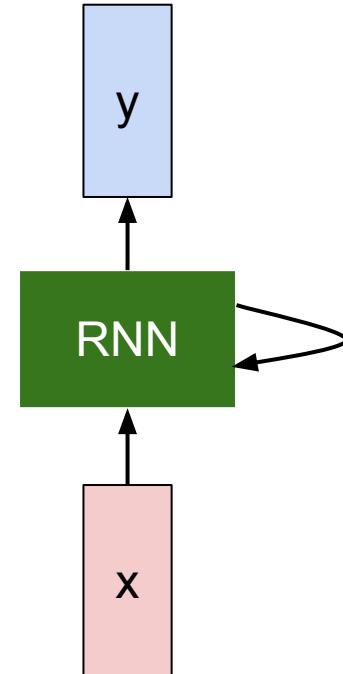


Recurrent Neural Network

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

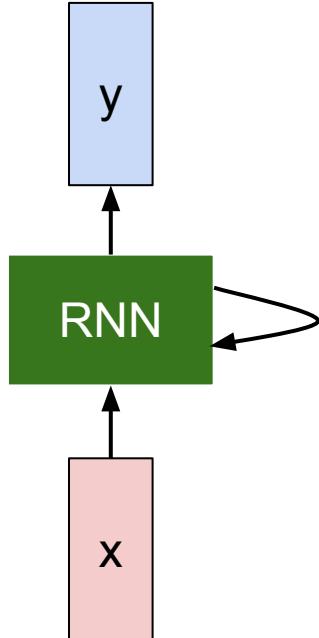
$$h_t = f_W(h_{t-1}, x_t)$$

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



(Vanilla) Recurrent Neural Network

The state consists of a single “*hidden*” vector \mathbf{h} :



$$\mathbf{h}_t = f_W(\mathbf{h}_{t-1}, \mathbf{x}_t)$$



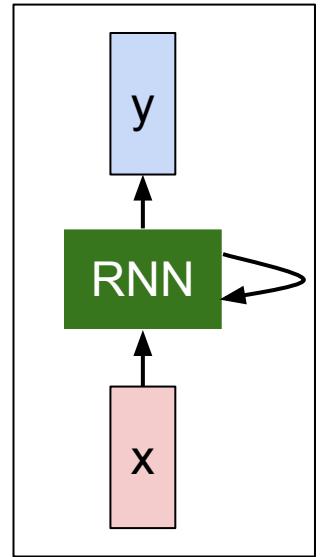
$$\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$$

$$y_t = W_{hy}\mathbf{h}_t$$

Character-level language model example

Vocabulary:
[h,e,l,o]

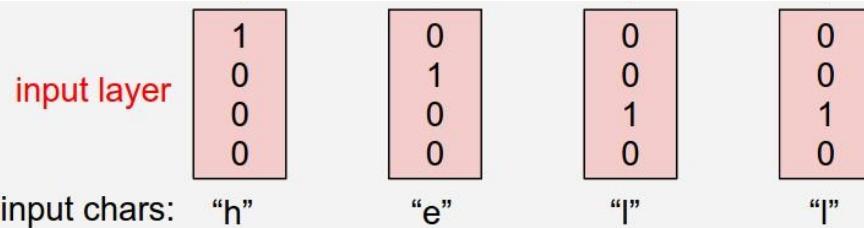
Example training
sequence:
“hello”



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

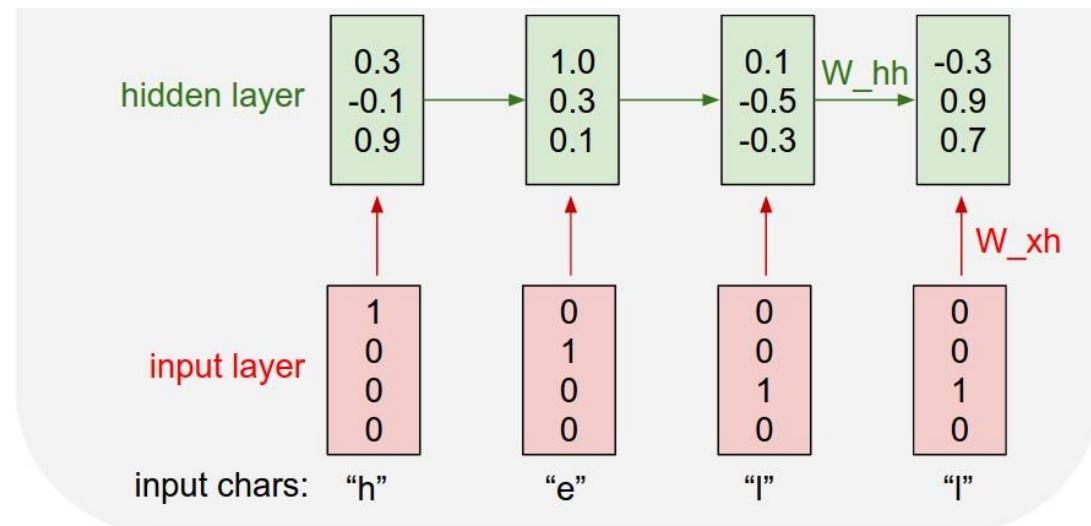


Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

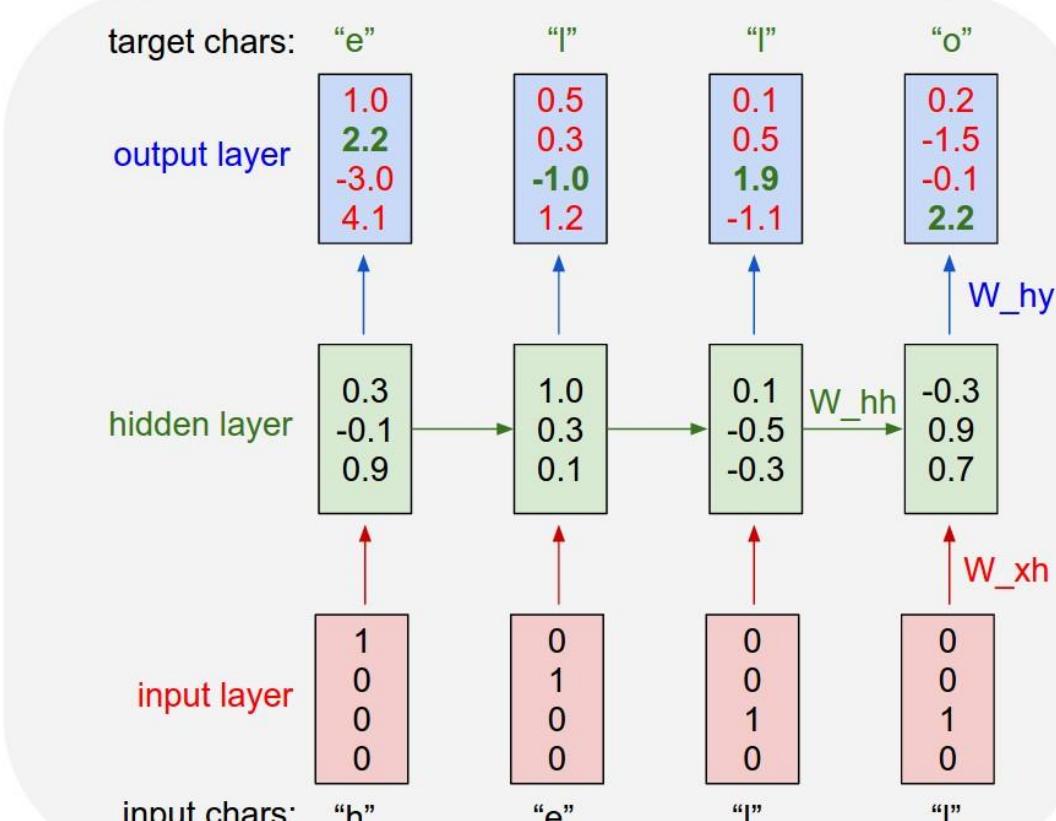
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”



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

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print('data has %d characters, %d unique.' % (data_size, vocab_size))
12 char_to_ix = {ch:i for i,ch in enumerate(chars)}
13 ix_to_char = {i:ch for i,ch in enumerate(chars)}
14
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 wkh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(wkh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) - by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44
45         # backward pass: compute gradients going backwards
46         dwhh, dwhy = np.zeros_like(whh), np.zeros_like(why)
47         dbh, dby = np.zeros_like(bh), np.zeros_like(by)
48         dhnext = np.zeros_like(hs[0])
49         for t2 in reversed(xrange(len(inputs))):
50             dy = np.copy(ps[t2])
51             dy[targets[t2]] -= 1 # backprop into y
52             dby += np.dot(dy, hs[t2].T)
53             dh = np.dot(why.T, dy) + dhnext # backprop into h
54             ddraw = (i - hs[t2].T) * dh # backprop through tanh nonlinearity
55             dbh += ddraw
56             dwhh += np.dot(ddraw, xs[t2].T)
57             dwhy += np.dot(ddraw, hs[t2].T)
58             dhnext = np.dot(whh.T, ddraw)
59             for dparam in [dwhh, dwhy, dbh, dby]:
60                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61
62     return loss, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

```
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(wkh, 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
80     return ixes
81
82 n, p = 0, 0
83 mxwh, mwhh, mmhy = np.zeros_like(wkh), np.zeros_like(whh), np.zeros_like(why)
84 mbh, mbv = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
85 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
86 while True:
87     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size,1)) # reset RNN memory
90         p = 0 # go from start of data
91         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92         targets = [char_to_ix[ch] for ch in data[p+seq_length+1:p+2*seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print '----\n%s\n----' % (txt, )
99
100     # forward seq_length characters through the net and fetch gradient
101     loss, dwhh, dwbh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102     smooth_loss = smooth_loss * 0.999 + loss * 0.001
103     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
104
105     # perform parameter update with Adagrad
106     for param, dparam, mem in zip([wkh, whh, why, bh, by],
107                                   [dwhh, dwbh, dwhy, dbh, dby],
108                                   [mxwh, mwhh, mmhy, mbh, mbv]):
109         mem += dparam * dparam
110         param += -learning_rate * param / np.sqrt(mem + 1e-8) # adagrad update
111
112     p += seq_length # move data pointer
113     n += 1 # iteration counter
```

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

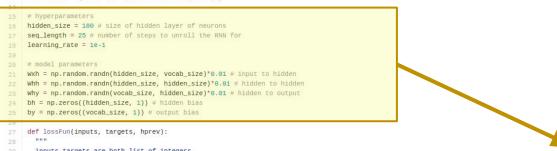
min-char-rnn.py gist

Data I/O

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

```
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) }
```

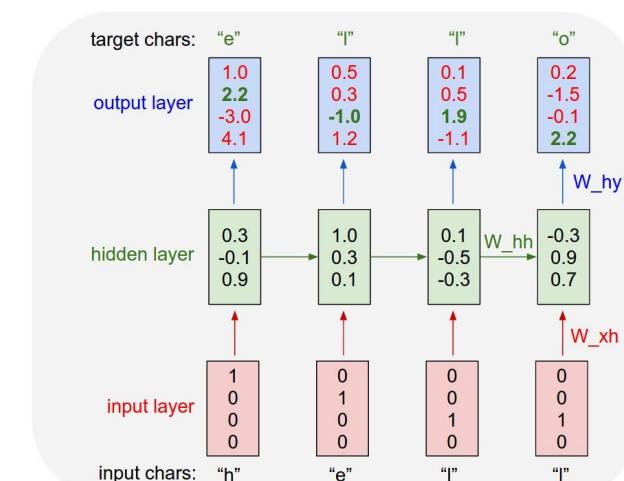
min-char-rnn.py gist



Initializations

```
15 # hyperparameters  
16 hidden_size = 100 # size of hidden layer of neurons  
17 seq_length = 25 # number of steps to unroll the RNN for  
18 learning_rate = 1e-1  
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
```

recal



min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size = len(data)
12 print("data has %d characters, %d unique." % (data_size, vocab_size))
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # 14
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def init_randomhidden(hidden_size, vocab_size)*0.01 # input to hidden
24 wh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
25 why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
26 bh = np.zeros(hidden_size, 1) # hidden bias
27 by = np.zeros(vocab_size, 1) # output bias
28
29 def lossFun(inputs, targets, hprev):
30
31     inputs,targets = both lists of integers.
32
33     hprev is Hx1 array of initial hidden state
34     return the loss, gradients on model parameters, and last hidden state
35
36     xs, hs, ys, ps = O, O, O, O
37     h0 = -1 * np.copy(hprev)
38
39     for t in xrange(seq_length):
40         # encode in 1-of-k representation
41         x_t = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
42         x_t[inputs[t]] = 1
43
44         h_t1 = np.tanh(np.dot(wh, x_t) + np.dot(bh, hs[-1]) + bh) # hidden state
45         ps_t = np.exp(x_t) / np.sum(np.exp(x_t)) = softmax # probabilities for next chars
46         loss += -np.exp(y[t]) / np.sum(np.exp(y[t])) = cross-entropy loss
47
48         # backprop through tanh nonlinearity
49         dprev, dh0, dy, dwhy, dbh, dby, dwh, dby = backward pass
50         dprev = np.zeros_like(h0)
51         dh0 = np.zeros_like(h0)
52         dy = np.copy(ps_t)
53         dy[targets[t]] = 1 # a backward pass y
54         dprev += dy * dprev
55         dh0 += dy * dh0
56         dwh += np.dot(dy.T, x_t) + dh0 # backprop through tanh nonlinearity
57         dby += np.dot(dy, np.ones((vocab_size, 1)))
58         dbh += np.dot(dy, np.zeros((hidden_size, 1)))
59         dwhy += np.zeros_like(why)
60
61         for dparam in [dwh, dwhy, dbh, dby]:
62             np.clip(dparam, -5, 5, out=dparam) = clip to mitigate exploding gradients
63             rangrad = np.random.randn(*dparam.shape) * 0.01
64             dparam -= learning_rate * dparam + np.sqrt(mem) * rangrad
65             mem += dparam * dparam
66
67     smooth_loss = smooth_loss * 0.999 + loss * 0.001
68
69     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
70
71     # sample from the model now and then
72     if n % 1000 == 0:
73         sample_ix = sample(hprev, inputs[0], 200)
74         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
75         print '----\n %s ----' % (txt, )
76
77     # forward seq_length characters through the net and fetch gradient
78     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
79     smooth_loss = smooth_loss * 0.999 + loss * 0.001
80
81     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
82
83     # perform parameter update with Adagrad
84     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
85                                   [dWxh, dWhh, dWhy, dbh, dby],
86                                   [mWxh, mWhh, mWhy, mbh, mby]):
87
88         mem += dparam * dparam
89         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
90
91     p += seq_length # move data pointer
92
93     n += 1 # iteration counter
94
```

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

min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} 
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # Model parameters
22 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
23 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
24 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
25
26 while True:
27     if p+seq_length+1 >= len(data) or n == 0:
28         hprev = np.zeros((hidden_size,1)) # reset RNN memory
29         p = 0 # go from start of data
30         inputs, targets = [ord(c) for c in data[p:p+seq_length]] # encode as 1-of-k representation
31         inputs = np.array(inputs).reshape((1, seq_length)) # encode in 1-of-k representation
32         targets = np.array(targets).reshape((1, seq_length))
33         x1 = inputs[0]
34         h1 = np.tanh(np.dot(Wxh, x1) + np.dot(Whh, h1[-1]) + bh) # hidden state
35         y1 = np.exp(np.dot(Why, h1)) / np.sum(np.exp(y1)) # softmax (cross-entropy loss)
36         loss = -np.log(y1[targets[0][0]]) # cross-entropy loss
37         dx1 = np.dot(Why.T, np.zeros_like(y1)) # backprop into x
38         dh1 = np.zeros_like(h1) # backprop into h
39         dWxh += np.outer(x1, h1) # compute gradients
40         dWhh += np.outer(h1, h1)
41         dbh += np.zeros_like(bh)
42         dWhy += np.outer(h1, np.zeros_like(y1))
43         for i in range(1, seq_length):
44             x1 = inputs[0][i]
45             dy1 = np.copy(y1)
46             dy1[targets[0][i]] -= 1 # backprop into y
47             dh1 = np.tanh(dy1 * dh1) # backprop through tanh nonlinearity
48             dWxh += np.outer(x1, dh1) # backprop into x
49             dWhh += np.outer(dh1, dh1)
50             dbh += np.outer(dh1, np.zeros_like(bh))
51             dWhy += np.outer(dh1, np.zeros_like(y1))
52             dh1 = np.tanh(dy1 * dh1) # backprop through tanh nonlinearity
53             for opname in [dWxh, dWhh, dbh, dWhy]:
54                 np.clip(opname, -5, out=opname) # clip to mitigate exploding gradients
55             if np.isnan(dWxh).any() or np.isnan(dWhh).any() or np.isnan(dbh).any() or np.isnan(dWhy).any():
56                 raise ValueError("NaN in gradient computation")
57             dWxh, dWhh, dbh, dWhy, mem = adamUpdate(dWxh, dWhh, dbh, dWhy, mem, inputs[0])
58             smooth_loss = smooth_loss * 0.999 + loss * 0.001
59             if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
60             if n % 1000 == 0: print '----\n%s\n----' % (txt, )
61             if n % 10000 == 0: print 'Saving state'
62             np.savez('state.npz', seed=ix, n=n)
63
64     # Sample a sequence of integers from the model
65     h = np.zeros((hidden_size,1)) # h is memory state, seed is first input for first time step
66     x = np.zeros((vocab_size, 1))
67     x[seed_ix] = 1
68
69    for t in range(seq_length):
70        h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
71        y = np.exp(np.dot(Why, h))
72        p = np.random.choice(range(vocab_size), p=y.ravel())
73        ix = np.argmax(chars[p])
74        x[seed_ix] = 0
75        x[i] = 1
76
77    return ixes
```



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

min-char-rnn.py gist

Main loop

```
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(data[1:(1)])
10 vocab_size = len(chars), len(data), len(chars)
11 print("data has %d characters, %d unique." % (data_size, vocab_size))
12 char_to_ix = {ch:i for i in range(len(chars))}
13 ix_to_char = {i:ch for ch in range(len(chars))} # 14
14
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
22 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
23 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
24
25 while True:
26     # prepare inputs (we're sweeping from left to right in steps seq_length long)
27     if p+seq_length+1 >= len(data) or n == 0:
28         hprev = np.zeros((hidden_size,1)) # reset RNN memory
29         p = 0 # go from start of data
30
31     inputs, targets = data[p:p+seq_length], data[p+1:p+seq_length+1]
32
33     xs, hs, ys, ps = [0, 0, 0, 0]
34     hprev = np.copy(hprev)
35     loss = 0
36
37     for t in range(seq_length):
38         # forward pass
39         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
40         x[inputs[t]] = 1
41         hprev = np.tanh(np.dot(wh, hprev) + np.dot(bh, hprev[-1]) + bh)
42         y = np.exp(hprev)
43         ps = np.exp(y)/np.sum(np.exp(y)) # probabilities for next chars
44         loss += -np.log(ps[targets[t]]/np.sum(np.exp(y))) # softmax (cross-entropy loss)
45
46         # backward pass: compute gradients to params
47         dWxh += np.outer(x, hprev)
48         dbh += np.zeros_like(bh)
49         dWhy += np.outer(hprev, np.zeros_like(why))
50         dWxh, dbh, dWhy = np.zeros_like(dWxh), np.zeros_like(dbh), np.zeros_like(dWhy)
51         dWhh += np.outer(hprev, hprev)
52         dWxh += np.outer(x, hprev)
53         dy = np.copy(ps[1:])
54         dy[targets[t]] -= 1 # backprop into y
55         dh = np.zeros_like(hprev)
56         dh += np.dot(why.T, dy) + dWhh # backprop into h
57         dh += np.dot(dWxh.T, x) * hprev # backprop through tanh nonlinearity
58         dWxh += np.outer(dh, x)
59         dbh += np.sum(dh, axis=0, keepdims=True) / seq_length
60         dWhh += np.outer(dh, hprev)
61
62         for dparam in [dWxh, dbh, dWhy, dWhh, dy]:
63             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
64             if dparam != dy:
65                 dparam *= learning_rate
66
67         if n > 0:
68             mem += dparam * dparam
69             dparam *= learning_rate / np.sqrt(mem + 1e-8) # adagrad update
70
71         dWxh, Whh, Why, bh, by = dWxh, dWhh, dWhy, dbh, dby
72
73         if n % 100 == 0:
74             sample_ix = sample(hprev, inputs[0], 200)
75             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
76             print '----\n %s ----' % (txt, )
77
78
79     # forward seq_length characters through the net and fetch gradient
80     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
81     smooth_loss = smooth_loss * 0.999 + loss * 0.001
82
83     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
84
85
86     # perform parameter update with Adagrad
87     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
88                                  [dWxh, dWhh, dWhy, dbh, dby],
89                                  [mWxh, mWhh, mWhy, mbh, mby]):
90
91         mem += dparam * dparam
92         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
93
94
95     p += seq_length # move data pointer
96     n += 1 # iteration counter
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
```

min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # 14
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # Model parameters
22 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
23 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
24 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
25
26 while True:
27     # Prepare inputs (we're sweeping from left to right in steps seq_length long)
28     if p+seq_length+1 >= len(data) or n == 0:
29         hprev = np.zeros((hidden_size,1)) # reset RNN memory
30         p = 0 # go from start of data
31         inputs = [c for c in data[p:p+seq_length]] # 31
32         targets = [c for c in data[p+1:p+seq_length+1]]
33
34     x, hs, ys, ps = o, O, O, O
35     hprev = np.copy(hprev)
36     loss = 0
37
38     for t in range(seq_length):
39         x = np.zeros((vocab_size,1)) # encode in 1-of-k representation
40         x[0][char_to_ix[inputs[t]]] = 1
41         hprev = np.tanh(np.dot(wh, hprev) + np.dot(bh, hprev[-1]) + bh) # hidden state
42         y = np.dot(Wxh, hprev) + bh # output neuron
43         yprob = np.exp(y)/np.sum(np.exp(y)) # probabilities for next chars
44         loss += -np.log(yprob[targets[t]]) # softmax (cross-entropy loss)
45
46         dy = np.copy(ps) # 46
47         dy[targets[t]] -= 1 # backprop into y
48         dh = np.dot(dy, Wxh.T) # 48
49         dh += np.dot(dy, bh.T) # dh = backprop through tanh nonlinearity
50         dh += dh * (1-hprev**2) # 50
51         dWxh += np.dot(inputs[t].T, dy) # 51
52         dbh += np.sum(dy, axis=0) # 52
53         dWhh += np.dot(hprev.T, dy) # 53
54         dWhy += np.sum(dy*x, axis=0) # 54
55
56         dx = np.dot(dy, Wxh) # 56
57         dh = np.tanh(dx) # 57
58         dh *= (1-dh**2) # 58
59
60         for dparam in [dWxh, dbh, dWhh, dWhy]:
61             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
62             dparam *= learning_rate # 62
63             dbh *= learning_rate # 63
64             dWhh *= learning_rate # 64
65             dWhy *= learning_rate # 65
66
67         dWxh, dbh, dWhh, dWhy, hprev, hprev = lossFun(inputs, targets, hprev)
68         smooth_loss = smooth_loss * 0.999 + loss * 0.001
69
70         if n % 100 == 0: print '----\n%d %f' % (n, smooth_loss) # print progress
71
72         sample_ix = sample(hprev, inputs[0], 200) # 72
73         txt = ''.join(ix_to_char[ix] for ix in sample_ix) # 73
74         print '----\n%s ----' % (txt, )
75
76
77     # forward seq_length characters through the net and fetch gradient
78     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
79     smooth_loss = smooth_loss * 0.999 + loss * 0.001
80     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
81
82     # perform parameter update with Adagrad
83     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
84                                 [dWxh, dWhh, dWhy, dbh, dby],
85                                 [mWxh, mWhh, mWhy, mbh, mby]):
86         mem += dparam * dparam
87         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
88
89     p += seq_length # move data pointer
90     n += 1 # iteration counter
```

Main loop



min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(data[1:(1)])
10 vocab_size = len(chars), len(data), len(chars)
11 print("data has %d characters, %d unique." % (data_size, vocab_size))
12 char_to_ix = {ch:i for i in range(len(chars))}
13 ix_to_char = {i:ch for ch in range(len(chars))} 
14
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # Model parameters
21 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
22 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
23 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
24
25 while True:
26     # prepare inputs (we're sweeping from left to right in steps seq_length long)
27     if p+seq_length+1 >= len(data) or n == 0:
28         hprev = np.zeros((hidden_size,1)) # reset RNN memory
29         p = 0 # go from start of data
30         inputs, targets = [char_to_ix[ch] for ch in data[p:p+seq_length]]
31         hprev = np.copy(hprev)
32
33     xs, hs, ys, ps = np.zeros((O, O, O, O))
34     h1_t1 = np.copy(hprev)
35     loss = 0
36
37     # forward pass
38     for t in range(seq_length):
39         # encode in 1-of-k representation
40         x1_t1 = np.zeros((vocab_size,1)) + encode_in_1-of-k_representation(chars[t])
41         h1_t1 = np.tanh(np.dot(wh, x1_t1) + np.dot(bh, h1_t1) + bh) # hidden state
42         ps1_t1 = np.exp(ps1_t1) / np.sum(np.exp(ps1_t1)) = softmax # probabilities for next chars
43         loss += -np.exp(ps1_t1)[targets[t]] / np.sum(np.exp(ps1_t1)) = softmax # (cross-entropy loss)
44
45         # backward pass: compute gradients
46         dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
47         dbh, dby = np.zeros_like(bh), np.zeros_like(by)
48         dnextx = np.zeros_like(x1_t1)
49         for i in range(O):
50             dy = np.copy(ps1_t1)
51             dy[targets[t]] -= 1 # backprop into y
52             dh = np.dot(dy, dbh) # backprop into h
53             dh += np.dot(dy, Wxh.T) # dh = backprop through tanh nonlinearity
54             dh += np.dot(dy, wh * h1_t1) # dh = backprop through dot product
55             dWxh += np.dot(dy, x1_t1.T)
56             dbh += np.sum(dy * dh, axis=1).T
57             dWhh += np.dot(dy, h1_t1.T)
58             dh1_t1 = np.dot(dy, wh)
59
60             for dparam in [dWxh, dWhh, dbh, dby]:
61                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
62             if dparam != 0:
63                 dparam *= learning_rate # update parameters
64
65         if n % 100 == 0:
66             print('----\n%d %s ----' % (n, smooth_loss))
67
68     # sample from the model now and then
69     if n % 100 == 0:
70         sample_ix = sample(hprev, inputs[0], 200)
71         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
72         print('----\n%s ----' % (txt, ))
73
74     # forward seq_length characters through the net and fetch gradient
75     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
76     smooth_loss = smooth_loss * 0.999 + loss * 0.001
77
78     if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
79
80     # perform parameter update with Adagrad
81     n, p = 0, 0
82     mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83     mbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84
85     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
86                                   [dWxh, dWhh, dWhy, dbh, dby],
87                                   [mWxh, mWhh, mWhy, mbh, mby]):
88
89         mem += dparam * dparam
90         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
91
92         p += seq_length # move data pointer
93
94         n += 1 # iteration counter
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
```

Main loop



min-char-rnn.py gist

```
***  
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)  
BSD License  
***  
Import numpy as np  
  
# Data I/O  
data = open('train.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 ch in enumerate(chars)}  
  
# Hyperparameters  
hidden_size = 100 # size of hidden layer of neurons  
seq_length = 20 # number of steps to unroll the RNN for  
learning_rate = 1e-1  
  
# Model parameters  
wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden  
bh = 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  
bh0 = 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  
    for t in xrange(len(inputs)):  
        xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation  
        xs[t][inputs[t]] = 1  
        wht = np.tanh(np.dot(wh, xs[t]) + np.dot(bh, hs[t-1]) + bh) # hidden state  
        whyt = np.dot(why, wht) + by # unnormalized log probabilities for next chars  
        ps[t] = np.exp(whyt) / np.sum(np.exp(whyt)) # softmax (cross-entropy loss)  
        loss += -np.log(ps[t][targets[t],0])  
        dy = np.copy(ps[t])  
        dy[targets[t]] -= 1 # backprop into y  
        dhnext = np.zeros_like(hs[t])  
        dy *= dy  
        dhy = np.dot(dy, wh.T)  
        dh = np.dot(dhy, bh) + dhnext # backprop through tanh nonlinearity  
        dbh += np.sum(dhy, axis=0)  
        dwh = np.dot(dy, xs[t].T)  
        dhyt = np.dot(why, dy) + by  
        dhyt += np.sum(dhyt, axis=1, keepdims=True)  
        dhyt *= 1.0 / float(len(inputs))  
        dhyt -= np.mean(dhyt)  
        for dparam in [dwh, dbh, dhy]:  
            np.clip(dpараметre, -5, 5, out=dpараметре) # clip to mitigate exploding gradients  
        dhnext = dhyt  
    return loss, xs, hs, ys, ps  
  
def sample(hx, ix):  
    """  
    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  
  
    for t in xrange(n):  
        h = np.tanh(np.dot(wh, x) + np.dot(bh, h) + bh)  
        p = np.exp(why * np.dot(h, wh))  
        ix = np.argmax(np.random.multinomial(1, p.ravel()))  
        x[0:1] = np.zeros((vocab_size, 1))  
        x[i] = 1  
  
    return ix  
  
n, p, t, b  
moch, mohy = np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(why)  
mohw, mohy, mohb = np.zeros_like(bh), np.zeros_like(why), np.zeros_like(bh)  
smooth_loss = 0  
for i in range(seq_length):  
    if i > 0:  
        print 'resetting from left to right in step', i, 'iteration', t  
    if p+seq_length > len(data) or t == 0:  
        np.random.seed(1000000000+i)  
        h = np.zeros((hidden_size, 1))  
        inputs = [char_to_ix[ch] for ch in data[p:seq_length]]  
        targets = [char_to_ix[ch] for ch in data[p+seq_length:]]  
  
    if t > 0:  
        print 'resetting from the model now and then'  
        n = 1000000000  
        sample_ix = sample(hprev, inputs[0], 200)  
        ix = np.argmax(ix_to_char[sample_ix])  
        print 'resetting from the model now and then'  
  
    # Forward seq length characters through the net and fetch gradient  
    loss, dh, dwh, dbh, dhy, hprev = lossFun(inputs, targets, hprev)  
    smooth_loss = smooth_loss * 0.999 + loss * 0.001  
    t += 1  
    if t % 100 == 0:  
        print 'iter %d, loss: %f' % (t, smooth_loss) # print progress  
  
    # For parameter, mem in zip(wh, wh, why, why, bh, bh, dbh, dbh, dhy, dhy):  
    #     (dhv, dwh, dbh, dhy, dhy, dbh, dbh, dhy, dhy);  
    #     mem += dparam  
    #     mem -= learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update  
    #     mem += dparam  
  
    p = seq_length # move data pointer  
    n += 1 # iteration counter
```

Loss function

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

```
27 def lossFun(inputs, targets, hprev):  
28     """  
29         inputs,targets are both list of integers.  
30         hprev is Hx1 array of initial hidden state  
31         returns the loss, gradients on model parameters, and last hidden state  
32     """  
33     xs, hs, ys, ps = {}, {}, {}, {}  
34     hs[-1] = np.copy(hprev)  
35     loss = 0  
36     # forward pass  
37     for t in xrange(len(inputs)):  
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation  
39         xs[t][inputs[t]] = 1  
40         hs[t] = np.tanh(np.dot(wh, xs[t]) + np.dot(bh, 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     dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)  
46     dbh, dyb = 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         dbh += dy  
53         dh = np.dot(Why.T, dy) + dhnext # backprop into h  
54         dWxh = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity  
55         dbh += dWxh  
56         dWhh += np.dot(dWxh, xs[t].T)  
57         dWhh += np.dot(dWxh, hs[t-1].T)  
58         dhnext = np.dot(Why.T, dWxh)  
59     for dparam in [dWxh, dWhh, dWhy, dbh, dyb]:  
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
61     return loss, dWxh, dWhh, dWhy, dbh, dyb, hs[len(inputs)-1]
```

min-char-rnn.py gist

```

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

15 # Hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # Model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros(hidden_size, 1) # hidden bias
25 by = np.zeros(vocab_size, 1) # output bias
26
27 by = np.zeros(vocab_size, 1) # output bias

```

```

28 def lossFun(inputs, targets, hprev):
29     """"
30     inputs,targets are both list of integers.
31     hprev is Hx1 array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34
35     xs, hs, ys, ps = {}, {}, {}, {}
36     hs[-1] = np.copy(hprev)
37     loss = 0
38
39     # forward pass
40     for t in xrange(len(inputs)):
41         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
42         xs[t][inputs[t]] = 1
43
44         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
45
46         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
47         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
48
49         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
50
51     # backward pass: compute gradients going backwards
52     dchh, dbh, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
53     dch, dby = np.zeros_like(bh), np.zeros_like(by)
54     dhnext = np.zeros_like(hs[0])
55
56     for t in reversed(xrange(len(inputs)-1)):
57         dy = np.copy(ps[t])
58         dy[targets[t]] -= 1 # backprop into y
59         dhy = np.dot(Why.T, dy) # backprop through Why
60         dbh += np.sum(dy * bh)
61         dchh += np.sum(dy * dhnext)
62         dch += np.sum(dy * xs[t])
63         dby += np.sum(dy * by)
64         dhnext = np.dot(Wxh.T, dy) # backprop through tanh nonlinearity
65
66     for opname in [dchh, dbh, dhy, dch, dby]:
67         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
68
69     return loss, hs[-1]
70
71 def sample(hprev, seed_ix, n):
72     """"
73     sample a sequence of integers from the model
74     h is memory state, seed_ix is seed letter for first time step
75     """
76
77     x = np.zeros((vocab_size, 1))
78     x[seed_ix] = 1
79
80     for t in xrange(n):
81         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
82         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
83         ix = np.argmax(np.random.ranf(vocab_size), p)
84         x[1:] = 0
85
86     return ix
87
88 n, p = 0, 0
89
90 mem, memh, memy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
91 mem, memh, memy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
92 smooth_loss = 0
93 seq_length = 20
94 seq_length += 1
95
96 while True:
97     if p == seq_length:
98         print 'done'
99     else:
100        p += 1
101
102        # Sample from the model, now that we have the context
103        sample_ix = sample(hprev, inputs[p], 200)
104        sample_ix = joinix_to_char[sample_ix] # join ix to char for printing
105        print ' '.join(sample_ix)
106
107    # Forward seq length characters through the net and fetch gradient
108    loss, dchh, dbh, dhy, dch, dby, hprev = lossFun(inputs, targets, hprev)
109    smooth_loss = smooth_loss * 0.999 + loss * 0.001
110
111    if p % 100 == 0: print 'iter %d, loss: %f, smooth loss: %f' % (p, smooth_loss)
112
113    # Compute gradients, backprop through time
114    for opname, mem in zip([dchh, dbh, dhy, dch, dby],
115                           [memh, memh, memy, mem, mem]):
116        np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
117
118    dchh, dbh, dhy, dch, dby = np.zeros_like(dchh), np.zeros_like(dbh),
119                             np.zeros_like(dhy), np.zeros_like(dch), np.zeros_like(dby)
120
121    mem += -learning_rate * dchh / np.sqrt(mem + 1e-8) # adam update
122    memh += -learning_rate * dbh / np.sqrt(memh + 1e-8) # adam update
123    memy += -learning_rate * dhy / np.sqrt(memy + 1e-8) # adam update
124
125    hprev = hprev + dch + dby
126
127    p = seq_length # move data pointer
128
129    n += 1 # iteration counter

```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

```

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
34     xs, hs, ys, ps = {}, {}, {}, {}
35     hs[-1] = np.copy(hprev)
36     loss = 0
37
38     # forward pass
39     for t in xrange(len(inputs)):
40         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
41         xs[t][inputs[t]] = 1
42
43         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
44
45         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
46         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
47
48         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
49
50     # backward pass: compute gradients going backwards
51     dchh, dbh, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
52     dch, dby = np.zeros_like(bh), np.zeros_like(by)
53     dhnext = np.zeros_like(hs[0])
54
55     for t in reversed(xrange(len(inputs)-1)):
56         dy = np.copy(ps[t])
57         dy[targets[t]] -= 1 # backprop into y
58         dhy = np.dot(Why.T, dy) # backprop through Why
59         dbh += np.sum(dy * bh)
60         dchh += np.sum(dy * dhnext)
61         dch += np.sum(dy * xs[t])
62         dby += np.sum(dy * by)
63         dhnext = np.dot(Wxh.T, dy) # backprop through tanh nonlinearity
64
65     for opname in [dchh, dbh, dhy, dch, dby]:
66         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
67
68     return loss, hs[-1]
69
70
71 def sample(hprev, seed_ix, n):
72     """"
73     sample a sequence of integers from the model
74     h is memory state, seed_ix is seed letter for first time step
75     """
76
77     x = np.zeros((vocab_size, 1))
78     x[seed_ix] = 1
79
80     for t in xrange(n):
81         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
82         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
83         ix = np.argmax(np.random.ranf(vocab_size), p)
84         x[1:] = 0
85
86     return ix
87
88 n, p = 0, 0
89
90 mem, memh, memy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
91 mem, memh, memy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
92 smooth_loss = 0
93 seq_length = 20
94 seq_length += 1
95
96 while True:
97     if p == seq_length:
98         print 'done'
99     else:
100        p += 1
101
102        # Sample from the model, now that we have the context
103        sample_ix = sample(hprev, inputs[p], 200)
104        sample_ix = joinix_to_char[sample_ix] # join ix to char for printing
105        print ' '.join(sample_ix)
106
107    # Forward seq length characters through the net and fetch gradient
108    loss, dchh, dbh, dhy, dch, dby, hprev = lossFun(inputs, targets, hprev)
109    smooth_loss = smooth_loss * 0.999 + loss * 0.001
110
111    if p % 100 == 0: print 'iter %d, loss: %f, smooth loss: %f' % (p, smooth_loss)
112
113    # Compute gradients, backprop through time
114    for opname, mem in zip([dchh, dbh, dhy, dch, dby],
115                           [memh, memh, memy, mem, mem]):
116        np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
117
118    dchh, dbh, dhy, dch, dby = np.zeros_like(dchh), np.zeros_like(dbh),
119                             np.zeros_like(dhy), np.zeros_like(dch), np.zeros_like(dby)
120
121    mem += -learning_rate * dchh / np.sqrt(mem + 1e-8) # adam update
122    memh += -learning_rate * dbh / np.sqrt(memh + 1e-8) # adam update
123    memy += -learning_rate * dhy / np.sqrt(memy + 1e-8) # adam update
124
125    hprev = hprev + dch + dby
126
127    p = seq_length # move data pointer
128
129    n += 1 # iteration counter

```

min-char-rnn.py gist

```

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

15 # Hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19 model_params = {}

20 wh = np.random.rand(hidden_size, size=vocab_size)*0.01 # input to hidden
21 bh = np.random.rand(hidden_size, size=hidden_size)*0.01 # hidden to hidden
22 why = np.random.rand(vocab_size, size=hidden_size)*0.01 # hidden to output
23 bh_0 = np.zeros(hidden_size, size=1) # hidden bias
24 by = np.zeros(vocab_size, size=1) # output bias

25 def lossFun(inputs, targets, hprev):
26     """ inputs,targets are both lists of integers.
27     hprev is Hx1 array of initial hidden state
28     returns the loss, gradients on model parameters, and last hidden state
29     """
30
31     xs, hs, ys, ps = [], [], [], []
32     h0 = np.copy(hprev)
33     loss = 0
34     for t in xrange(len(inputs)):
35         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
36         x[inputs[t]] = 1
37         h0 = np.tanh(np.dot(wh, h0) + np.dot(bh, h0[-1]) + bh)
38         ys[t] = np.exp(np.dot(why, h0)) / np.sum(np.exp(ys[t])) # next chars
39         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
40         loss += -np.log(ps[t][targets[t]] if targets[t] > 0 else ps[t])
41
42         dnh, dwh, dbh = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(bh)
43         dby = np.zeros_like(by)
44         dhnext = np.zeros_like(h0)
45         for i in xrange(vocab_size):
46             dy = np.copy(ps[t])
47             dy[i] -= 1 # backprop into y
48             dwhy += np.dot(dy, hs[t].T)
49             dby += dy
50             dh = np.dot(why.T, dy) + dhnext # backprop into h
51             ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
52             dbh += ddraw
53             dwh += np.dot(ddraw, xs[t].T)
54             dnh += np.dot(dh, hs[t-1].T)
55             dhnext = np.dot(wh.T, ddraw)
56
57     for dparam in [dwh, dwhh, dwhy, dbh, dby]:
58         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
59
60     return loss, dwh, dwhh, dwhy, dbh, dby, hs[-len(inputs):-1]
61
62 def sample(prev_h, ix):
63     """ sample a sequence of integers from the model
64     h is memory state, seed_ix is seed letter for first time step
65     """
66
67     x = np.zeros((vocab_size, 1))
68     x[seed_ix] = 1
69
70     for t in xrange(n):
71         h = np.tanh(np.dot(wh, x) + np.dot(bh, h) + bh)
72         p = np.exp(h) / np.sum(np.exp(h))
73         ix = np.random.choice(range(vocab_size), p=p.ravel())
74         x[1:] = 0
75         x[ix] = 1
76
77     return ix
78
79 n, p = 0
80
81 mem, m0, why = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(why)
82 mem0, m00, bh0 = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(bh)
83 smooth_loss = 0
84 seqLength = len(data)
85
86 while True:
87     if p == seqLength-1 or n == 0:
88         hprev = np.zeros((hidden_size, size))
89         reset_rnn()
90
91     inputs = [char_to_ix[ch] for ch in data[p:seqLength]]
92     targets = [char_to_ix[ch] for ch in data[p+1:seqLength+1]]
93
94     a = np.zeros((size, 1))
95     a[0] = 1
96
97     sample_ix = sample(hprev, inputs[0], 200)
98     print " ".join(ix_to_char[i] for i in sample_ix)
99     print " "
100
101     # Forward pass: Compute activations through the net and fetch gradient
102     loss, dwh, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
103     smooth_loss = smooth_loss * 0.999 + loss * 0.001
104     if n % 100 == 0: print "iter %d, loss: %f, smooth loss: %f" % (n, loss, smooth_loss)
105
106     # Backward pass: Compute gradients
107     for param, mem in zip([wh, whh, why, bh, bh0],
108                           [dwh, dwhh, dwhy, dbh, dbh0]):
109         np.clip(param, -5, 5, out=mem)
110         np.dot(mem, dparam, out=mem)
111         mem *= learning_rate
112         param += np.sqrt(mem + 1e-8) * dparam # adam update
113
114     n += 1 # iteration counter

```

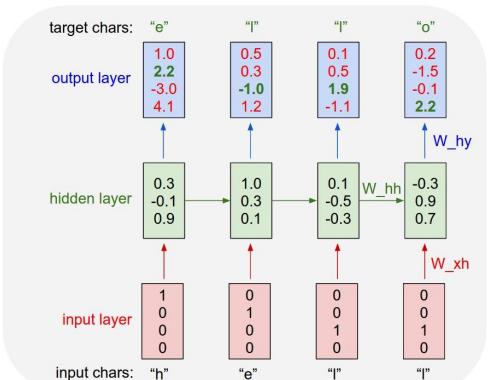


```

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

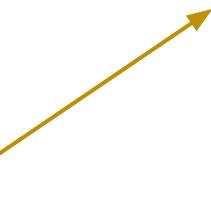
```

recall:

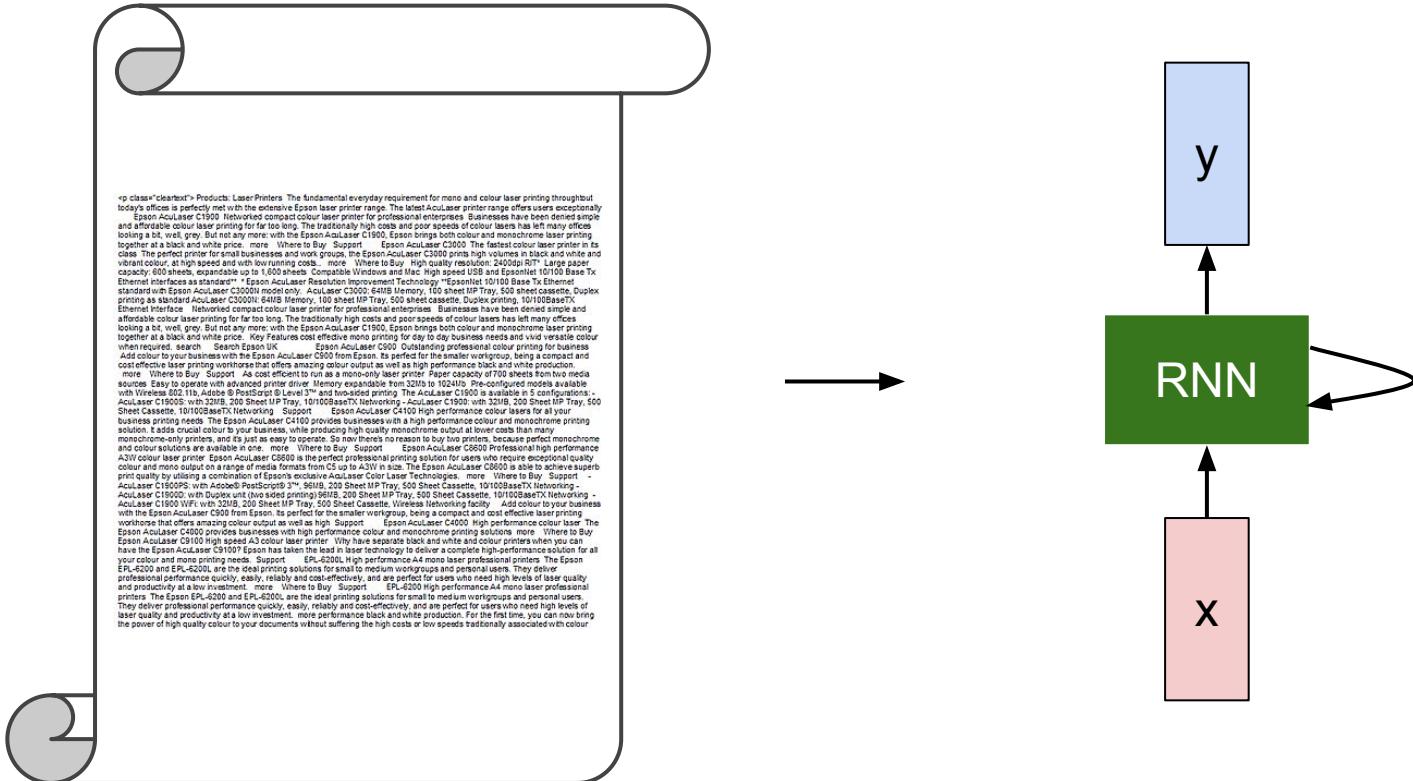


min-char-rnn.py gist

```
1 /**
2 * Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 * BSD License
4 */
5
6 import numpy as np
7
8 # Data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print('data has %d characters, %d unique.' % (data_size, vocab_size))
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} 
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def init_randomhidden(state_size, vocab_size):
24     wh = np.random.rand(hidden_size, state_size)*0.01 # input to hidden
25     bh = np.random.rand(hidden_size, 1) # hidden bias
26     why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
27     by = np.zeros(vocab_size, 1) # hidden bias
28     bb = np.zeros(vocab_size, 1) # output bias
29
30 def lossfun(inputs, targets, hprev):
31     """ Inputs,targets are both lists of integers.
32     hprev is RNN array of initial hidden state
33     returns the loss, gradients on model parameters, and last hidden state
34 """
35     xs, ys, ps = [], [], []
36     hprev = np.copy(hprev)
37     loss = 0
38
39     for t in xrange(len(inputs)):
40         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
41         x[inputs[t], 0] = 1
42
43         hprev = np.tanh(np.dot(wih, x) + np.dot(whh, hprev) + bh) # hidden state
44
45         y = np.exp(why.T) / np.sum(np.exp(why.T)) # probabilities for next chars
46         ps[t] = y
47
48         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
49
50         # backward pass: compute gradients up to backprop into hprev
51         dprev, dhprev, dwih, dwhh, dbh, dwhy, dy, dbb = 0
52
53         dy = np.copy(ps[t])
54         dy[targets[t]] -= 1 # backprop into y
55         dbb += dy
56         dprev = np.dot(dy, wh.T) # backprop into hprev
57         dhprev = np.dot(dprev, wih.T) # backprop through tanh nonlinearity
58         dwih += np.dot(dhprev, x.T)
59         dwhh += np.dot(dhprev, dhprev.T)
60         dbh += np.sum(dhprev, 1, 1)
61
62         for opname in [dprev, dhprev, dwih, dwhh, dbb]:
63             if np.isnan(opname).any():
64                 raise ValueError("%s contains nan" % opname)
65
66         for opname in [dwih, dwhh, dbh, dwhy]:
67             np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
68
69         if np.isnan(dwih).any() or np.isnan(dwhh).any() or np.isnan(dbh).any() or np.isnan(dwhy).any():
70             raise ValueError("NaNs found in dwih, dwhh, dbh, dwhy, dbb")
71
72     for i in range(len(inputs)-1):
73         loss += smooth_loss(dwih[i], dwih[i+1])
74
75     return loss
76
77
78 # Sample sequence from the model
79 def sample(h, seed_ix, n):
80     """ sample a sequence of integers from the model
81     h is memory state, seed_ix is seed letter for first time step
82     n is length of sequence to sample
83     """
84
85     x = np.zeros((vocab_size, 1))
86     x[seed_ix] = 1
87
88    for t in xrange(n):
89        h = np.tanh(np.dot(wih, x) + np.dot(whh, h) + bh)
90        y = np.dot(why, h) + by
91        p = np.exp(y) / np.sum(np.exp(y))
92        ix = np.random.choice(range(vocab_size), p=p.ravel())
93        x = np.zeros((vocab_size, 1))
94        x[ix] = 1
95
96    return ixes
97
98
99 # Train the model, now!
100 n = 100
101 sample_ix = sample(hprev, inputs[0], 200)
102 print 'Initial sequence: ', ''.join(ix_to_char[i] for i in sample_ix)
103 print 'Sampling...'
104
105 # Forward seq length characters through the net and fetch gradient
106 loss, dhprev, dwhh, dwih, dbh, dby, dbb, dwhy = lossfun(inputs, targets, hprev)
107 smooth_loss = smooth_loss * 0.999 + loss * 0.001
108
109 if n < 100: n = 100 # Print first 100 chars
110 print 'Initial loss: %.3f' % smooth_loss
111
112 for para, opname in zip([wh, why, by, bh],
113                         [dwh, dwhy, dy, dbb]):
114     print '%s = %s' % (para, opname)
115
116     if np.isnan(opname).any():
117         raise ValueError("%s contains nan" % opname)
118
119     opname *= -learning_rate # update
120     opname /= np.sqrt(dwih + 1e-8) # adagrad update
121
122     para += opname
123
124 p = np.zeros((vocab_size, 1))
125
126 n += 1 # iteration counter
```



```
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(wih, 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
    return ixes
```



Sonnet 116 – Let me not ...

by William Shakespeare

Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove:
O no! it is an ever-fixed mark
That looks on tempests and is never shaken;
It is the star to every wandering bark,
Whose worth's unknown, although his height be taken.
Love's not Time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
If this be error and upon me proved,
I never writ, nor no man ever loved.

at first:

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

↓ train more

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

↓ train more

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

↓ train more

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

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

VIOLA:

Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:

O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

open source textbook on algebraic geometry

The Stacks Project

home about tags explained tag lookup browse search bibliography recent comments blog add slogans

Browse chapters

Part	Chapter	online	TeX source	view pdf
Preliminaries	1. Introduction	online	tex	pdf
	2. Conventions	online	tex	pdf
	3. Set Theory	online	tex	pdf
	4. Categories	online	tex	pdf
	5. Topology	online	tex	pdf
	6. Sheaves on Spaces	online	tex	pdf
	7. Sites and Sheaves	online	tex	pdf
	8. Stacks	online	tex	pdf
	9. Fields	online	tex	pdf
	10. Commutative Algebra	online	tex	pdf

Parts

- [Preliminaries](#)
- [Schemes](#)
- [Topics in Scheme Theory](#)
- [Algebraic Spaces](#)
- [Topics in Geometry](#)
- [Deformation Theory](#)
- [Algebraic Stacks](#)
- [Miscellany](#)

Statistics

The Stacks project now consists of

- 455910 lines of code
- 14221 tags (56 inactive tags)
- 2366 sections

Latex source

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

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

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

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

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

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

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

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

is a unique morphism of algebraic stacks. Note that

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

and

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

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

Proof. See discussion of sheaves of sets. \square

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. Omitted. □

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

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

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

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

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

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

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

Proof. See Spaces, Lemma ??.

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

The following to the construction of the lemma follows.

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

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

be a morphism of algebraic spaces over S and Y .

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

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

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

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

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

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

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

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

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

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

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\bar{x}} \xrightarrow{-1} (\mathcal{O}_{X_{\text{étale}}}) \longrightarrow \mathcal{O}_{X_{\bar{\ell}}}^{-1} \mathcal{O}_{X_{\lambda}}(\mathcal{O}_{X_{\eta}}^{\bar{v}})$$

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

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

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

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

 torvalds / linux Watch · 3,711 Star · 23,054 Fork · 9,141

Linux kernel source tree

520,037 commits

1 branch

420 releases

5,039 contributors

branch: master · [linux](#) / +

Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux ...

 torvalds authored 9 hours agolatest commit 4b1786927d  Documentation

Merge git://git.kernel.org/pub/scm/linux/kernel/git/nab/target-pending

6 days ago

 arch

Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/l...

a day ago

 block

block: discard bdi_unregister() in favour of bdi_destroy()

9 days ago

 crypto

Merge git://git.kernel.org/pub/scm/linux/kernel/git/herbert/crypto-2.6

10 days ago

 drivers

Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux

9 hours ago

 firmware

firmware/hex2fw.c: restore missing default in switch statement

2 months ago

 fs

vfs: read file_handle only once in handle_to_path

4 days ago

 include

Merge branch 'perl-urgent-for-linus' of git://git.kernel.org/pub/scm/...

a day ago

 init

init: fix regression by supporting devices with major:minor:offset fo...

a month ago

 io

allowance: fix race between writeback and multi-thread traversal when doing unlinked list removal

a month ago

 Code Pull requests
74 Pulse Graphs

HTTPS clone URL

<https://github.com/torvalds/linux> You can clone with [HTTPS](#), [SSH](#), or [Subversion](#).  Clone in Desktop Download ZIP

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << i))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 0x000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

Generated C code

```
/*
 * Copyright (c) 2006-2010, Intel Mobile Communications. All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/seteew.h>
#include <asm/pgproto.h>
```

```

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/seteew.h>
#include <asm/pgproto.h>

#define REG_PG      vesa_slot_addr_pack
#define PFM_NOCOMP  AFSR(0, load)
#define STACK_DDR(type)      (func)

#define SWAP_ALLOCATE(nr)      (e)
#define emulate_sigs()  arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %esp, %0, %3" : : "r" (0)); \
    if (__type & DO_READ)

static void stat_PC_SEC __read_mostly offsetof(struct seq_argsqueue, \
    pC>[1]);

static void
os_prefix(unsigned long sys)
{
#endif CONFIG_PREEMPT
    PUT_PARAM_RAID(2, sel) = get_state_state();
    set_pid_sum((unsigned long)state, current_state_str(),
                (unsigned long)-1->lr_full, low;
}

```

Searching for interpretable cells

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
    */
}
```

[Visualizing and Understanding Recurrent Networks, Andrej Karpathy*, Justin Johnson*, Li Fei-Fei]

Searching for interpretable cells

"You mean to imply that I have nothing to eat out of.... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."

quote detection cell

Searching for interpretable cells

Cell sensitive to position in line:

The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.

line length tracking cell

Searching for interpretable cells

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
    siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (! (current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
            collect_signal(sig, pending, info);
        }
    }
    return sig;
}
```

if statement cell

Searching for interpretable cells

```
/* Duplicate LSM field information. The lsm_rule is opaque, so
 * re-initialized. */
static inline int audit_dupe_lsm_field(struct audit_field *df,
                                       struct audit_field *sf)
{
    int ret = 0;
    char *lsm_str;
    /* our own copy of lsm_str */
    lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
    if (unlikely(!lsm_str))
        return -ENOMEM;
    df->lsm_str = lsm_str;
    /* our own (refreshed) copy of lsm_rule */
    ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
                                   (void **) &df->lsm_rule);
    /* Keep currently invalid fields around in case they
     * become valid after a policy reload. */
    if (ret == -EINVAL) {
        pr_warn("audit rule for LSM \\'%s\\' is invalid\n",
               df->lsm_str);
        ret = 0;
    }
    return ret;
}
```

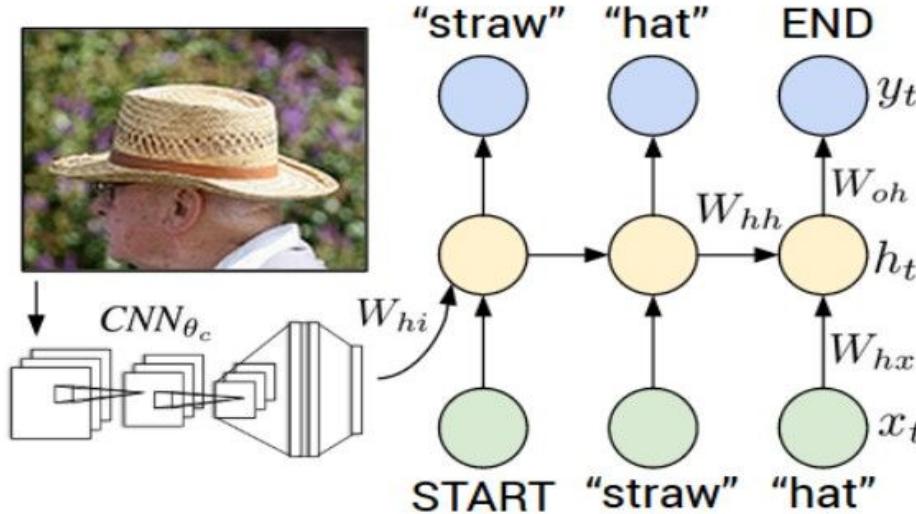
quote/comment cell

Searching for interpretable cells

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

code depth cell

Image Captioning



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

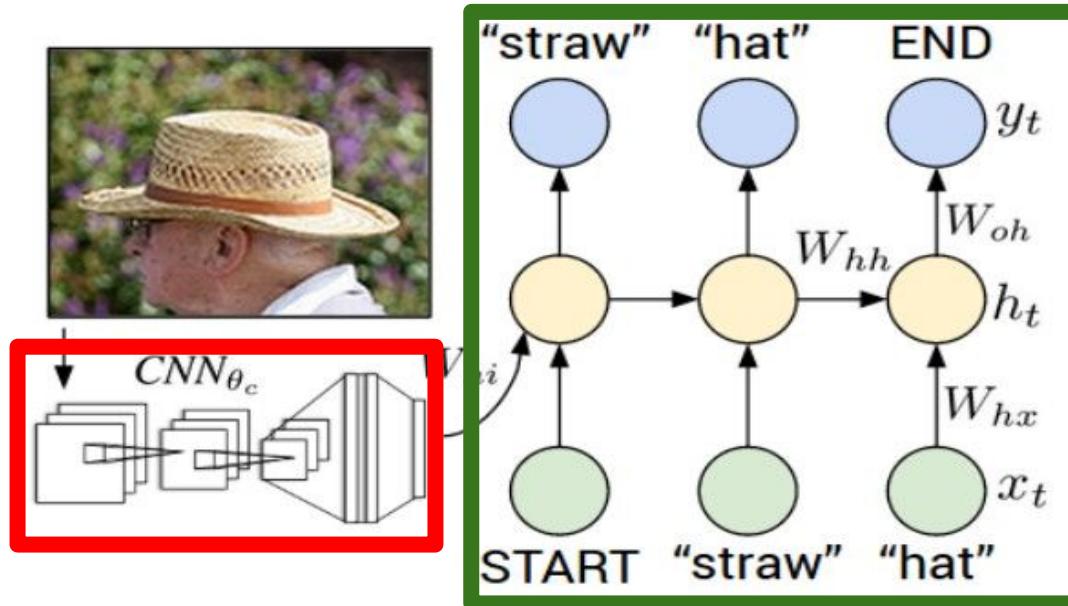
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

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

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

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

Recurrent Neural Network



Convolutional Neural Network

test image



image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

X

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

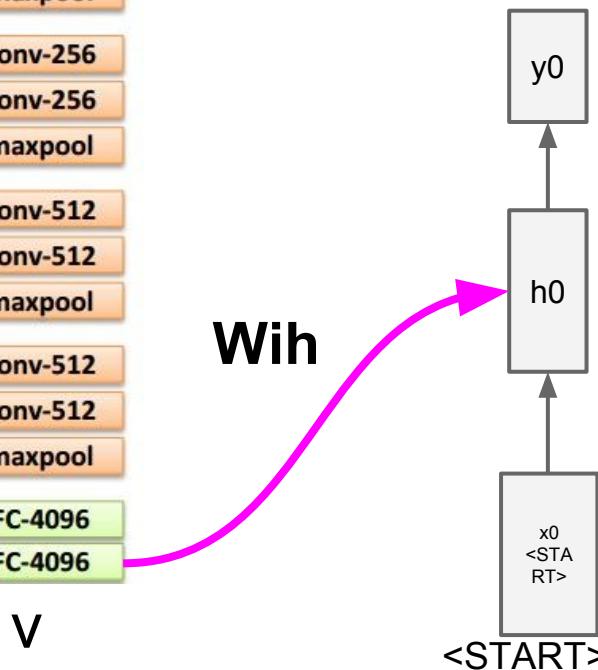
FC-4096



<START>



test image



before:

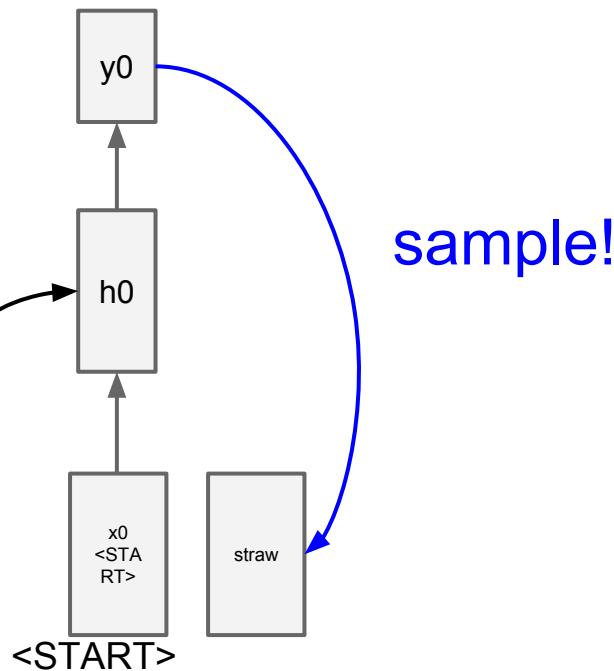
$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

now:

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



test image



image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

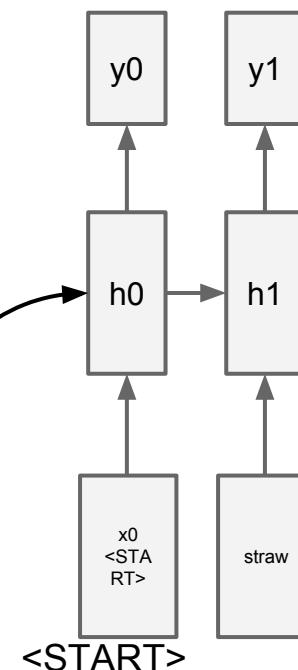
maxpool

FC-4096

FC-4096

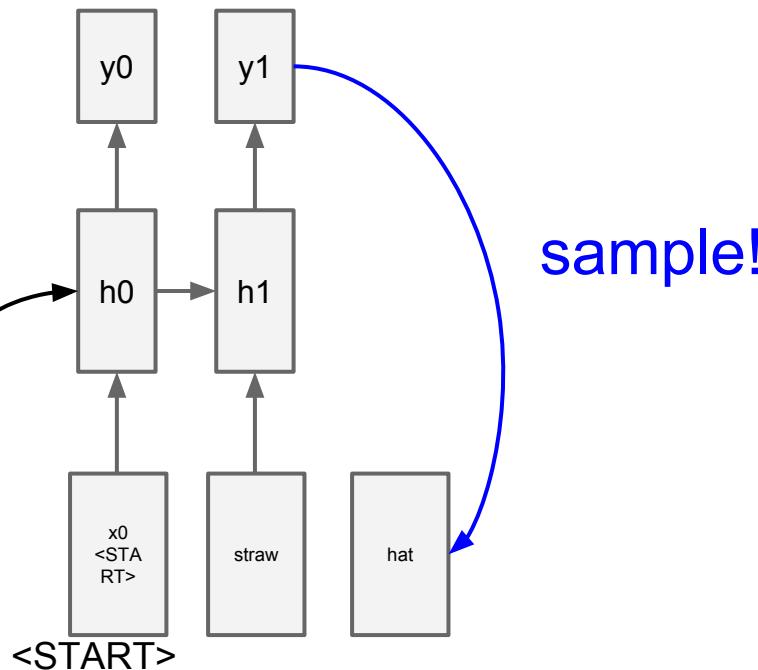


test image





test image



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

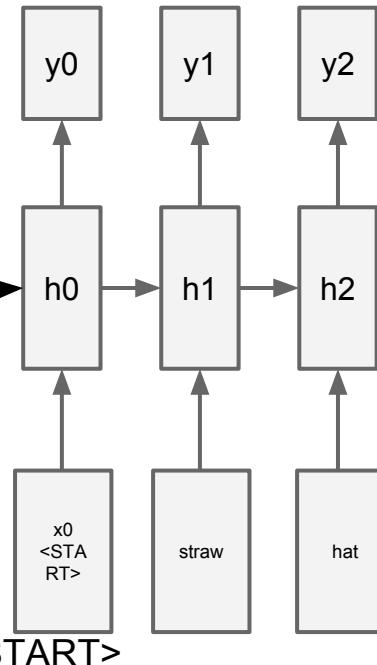
conv-512

conv-512

maxpool

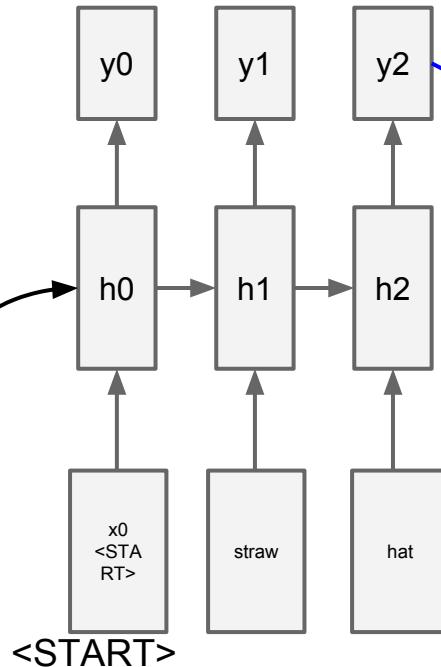
FC-4096

FC-4096





test image



sample
<END> token
=> finish.

Image Sentence Datasets

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



Microsoft COCO
[Tsung-Yi Lin et al. 2014]
mscoco.org

currently:
~120K images
~5 sentences each



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



"a cat is sitting on a couch with a remote control."



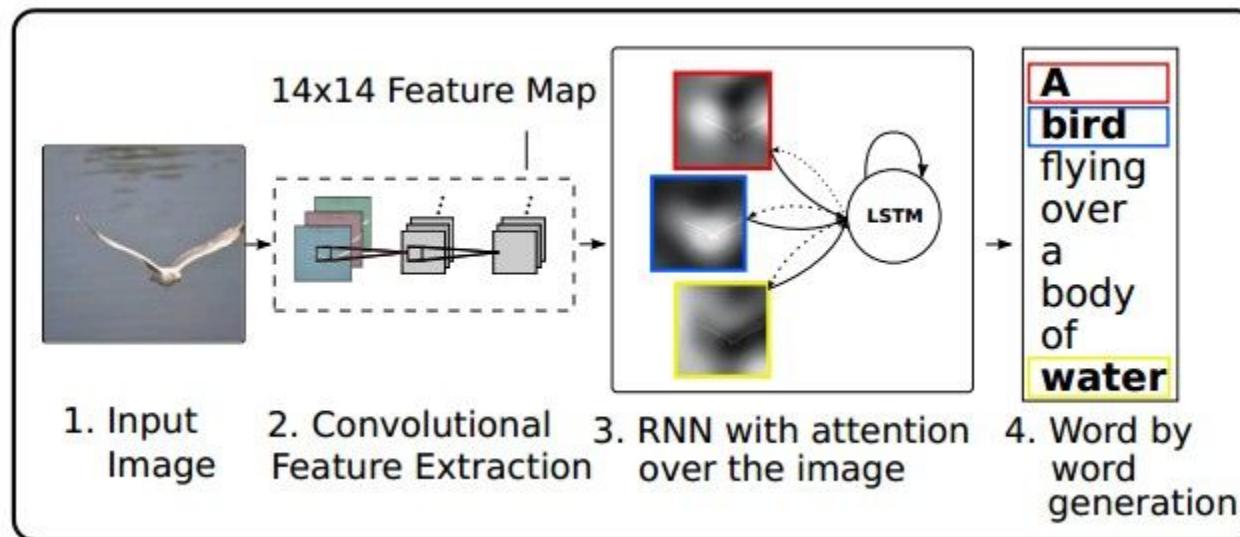
"a woman holding a teddy bear in front of a mirror."



"a horse is standing in the middle of a road."

Preview of fancier architectures

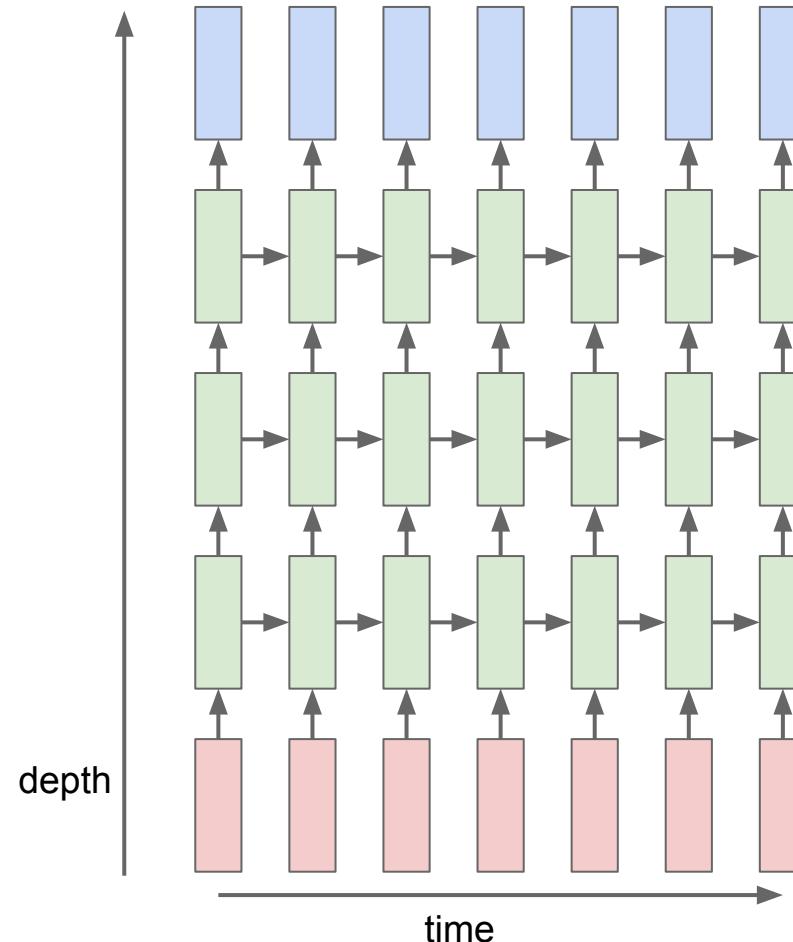
RNN attends spatially to different parts of images while generating each word of the sentence:



RNN:

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$h \in \mathbb{R}^n$. W^l [n × 2n]



RNN:

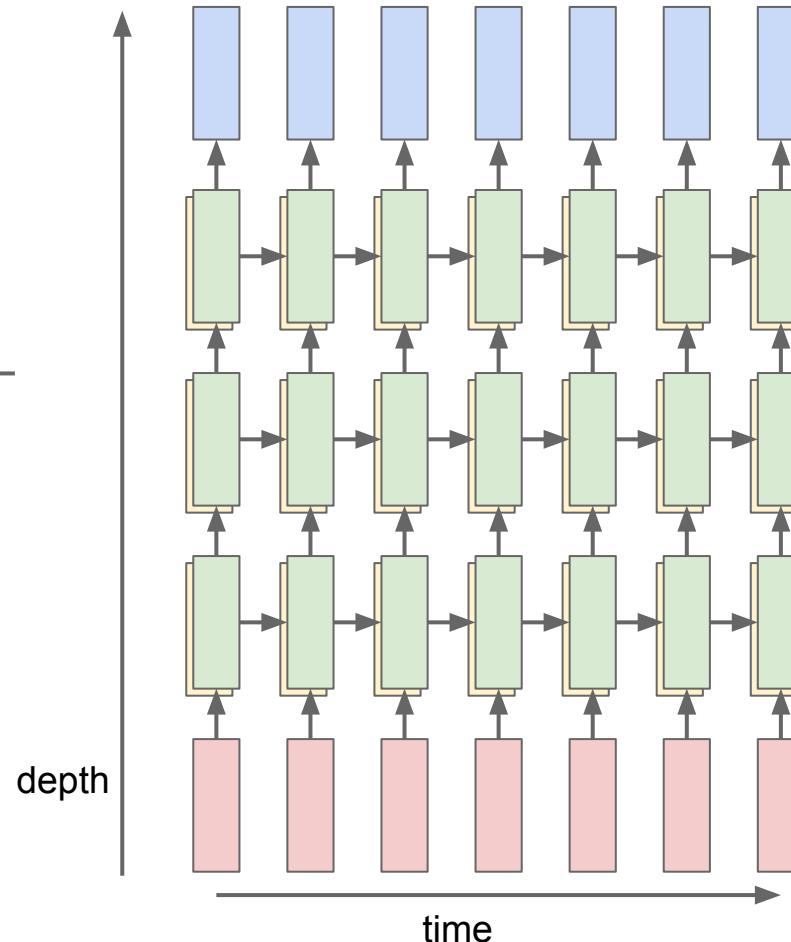
$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$h \in \mathbb{R}^n$ $W^l [n \times 2n]$

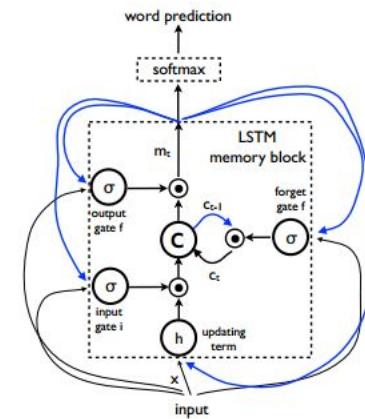
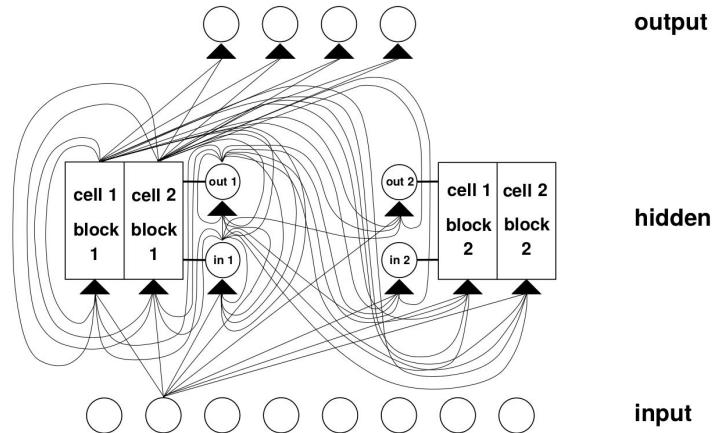
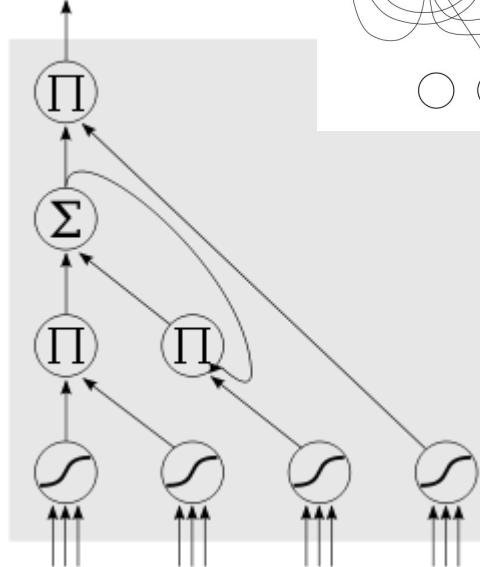
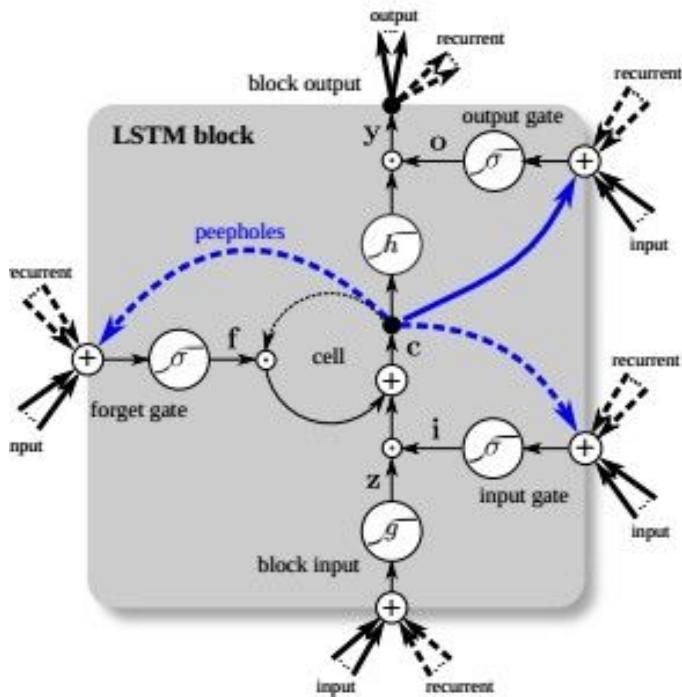
LSTM:

$$W^l [4n \times 2n]$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

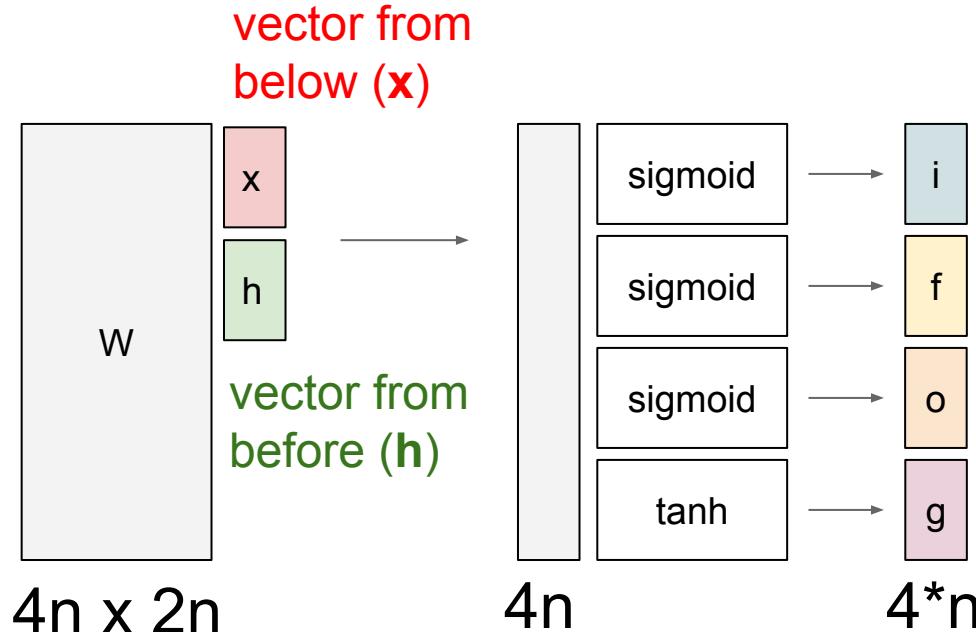


LSTM



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

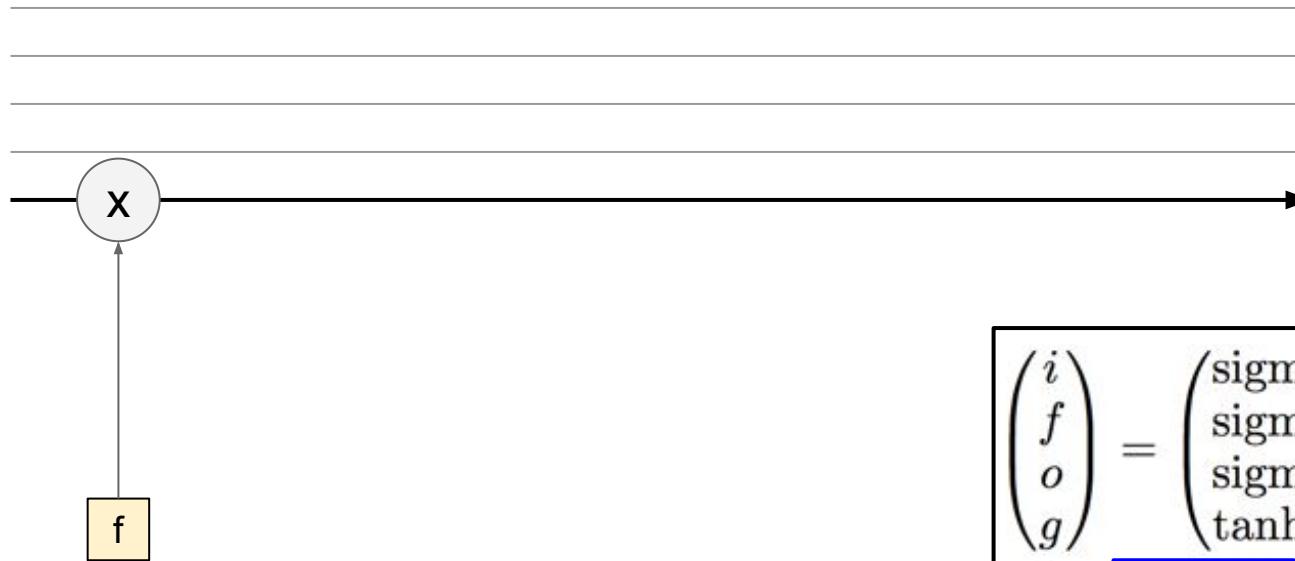


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_t^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state **c**

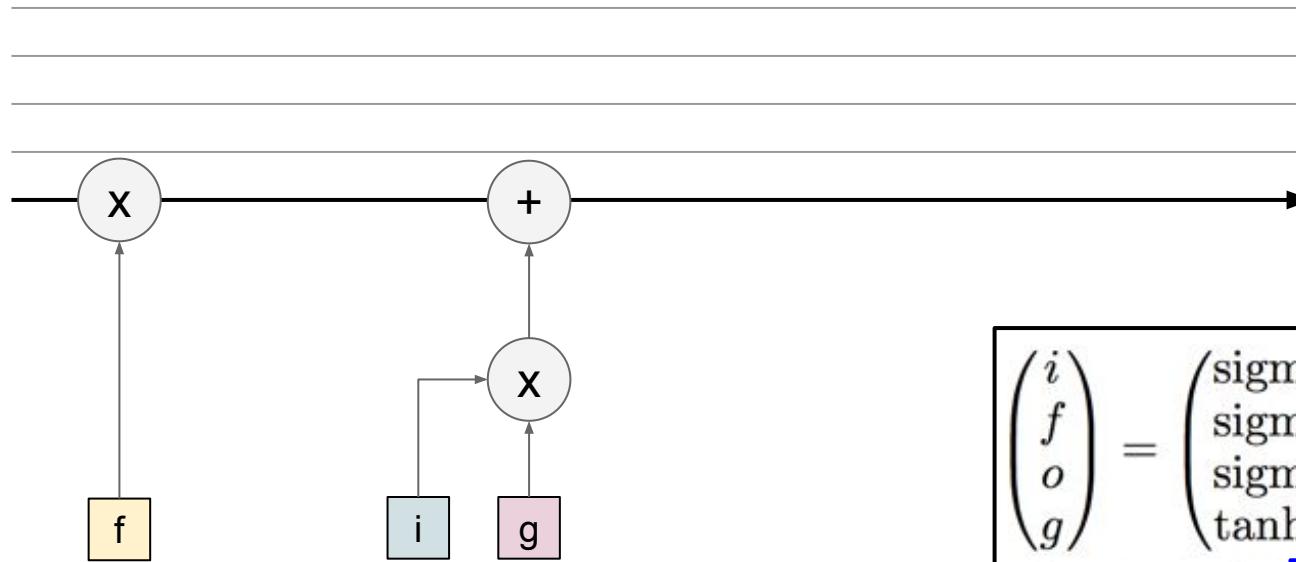


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c

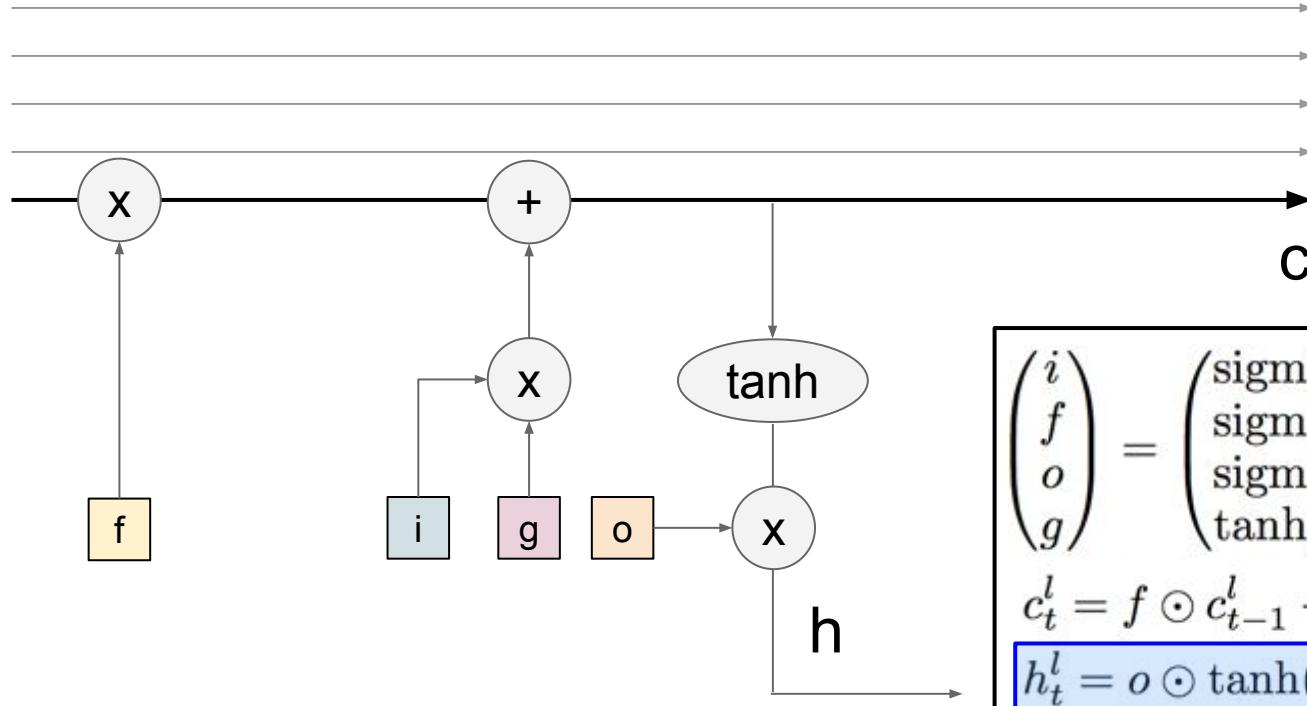


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c

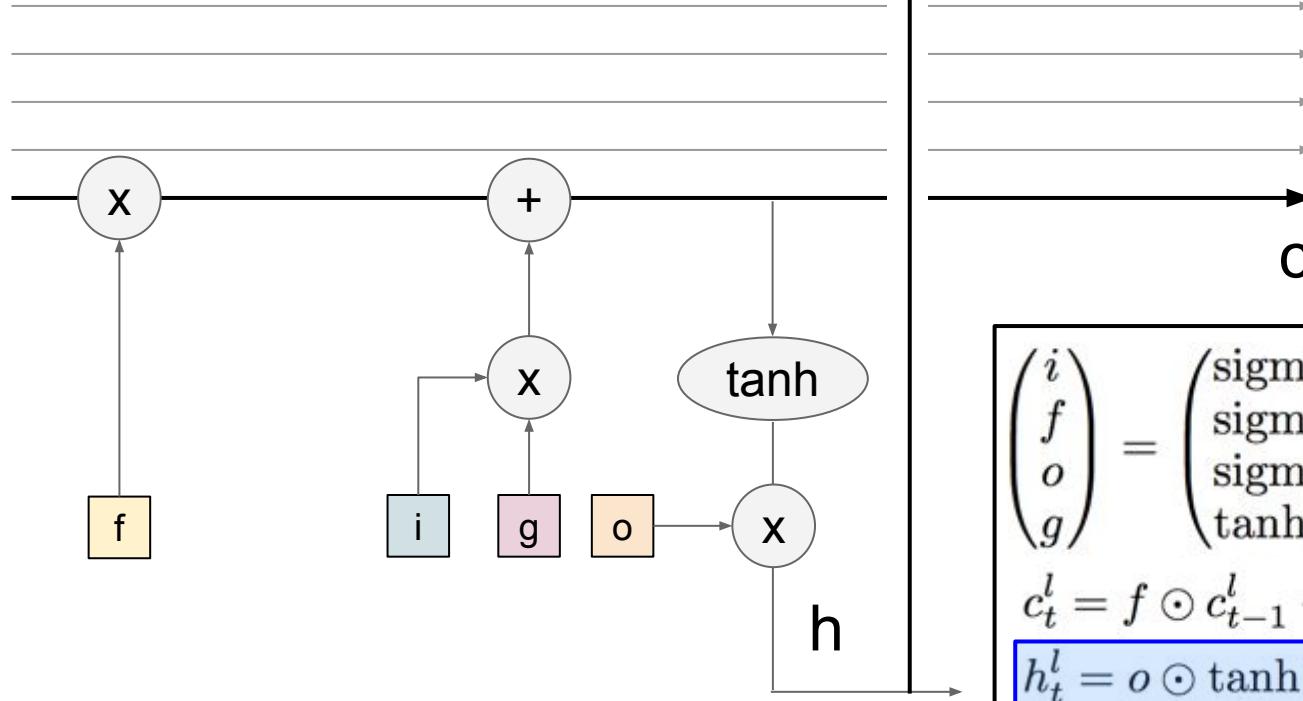


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c

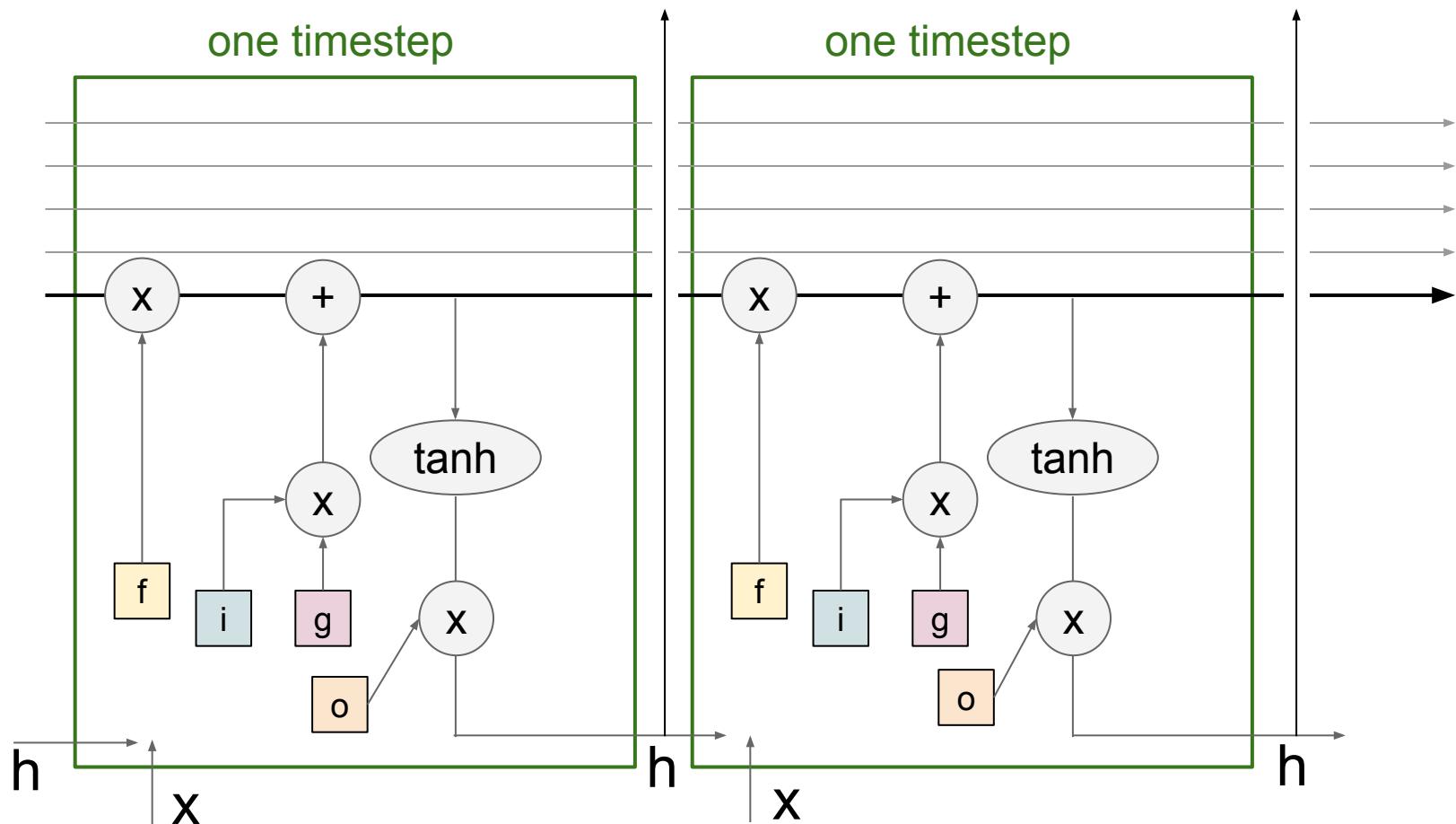


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

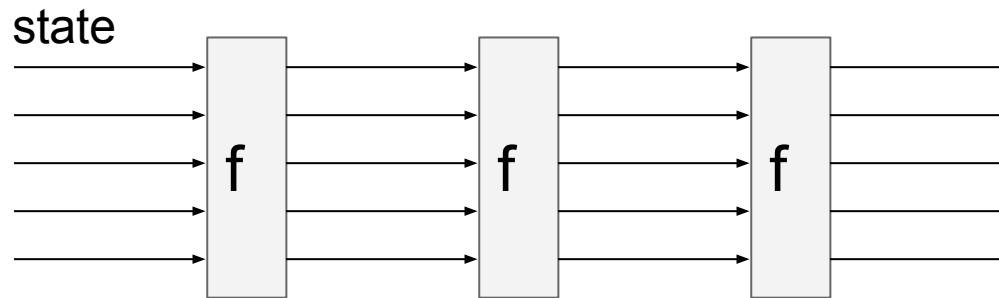
LSTM

one timestep

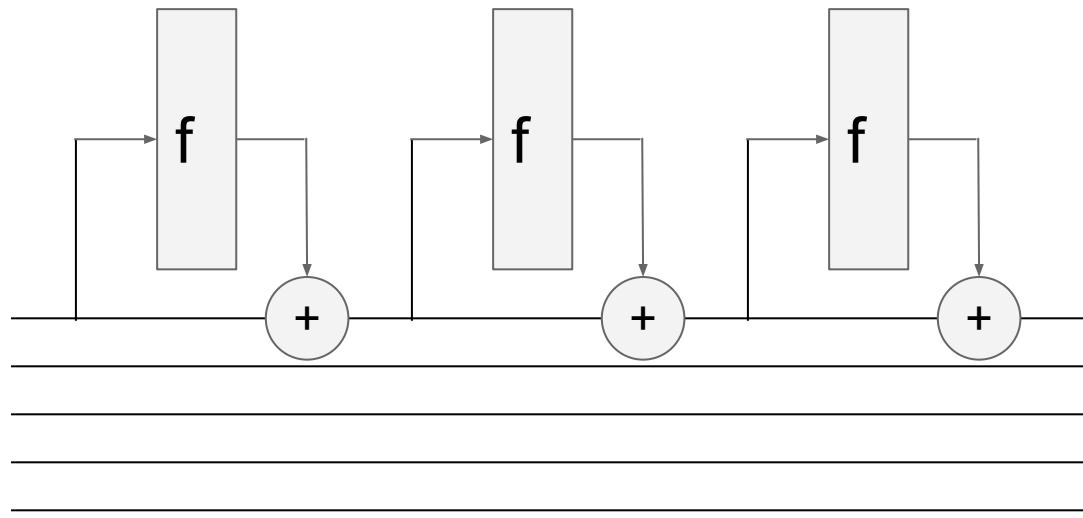
cell
state c



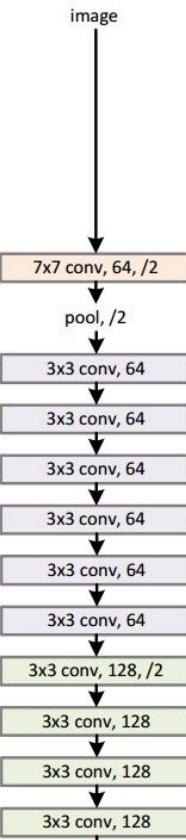
RNN



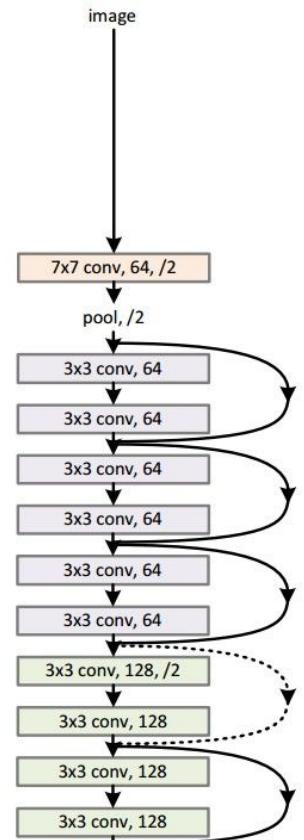
LSTM (ignoring forget gates)



34-layer plain

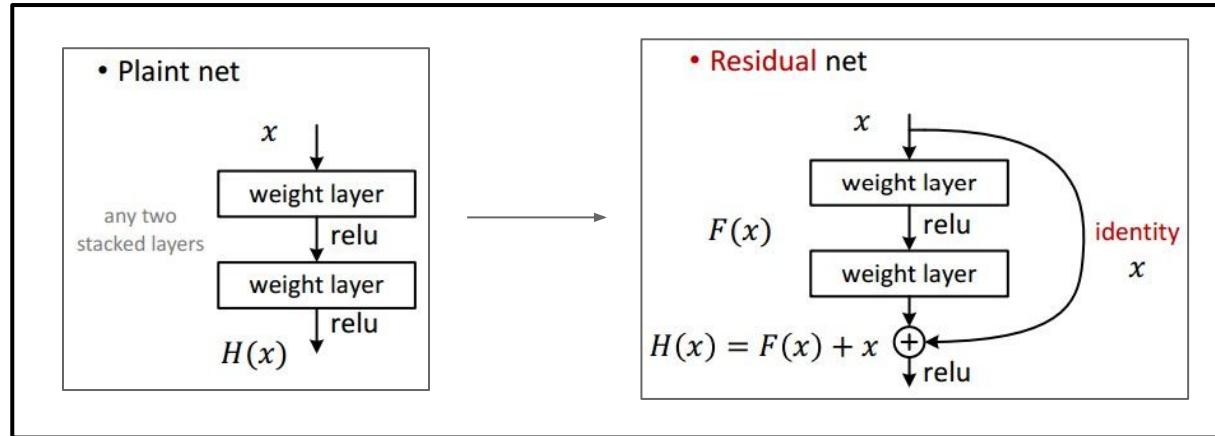


34-layer residual



Recall: “PlainNets” vs. ResNets

ResNet is to PlainNet what LSTM is to RNN, kind of.



Understanding gradient flow dynamics

Cute backprop signal video: <http://imgur.com/gallery/vaNahKE>

```
H = 5      # dimensionality of hidden state
T = 50     # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

Understanding gradient flow dynamics

```
H = 5      # dimensionality of hidden state
T = 50     # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

if the largest eigenvalue is > 1 , gradient will explode
if the largest eigenvalue is < 1 , gradient will vanish

[On the difficulty of training Recurrent Neural Networks, Pascanu et al., 2013]

Understanding gradient flow dynamics

```
H = 5      # dimensionality of hidden state
T = 50     # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

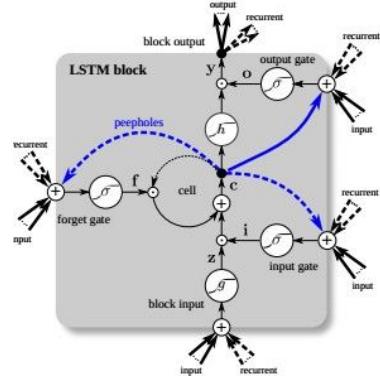
if the largest eigenvalue is > 1 , gradient will explode
if the largest eigenvalue is < 1 , gradient will vanish

can control exploding with gradient clipping
can control vanishing with LSTM

[On the difficulty of training Recurrent Neural Networks, Pascanu et al., 2013]

LSTM variants and friends

[*An Empirical Exploration of Recurrent Network Architectures*, Jozefowicz et al., 2015]



[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$\begin{aligned} r_t &= \text{sigm}(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \text{sigm}(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \end{aligned}$$

MUT1:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + \tanh(x_t) + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}h_t + b_z) \\ r &= \text{sigm}(x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}\tanh(h_t) + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.