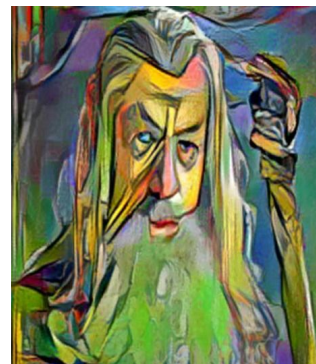
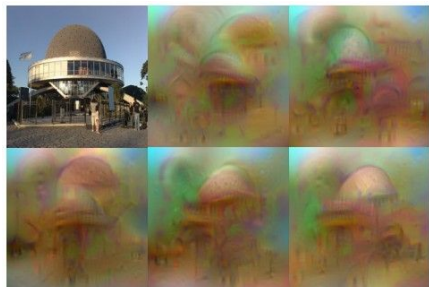
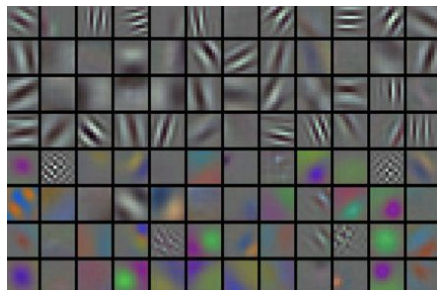


# Lecture 10:

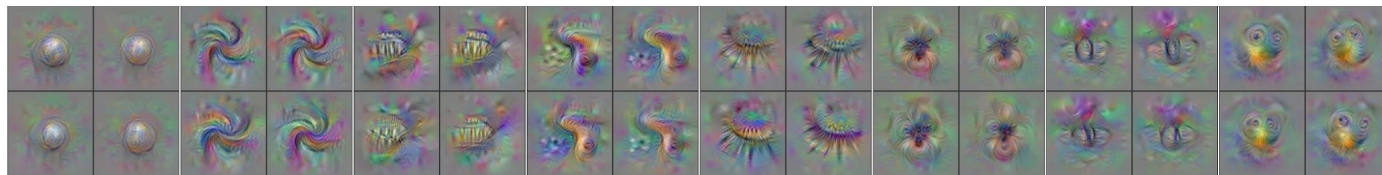
## Recurrent Neural Networks

# Administrative

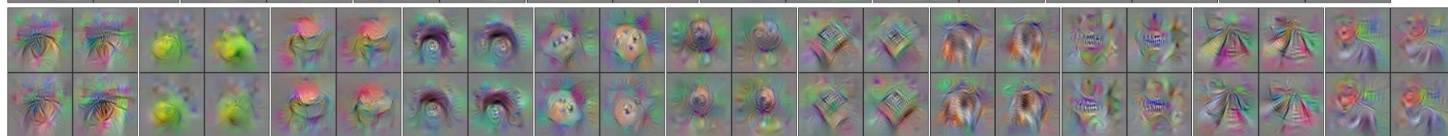
- Midterm this Wednesday! woohoo!
- A3 will be out ~Wednesday



Layer 4



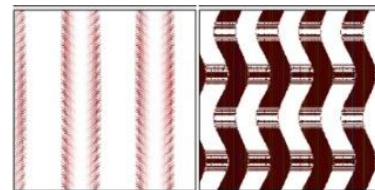
Layer 3



Layer 2

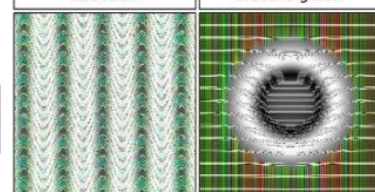


Layer 1



baseball

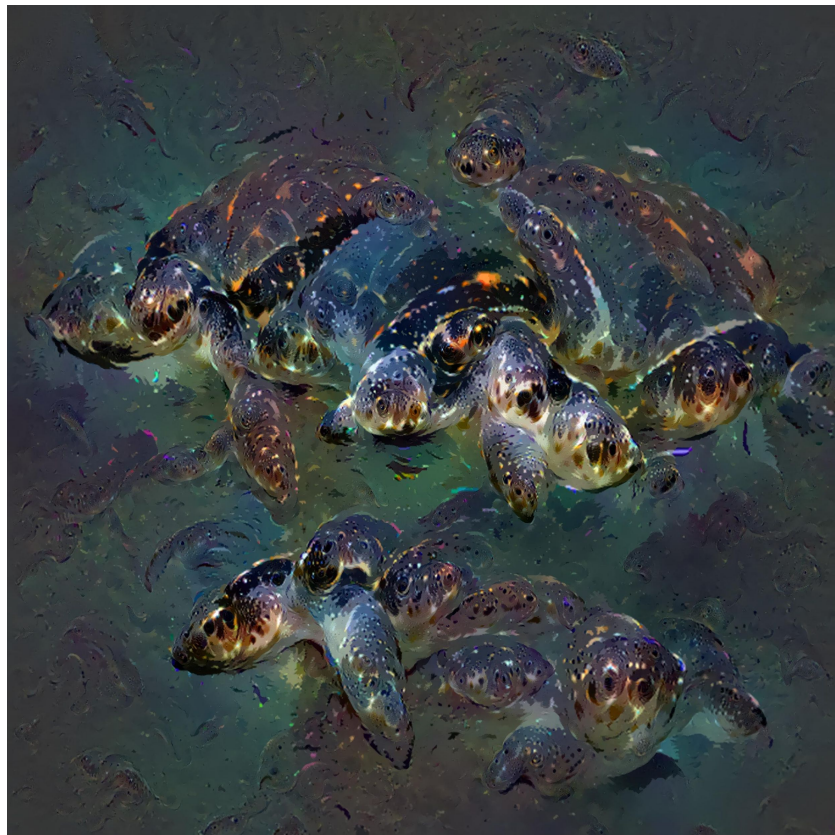
electric guitar

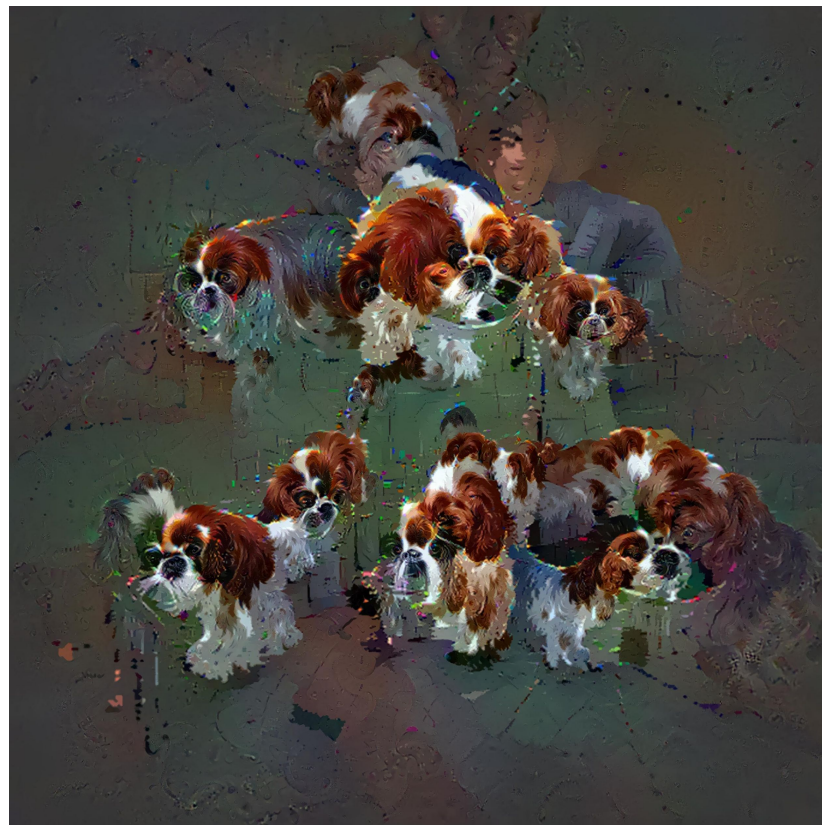


peacock

African grey



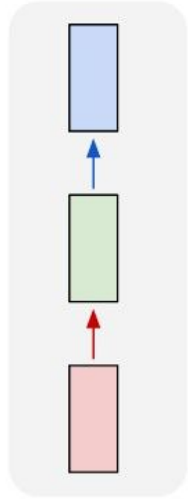




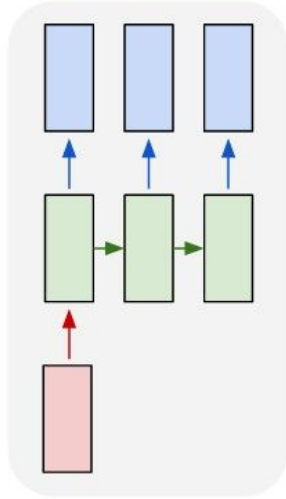


# Recurrent Networks offer a lot of flexibility:

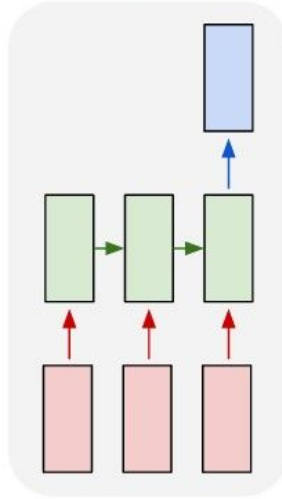
one to one



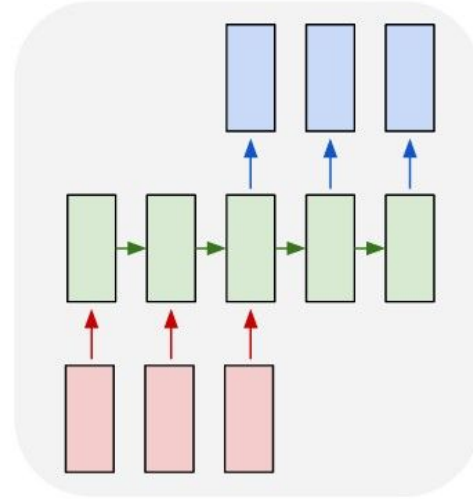
one to many



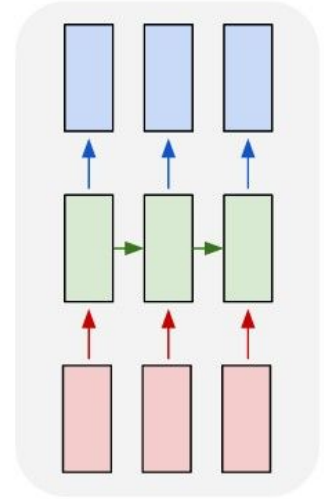
many to one



many to many



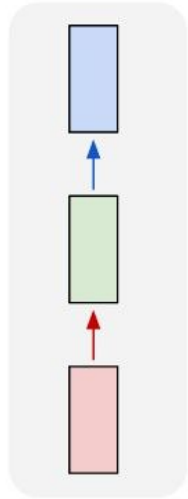
many to many



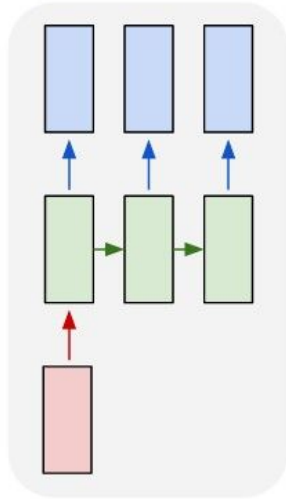
↙ **Vanilla Neural Networks**

# Recurrent Networks offer a lot of flexibility:

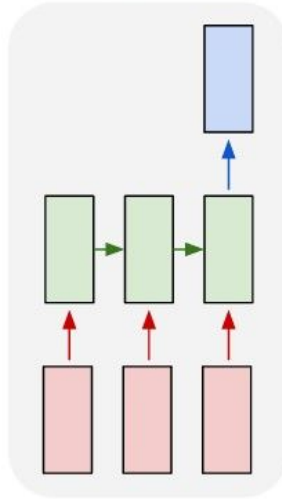
one to one



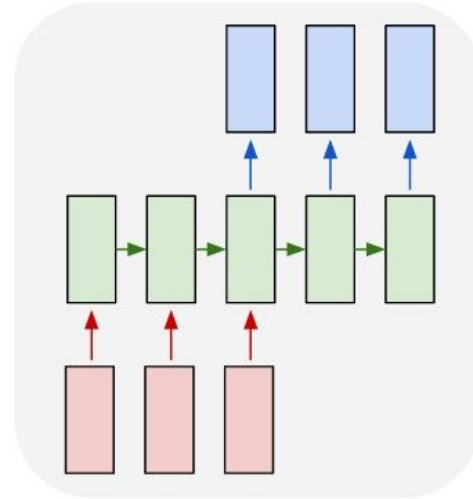
one to many



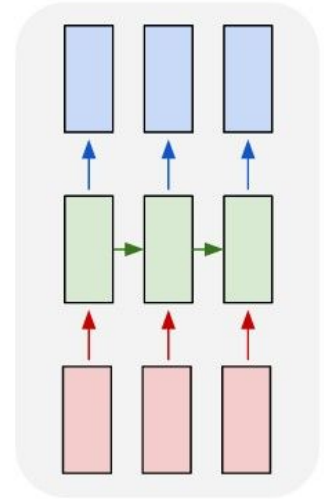
many to one



many to many



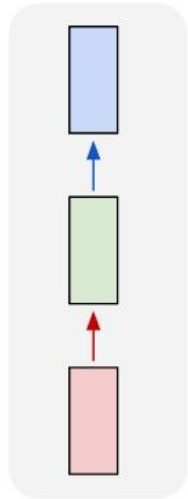
many to many



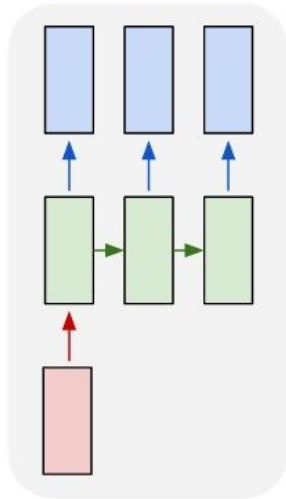
e.g. **Image Captioning**  
image -> sequence of words

# Recurrent Networks offer a lot of flexibility:

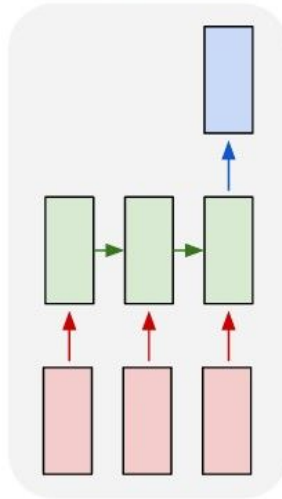
one to one



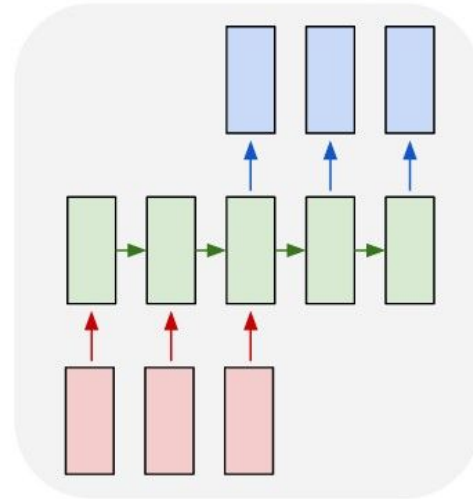
one to many



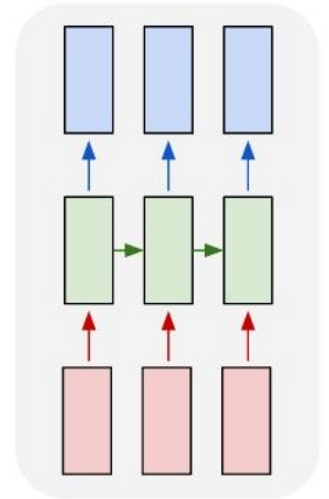
many to one



many to many



many to many

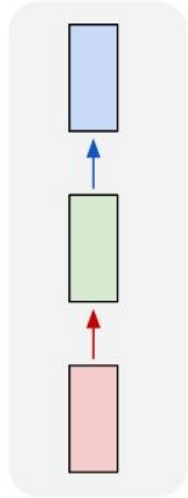


↖ e.g. **Sentiment Classification**  
sequence of words -> sentiment

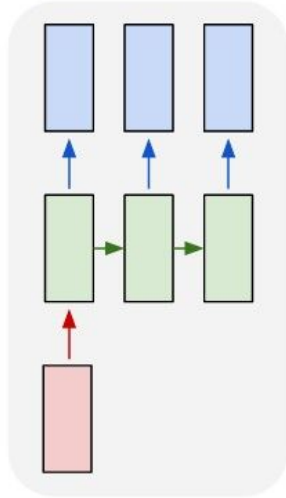


# Recurrent Networks offer a lot of flexibility:

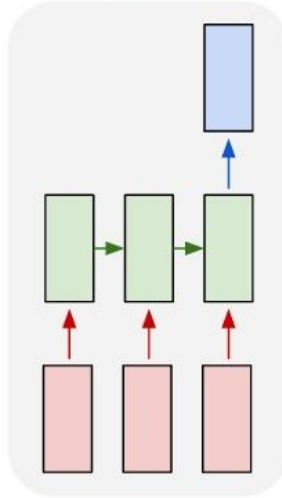
one to one



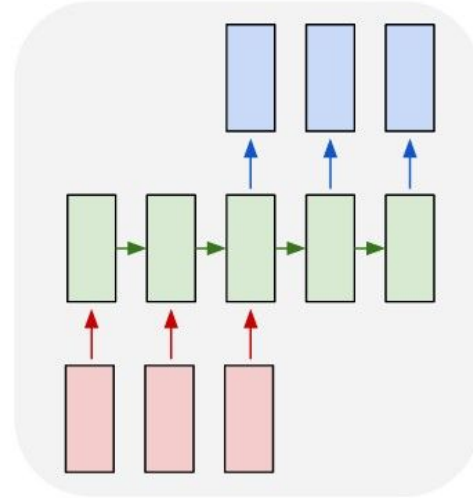
one to many



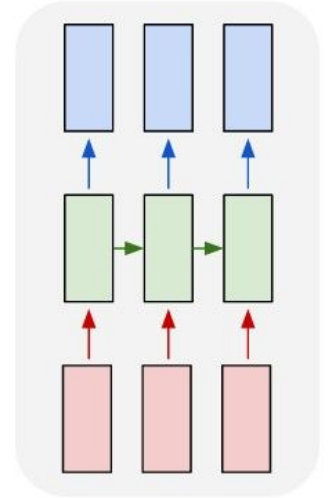
many to one



many to many



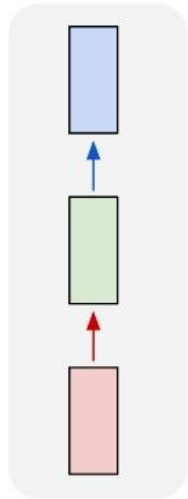
many to many



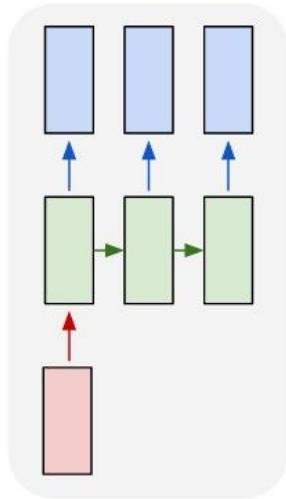
e.g. **Machine Translation**  
seq of words -> seq of words

# Recurrent Networks offer a lot of flexibility:

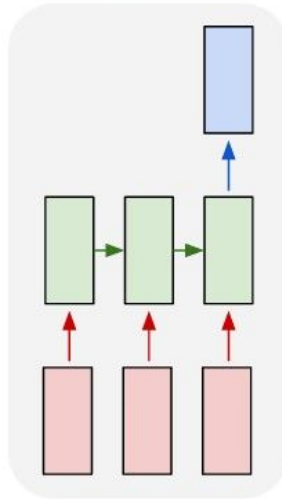
one to one



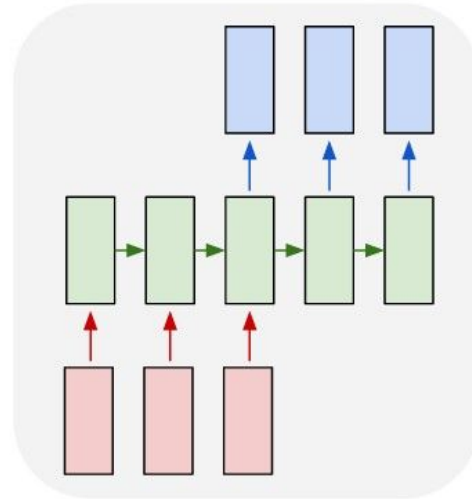
one to many



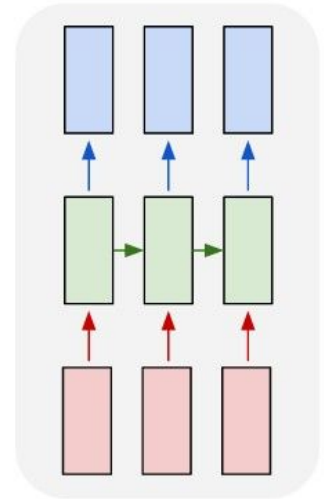
many to one



many to many



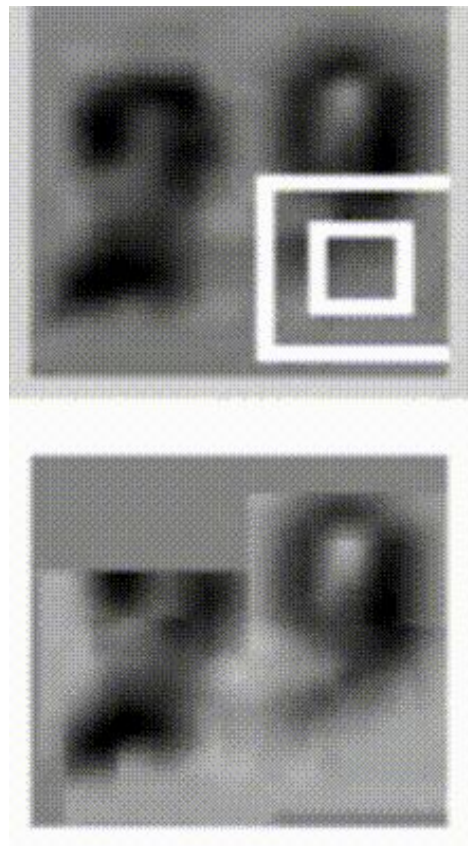
many to many



e.g. **Video classification on frame level**

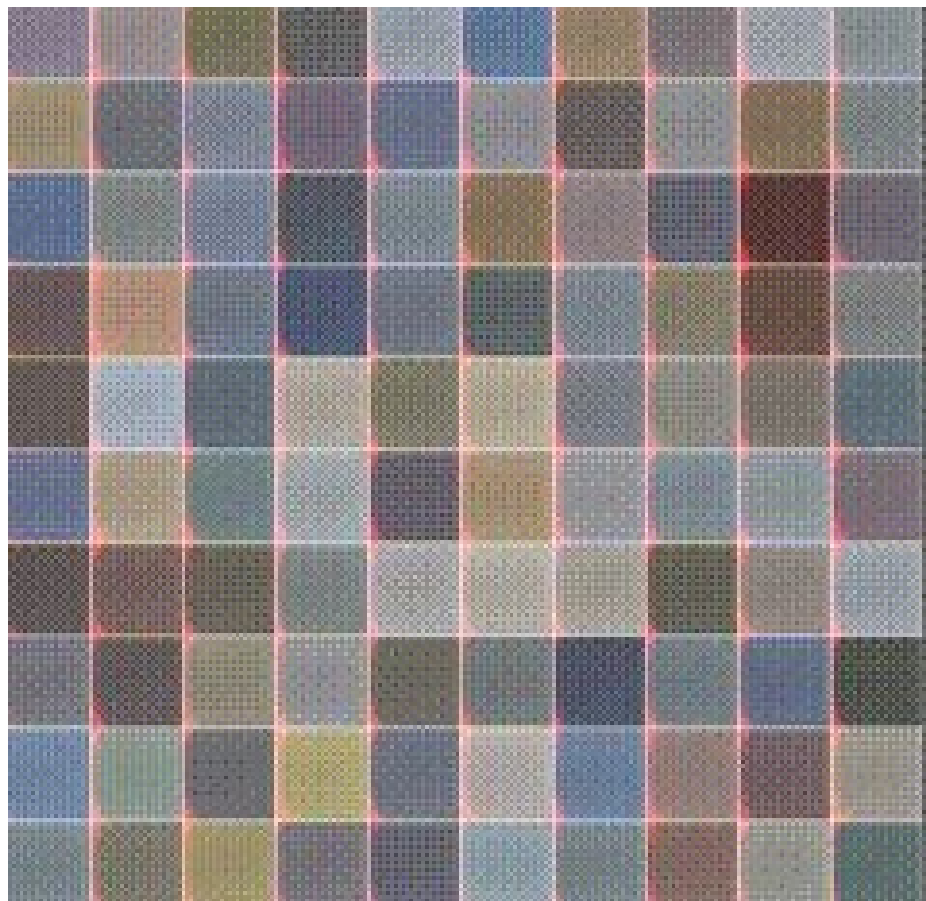
# Sequential Processing of fixed inputs

Multiple Object Recognition with  
Visual Attention, Ba et al.



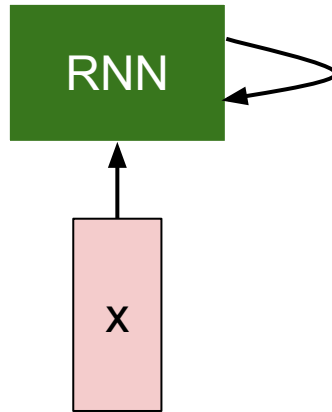
# Sequential Processing of fixed outputs

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

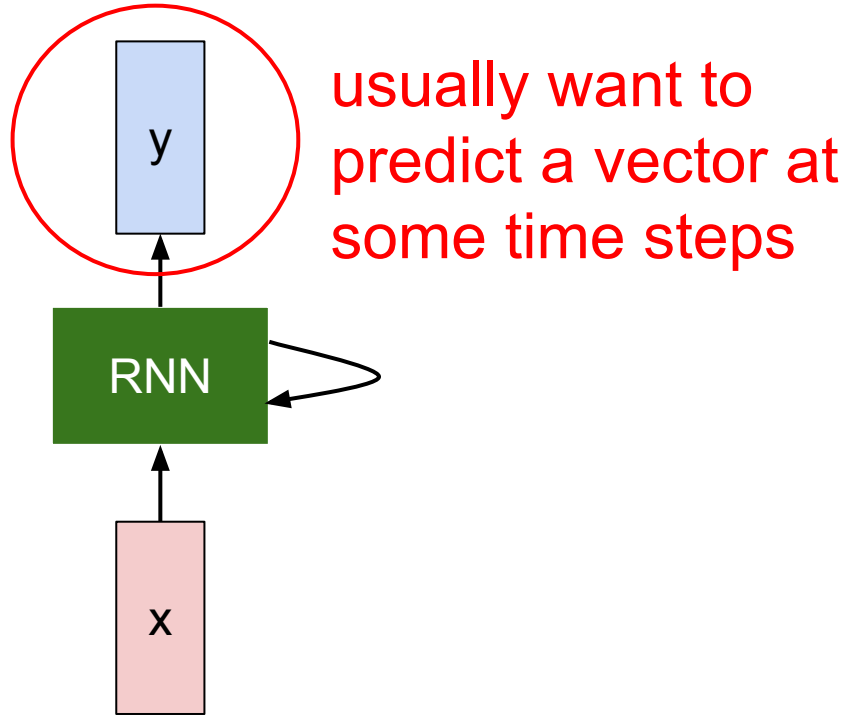




# Recurrent Neural Network



# Recurrent Neural Network

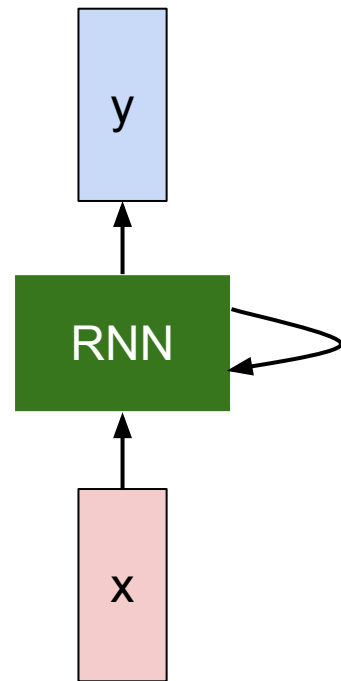


# Recurrent Neural Network

We can process a sequence of vectors  $\mathbf{x}$  by applying a recurrence formula at every time step:

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

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

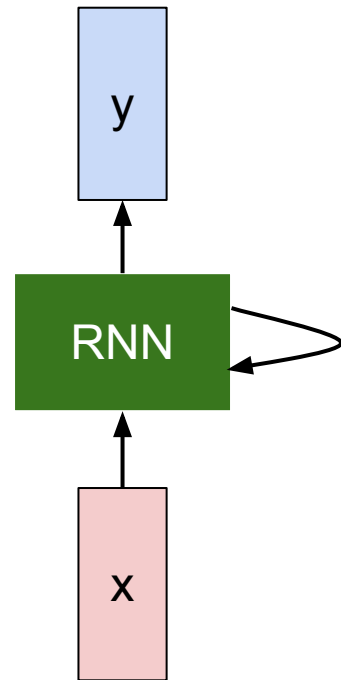


# Recurrent Neural Network

We can process a sequence of vectors  $\mathbf{x}$  by applying a recurrence formula at every time step:

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

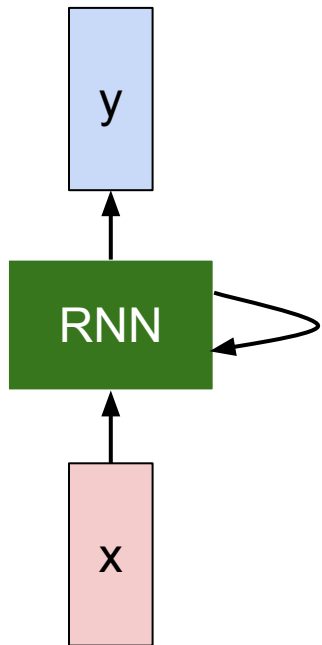
Notice: the same function and the same set of parameters are used at every time step.





# (Vanilla) Recurrent Neural Network

The state consists of a single “hidden” vector  $h$ :



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



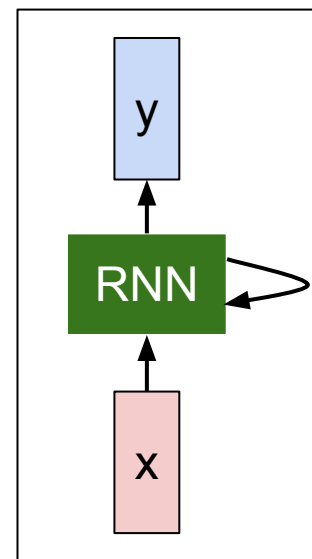
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

# Character-level language model example

Vocabulary:  
[h,e,l,o]

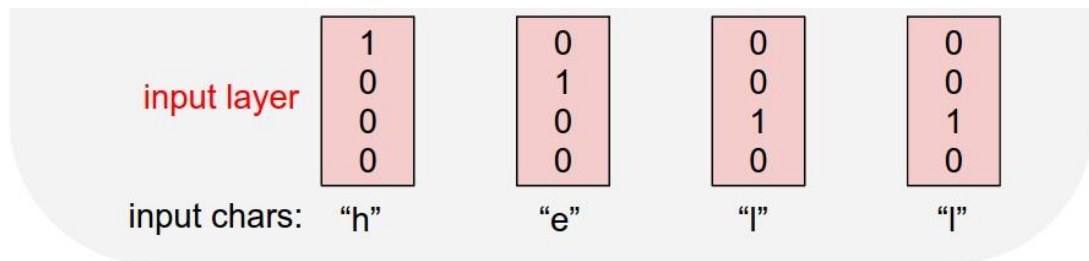
Example training  
sequence:  
“**hello**”



# Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”

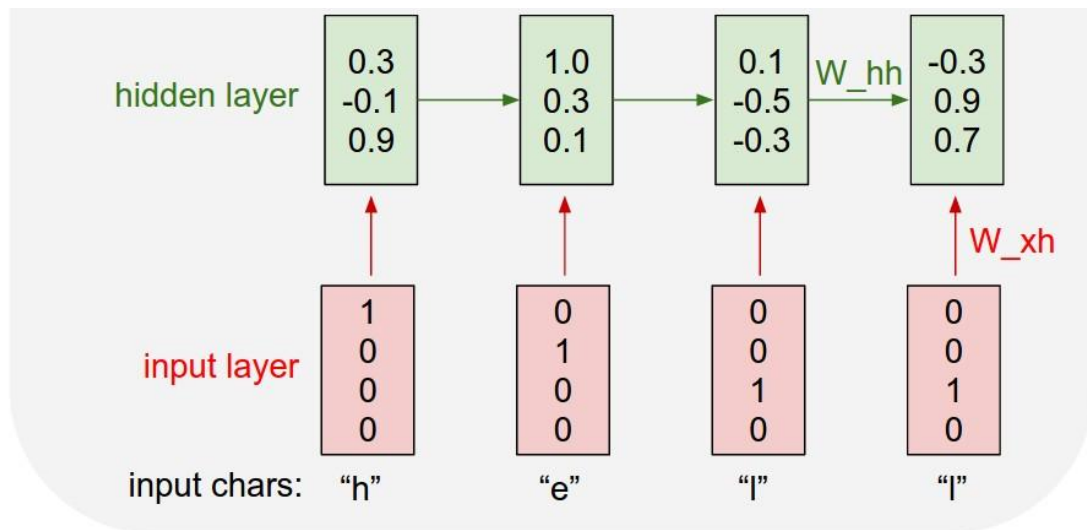


# Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training sequence:  
“hello”

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

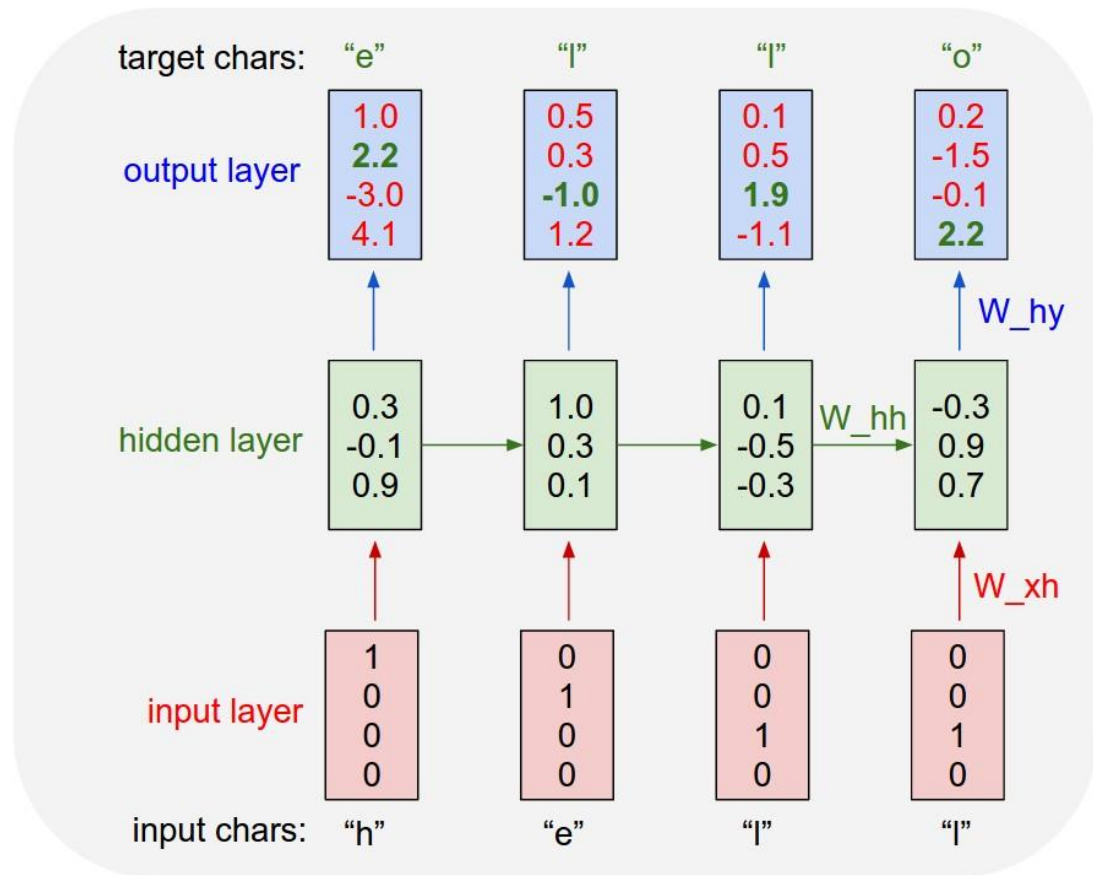




# Character-level language model example

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 wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(wxh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     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         dhrw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += dhrw
56         ddxh = np.dot(dhrw, xs[t].T)
57         dwhh += np.dot(dhrw, hs[t-1].T)
58         dhnext = np.dot(whh.T, dhrw)
59     for dparam in [dwxh, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwxh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

```
62 def sample(h, seed_ix, n):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     x = np.zeros((vocab_size, 1))
68     x[seed_ix] = 1
69     ixes = []
70     for t in xrange(n):
71         h = np.tanh(np.dot(wxh, x) + np.dot(whh, h) + bh)
72         y = np.dot(why, h) + by
73         p = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=p.ravel())
75         x = np.zeros((vocab_size, 1))
76         x[ix] = 1
77         ixes.append(ix)
78     return ixes
79
80 n, p = 0, 0
81 mxh, meth, mwhy = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)
82 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
83 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
84 while True:
85     # prepare inputs (we're sweeping from left to right in steps seq_length long)
86     if p+seq_length+1 >= len(data) or n == 0:
87         hprev = np.zeros((hidden_size,1)) # reset RNN memory
88         p = 0 # go from start of data
89         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
90         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
91
92     # sample from the model now and then
93     if n % 100 == 0:
94         sample_ix = sample(hprev, inputs[0], 200)
95         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
96         print '----\n %s \n----' % (txt, )
97
98     # forward seq_length characters through the net and fetch gradient
99     loss, dwxh, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
100     smooth_loss = smooth_loss * 0.999 + loss * 0.001
101     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
102
103     # perform parameter update with Adagrad
104     for param, dparam, mem in zip([dwxh, whh, why, bh, by],
105                                 [dwxh, dwhh, dwhy, dbh, dby],
106                                 [mxh, meth, mwhy, mbh, mby]):
107         mem += dparam * dparam
108         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
109
110     p += seq_length # move data pointer
111     n += 1 # iteration counter
```

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

# min-char-rnn.py gist

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

# Data I/O

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

```
14 # hyperparameters
15 hidden_size = 100 # size of hidden layer of neurons
16 seq_length = 25 # number of steps to unroll the RNN for
17 learning_rate = 1e-1
18
19 # model parameters
20 wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
21 ww = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
22 wh_ = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
23 bh = np.zeros((hidden_size, 1)) # hidden bias
24 by = np.zeros((vocab_size, 1)) # output bias
25
26 def lossfun(inputs, targets, h):
27     """
28     inputs, targets are both list of integers.
29     hprev is Nx1 array of initial hidden state
30     returns the loss, gradients on model parameters, and last hidden state
31     """
32     n, m, yk, oh = [], [], [], []
33     h[1:] = np.copy(hprev)
34     loss = 0
35     # forward pass
36     for t in xrange(len(inputs)):
37         x[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
38         x[t][inputs[t]] = 1
39         h[t] = np.tanh(np.dot(wh, x[t]) + np.dot(whh, h[t-1]) + bh) # hidden state
40         yk[t] = np.dot(wh_, h[t]) + by # unnormalized log probabilities for next chars
41         pt[t] = np.exp(yk[t]) / np.sum(np.exp(yk[t])) # probabilities for next chars
42         loss += -np.log(pt[t][targets[t],0]) # softmax (cross-entropy loss)
43     # backward pass: compute gradients going backward
44     dwh, dwh_, dby = np.zeros_like(wh), np.zeros_like(wh_), np.zeros_like(wh_)
45     dh, dby = np.zeros_like(h), np.zeros_like(by)
46     dhnext = np.zeros_like(h[0])
47     for t in reversed(xrange(len(inputs))):
48         dy = np.copy(pt[t])
49         dy[targets[t]] -= 1 # backprop into y
50         dby += np.dot(dy, h[t], T)
51         dby = dy
52         dh = np.dot(wh_ T, dy) + dhnext # backprop into h
53         dhrw = (1 - h[t]**2) * h[t]**2 # dh = backprop through tanh nonlinearity
54         dh = dhrw
55         dwh += np.dot(dhrw, x[t], T)
56         dwh_ += np.dot(dhrw, h[t], T)
57         dhnext = np.dot(wh T, dhraw)
58     for dparam in [wh, dwh, dwh_, dby, dh]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dwh, dwh_, dby, dh, h[len(inputs)-1]
61
62 def sample(h, seed_ix, n):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     x = np.zeros((vocab_size, 1))
68     s = list()
69     for i in xrange(n):
70         ix = list(range(vocab_size))
71         p = np.tanh(np.dot(wh, x) + np.dot(whh, h)) + bh
72         y = np.dot(wh_, p) # by
73         p = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=p, rand=1)
75         x = np.zeros((vocab_size, 1))
76         x[ix] = 1
77         s.append(ix)
78     return s
79
80 # h, p = 0, 0
81 mh, mh_, mby = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(wh_)
82 mh, mby = np.zeros_like(h), np.zeros_like(by) # memory variables for Adagrad
83 smooth_loss = -np.log(1.0/vocab_size)/seq_length # loss at iteration 0
84 while True:
85     # prepare inputs (we're sampling from left to right in steps seq_length long)
86     if prev_length == len(data) or n == 0:
87         hprev = np.zeros((hidden_size,1)) # reset rnn memory
88         p = 0 # no prev state of data
89         inputs = [char_to_ix[ch] for ch in data[prev_length:]]
90         targets = [char_to_ix[ch] for ch in data[prev_length+1:]]
91
92     # sample from the model one char then
93     if n % 100 == 0:
94         sample_ix = sample(hprev, inputs[0], 200)
95         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
96         print '....%s\n' % txt, # print
97
98     # forward pass: unroll RNN through the net and fetch gradients
99     loss, dwh, dwh_, dby, dh, hprev = lossfun(inputs, targets, hprev)
100     smooth_loss = smooth_loss * 0.999 + loss * 0.001
101     if n % 100 == 0: print 'iter %d, loss %f' % (n, smooth_loss) # print progress
102
103     # perform parameter update with Adagrad
104     for param, dparam, s in zip([wh, wh_, wh, by, bh], [dwh, dwh_, dby, dby, dh]):
105         [wh, wh_, wh, by, bh, dwh, dwh_, dby, dby, dh]
106         sm = s + dparam * dparam
107         param += -learning_rate * dparam / np.sqrt(sm + 1e-8) # adagrad update
108
109     p += seq_length # move data pointer
110     n += 1 # iteration counter
```

# min-char-rnn.py gist

```

1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD license
4  """
5  import numpy as np
6
7  # data I/O
8  data = open('input.txt', 'r').read() # should be some 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 # hyperparameters
15 hidden_size = 100 # size of hidden layer of neurons
16 seq_length = 25 # number of steps to unroll the RNN for
17 learning_rate = 1e-1
18

```

```

19 # model parameters
20 wlx = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
21 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
22 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
23 bh = np.zeros((hidden_size, 1)) # hidden bias
24 by = np.zeros((vocab_size, 1)) # output bias

```

```

25 def lossfun(inputs, targets, hprev):

```

```

26 """
27 inputs, targets are both list of integers.
28 hprev is list array of initial hidden state
29 returns the loss, gradients on model parameters, and last hidden state
30 """
31 n, m, n_h, n_o = [], [], 0, 0
32 h[0] = np.copy(hprev)
33 loss = 0
34 # forward pass
35 for t in xrange(len(inputs)):
36     x[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
37     x[t][inputs[t]] = 1
38     h[t] = np.tanh(np.dot(wlx, x[t]) + np.dot(whh, h[t-1]) + bh) # hidden state
39     y[t] = np.dot(why, h[t]) + by # unnormalized log probabilities for next chars
40     p[t] = np.exp(y[t]) / np.sum(np.exp(y[t])) # probabilities for next chars
41     loss += -np.log(p[t][targets[t], 0]) # softmax (cross-entropy) loss
42 # backward pass: compute gradients going backward
43 dwh, dwhy, dby = np.zeros_like(whh), np.zeros_like(why)
44 dh, dby = np.zeros_like(hh), np.zeros_like(hy)
45 dhnext = np.zeros_like(h[0])
46 for t in reversed(xrange(len(inputs))):
47     dy = np.copy(p[t])
48     dy[targets[t]] -= 1 # backprop into y
49     dby += np.dot(dy, h[t].T)
50     dby = dy
51     dh = np.dot(why.T, dy) + dhnext # backprop into h
52     ddraw = (1 - h[t]**2) * dh # dh = backprop through tanh nonlinearity
53     dwh = np.dot(ddraw, x[t].T)
54     dwhh = np.dot(ddraw, h[t-1].T)
55     dhnext = np.dot(whh.T, ddraw)
56     for dparam in [dwh, dwhy, dby, dh, dby]:
57         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients

```

```

58 return loss, dwh, dwhy, dby, dh, dby, h[len(inputs)-1]
59
60 def sample(h, ix):
61 """
62 sample a sequence of integers from the model
63 h is memory state, seed,ix is seed letter for first time step
64 """
65 x = np.zeros((vocab_size, 1))
66 ix_to_ix = ix
67 ixes = []
68 for t in xrange(1):
69     x = np.tanh(np.dot(wlx, x) + np.dot(whh, h) + bh)
70     y = np.dot(why, x) + by
71     p = np.exp(y) / np.sum(np.exp(y))
72     ix = np.random.choice(range(vocab_size), p=p, rand=True)
73     x = np.zeros((vocab_size, 1))
74     ixes.append(ix)
75 return ixes

```

```

76 #, p = 0, 0
77 mwh, mwhy = np.zeros_like(wlx), np.zeros_like(why)
78 mwh, mby = np.zeros_like(hh), np.zeros_like(hy) # memory variables for Adagrad
79 smooth_loss = np.log(1/float(seq_length)) # loss at iteration 0
80 while True:
81     # prepare steps (we're sampling from left to right in steps seq_length long)
82     if print_length > len(ixes) or n == 0:
83         hprev = np.zeros((hidden_size, 1)) # reset rnn memory
84         p = 0 # reset from start of data
85         inputs = [char_to_ix[ch] for ch in data[:print_length]]
86         targets = [char_to_ix[ch] for ch in data[print_length:]]
87
88     # sample from the model now and then
89     if n % 100 == 0:
90         sample_ix = sample(hprev, inputs[0], 0)
91         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
92         print '....%s\n' % txt
93
94     # forward seq length characters through the rnn and fetch gradients
95     loss, dwh, dwhy, dh, dby, hprev = lossfun(inputs, targets, hprev)
96     smooth_loss = smooth_loss * 0.999 + loss * 0.001
97     p = 1/n if n > 0 else print 'iter %d, loss %f' % (n, smooth_loss) # print progress
98
99     # perform parameter update with Adagrad
100     for dparam, dparam_name in zip([dwh, why, bh, by],
101                                  [dwh, dwhy, dby, dh, dby]):
102         sm = dparam * dparam
103         param_name += learning_rate * dparam / np.sqrt(sm + 1e-8) # adagrad update
104
105     p += seq_length # move data pointer
106     n += 1 # iteration counter

```

# Initializations

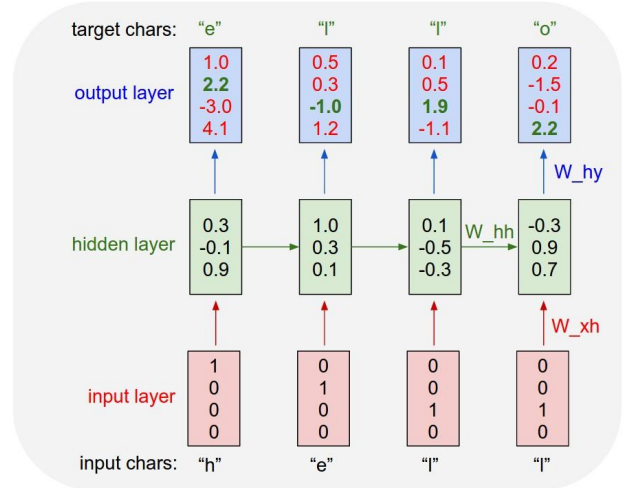
```

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

```



recall:





```

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 some 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 wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to output
24 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 Nxi array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     n, m, yk, ox = [], [], [], []
34     h[0] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xi[i] = np.zeros((vocab_size,1)) # encode i in 1-of-k representation
39         xi[i][inputs[t]] = 1
40         h[i] = np.tanh(np.dot(wih, xi[i]) + np.dot(whh, h[i-1]) + bh) # hidden state
41         yk[i] = np.dot(why, h[i]) + by # unnormalized log probabilities for next chars
42         pi[i] = np.exp(yk[i]) / np.sum(np.exp(yk[i])) # probabilities for next chars
43         loss += -np.log(pi[t][targets[t],0]) # softmax (cross-entropy) loss
44     # backward pass: compute gradients going backward
45     dwh, dwhh, dwhy = np.zeros_like(whh), np.zeros_like(whh), np.zeros_like(why)
46     dhh, dhb = np.zeros_like(hh), np.zeros_like(hb)
47     dhnext = np.zeros_like(h[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(pi[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dhb += np.dot(dy, h[t],T)
52         dhb += dy
53         dh = np.dot(why,T, dy) + dhnext # backprop into h
54         dhraw = (1 - h[i]**2) * dh # dh a backprop through tanh nonlinearity
55         dwh += np.dot(dhraw, xi[i],T)
56         dwhh += np.dot(dhraw, h[i],T)
57         dhnext = np.dot(whh,T, dhraw)
58     for dparam in [dwh, dwhh, dwhy, dhb, dhb]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dwh, dwhh, dwhy, dhb, dhb, dhh, dhb, h[len(inputs)-1]
61
62 def sample(h, ix, ix):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed,ix is seed letter for first time step
66     """
67     n = np.zeros((vocab_size, 1))
68     ix[seed,ix] = 1
69     ixes = []
70     for i in xrange(1):
71         n = np.tanh(np.dot(wih, xi) + np.dot(whh, h)) + bh
72         h = np.dot(why, h) + by
73         ix = np.exp(n) / np.sum(np.exp(n))
74         ix = np.random.choice(xrange(vocab_size), p=ix.ravel())
75         n = np.zeros((vocab_size, 1))
76         ixes.append(ix)
77     return ixes
78
79 n, p = 0, 0
80 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
81 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
82 while True:
83     # prepare inputs (we're sweeping from left to right in steps seq_length long)
84     if p+seq_length+1 >= len(data) or n == 0:
85         hprev = np.zeros((hidden_size,1)) # reset RNN memory
86         p = 0 # go from start of data
87     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
88     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
89
90     # sample from the model now and then
91     if n % 100 == 0:
92         sample_ix = sample(hprev, inputs[0], 200)
93         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
94         print '----\n %s \n----' % (txt, )
95
96     # forward seq_length characters through the net and fetch gradient
97     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
98     smooth_loss = smooth_loss * 0.999 + loss * 0.001
99     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
100
101     # perform parameter update with Adagrad
102     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
103                                 [dWxh, dWhh, dWhy, dbh, dby],
104                                 [mWxh, mWhh, mWhy, mbh, mby]):
105         mem += dparam * dparam
106         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
107
108     p += seq_length # move data pointer
109     n += 1 # iteration counter

```

## Main loop

```

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

```

# 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 some 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 wh = np.random.randn(hidden_size, hidden_size)*0.01 # input to hidden
22 whp = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     ns, nh, nhp, ns = [], [], [], []
34     h[0] = hprev
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         x[t] = np.zeros((vocab_size,1)) # encode 1-of-k representation
39         x[t][inputs[t]] = 1
40         h[t] = np.tanh(np.dot(wh, x[t]) + np.dot(whh, h[t-1]) + bh) # hidden state
41         y[t] = np.dot(why, h[t]) + by # unnormalized log probabilities for next chars
42         pt = np.exp(y[t]) / np.sum(np.exp(y[t])) # probabilities for next chars
43         loss += -np.log(pt)[targets[t],0] # softmax (cross-entropy) loss
44     # backward pass: compute gradients going backward
45     dwh, dwhp, dwhy = np.zeros_like(wh), np.zeros_like(why)
46     dh, dhv = np.zeros_like(h), np.zeros_like(h)
47     dhnext = np.zeros_like(h[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(pt[1])
50         dy[targets[t]] -= 1 # backprop into y
51         dby = np.dot(dy, h[t])
52         dhy = dy
53         dh = np.dot(why.T, dy) + dhnext # backprop into h
54         dhrw = (1 - h[t]**2) * dh # dh * backprop through tanh nonlinearity
55         dwh = dhrw
56         dhv = np.dot(dhrw, x[t])
57         dwhh = np.dot(dhrw, h[t-1])
58         dhnext = np.dot(whh.T, dhv)
59     for dparam in [dwh, dwhp, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwh, dwhp, dwhy, dbh, dby, h[len(inputs)-1]
62
63 def sample(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     h = hprev
70     seq_ix = 1
71     for i in xrange(n):
72         x = np.zeros_like(x)
73         h, p = np.dot(wh, x) + np.dot(whh, h) + bh
74         p = np.exp(p) / np.sum(np.exp(p))
75         ix = np.random.choice(vocab_size, 1, p=p).ravel()[0]
76         x[ix] = 1
77         h, loss = lossFun(x, ix, h)
78     return seq_ix
79
80 n, p = 0, 0
81 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
82 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
83
84 while True:
85     # prepare inputs (we're sweeping from left to right in steps seq_length long)
86     if p+seq_length+1 >= len(data) or n == 0:
87         hprev = np.zeros((hidden_size,1)) # reset RNN memory
88         p = 0 # go from start of data
89         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
90         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
91
92     # sample from the model now and then
93     if n % 100 == 0:
94         sample_ix = sample(hprev, inputs[0], 200)
95         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
96         print '----\n %s \n----' % (txt, )
97
98     # forward seq_length characters through the net and fetch gradient
99     loss, dwhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
100     smooth_loss = smooth_loss * 0.999 + loss * 0.001
101     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
102
103     # perform parameter update with Adagrad
104     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
105                                 [dwhx, dwhh, dwhy, dbh, dby],
106                                 [mWxh, mWhh, mWhy, mbh, mby]):
107         mem += dparam * dparam
108         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
109
110     p += seq_length # move data pointer
111     n += 1 # iteration counter
```

## Main loop

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

# min-char-rnn.py gist

```
1 """
2 Minimal character-level Vanilla RNN model, written by Andrej Karpathy (@karpathy)
3 BSD license
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be some plain text file
9 chrs = list(set(data))
10 data_size, vocab_size = len(data), len(chrs)
11 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
12 char_to_ix = { ch:i for i, ch in enumerate(chrs) }
13 ix_to_char = { i:ch for i, ch in enumerate(chrs) }
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 wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to output
24 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 Nxt array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     n, m, yk, yn = [], [], [], []
34     h[0] = hprev
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         x[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         x[t][inputs[t]] = 1
40         h[t] = np.tanh(np.dot(wih, x[t]) + np.dot(whh, h[t-1]) + bh) # hidden state
41         yk[t] = np.dot(why, h[t]) + by # unnormalized log probabilities for next chars
42         pi[t] = np.exp(yk[t]) / np.sum(np.exp(yk[t])) # probabilities for next chars
43         loss += -np.log(pi[t][targets[t]]) # softmax (cross-entropy) loss
44     # backward pass: compute gradients going backward
45     dwh, dwhh, dwhy = np.zeros_like(whh), np.zeros_like(why)
46     dh, dhv = np.zeros_like(h), np.zeros_like(hv)
47     dhnext = np.zeros_like(h[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(pi[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dby = np.dot(dy, h[t:T])
52         dhy = dy
53         dh = np.dot(why, T, dy) + dhnext # backprop into h
54         ddraw = (1 - h[t]**2) * dh # dh * backprop through tanh nonlinearity
55         dwh = ddraw
56         dwhh = np.dot(ddraw, x[t:T])
57         dhv = np.dot(ddraw, h[t:T])
58         dparam = [-5, 5, out-dparam] # clip to mitigate exploding gradients
59     return loss, dwh, dwhh, dwhy, dhv, dh, dby, dhnext[inputs[-1]]
60
61 def sample(seed_ix, n):
62     """
63     sample a sequence of integers from the model
64     h is memory state, seed_ix is seed letter for first time step
65     """
66     x = np.zeros((vocab_size, 1))
67     h = []
68     for i in xrange(n):
69         x = np.tanh(np.dot(wih, x) + np.dot(whh, h) + bh)
70         h = np.dot(why, h) + by
71         a = np.exp(x) / np.sum(np.exp(x))
72         ix = np.random.choice(range(vocab_size), p=a.ravel())
73         x = np.zeros((vocab_size, 1))
74         h.append(x)
75     return h
76
77 n, p = 0, 0
78 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
79 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
80 while True:
81     # prepare inputs (we're sweeping from left to right in steps seq_length long)
82     if p+seq_length+1 >= len(data) or n == 0:
83         hprev = np.zeros((hidden_size,1)) # reset RNN memory
84         p = 0 # go from start of data
85         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
86         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
87
88     # sample from the model now and then
89     if n % 100 == 0:
90         sample_ix = sample(hprev, inputs[0], 200)
91         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
92         print '----\n %s \n----' % (txt, )
93
94     # forward seq_length characters through the net and fetch gradient
95     loss, dwhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
96     smooth_loss = smooth_loss * 0.999 + loss * 0.001
97     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
98
99     # perform parameter update with Adagrad
100     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
101                                 [dwhx, dwhh, dwhy, dbh, dby],
102                                 [mWxh, mWhh, mWhy, mbh, mby]):
103         mem += dparam * dparam
104         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
105
106     p += seq_length # move data pointer
107     n += 1 # iteration counter
```

## Main loop

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





```

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 some 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 wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both lists of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     nI, nH, nO, nB = len(inputs), len(hprev), len(targets), 1
34     hI[0] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xiI = np.zeros((vocab_size, 1))
39         xiI[inputs[t]] = 1
40         hI[1:] = np.tanh(np.dot(wih, xiI) + np.dot(whh, hI[-1]) + bh) # hidden state
41         yI[0] = np.dot(why, hI[1]) + by # unnormalized log probabilities for next chars
42         piI = np.exp(yI) / np.sum(np.exp(yI)) # probabilities for next chars
43         loss += -np.log(piI[targets[t], 0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backward
45     dwh, dwhh, dwhy = np.zeros_like(whh), np.zeros_like(why)
46     dht, dby = np.zeros_like(hI), np.zeros_like(by)
47     dhnext = np.zeros_like(hI[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(piI)
50         dy[targets[t]] -= 1 # backprop into y
51         dby += np.dot(dy, hI[t, 1])
52         dht = dy
53         dh = np.dot(why, T, dy) + dhnext # backprop into h
54         dhtanh = (1 - hI[t, 1]**2) * dh # a backprop through tanh nonlinearity
55         dwh += np.dot(dhtanh, xiI[1])
56         dwhh += np.dot(dhtanh, hI[t, 1])
57         dhnext = np.dot(whh, T, dhtanh)
58     for dparam in [dwh, dwhh, dwhy, dht, dby]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dwh, dwhh, dwhy, dht, dby, hI[len(inputs)-1]
61
62 def sample(seed_ix, n):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     x = np.zeros((vocab_size, 1))
68     h = hprev[seed_ix-1]
69     for t in xrange(n):
70         xi = x
71         h, x = np.tanh(np.dot(wih, xi) + np.dot(whh, h)) + bh
72         y = np.dot(why, h) + by
73         a = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=a, reseed=True)
75         x = np.zeros((vocab_size, 1))
76         x[ix] = 1
77     loss.append(x)
78     return loss
69
79 n, p = 0, 0
80 mbh, mby, mwhy = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
81 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
82 while True:
83     # prepare inputs (we're sweeping from left to right in steps seq_length long)
84     if p+seq_length+1 >= len(data) or n == 0:
85         hprev = np.zeros((hidden_size,1)) # reset RNN memory
86         p = 0 # go from start of data
87     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
88     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
89
90     # sample from the model now and then
91     if n % 100 == 0:
92         sample_ix = sample(hprev, inputs[0], 200)
93         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
94         print '----\n %s \n----' % (txt, )
95
96     # forward seq_length characters through the net and fetch gradient
97     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
98     smooth_loss = smooth_loss * 0.999 + loss * 0.001
99     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
100
101     # perform parameter update with Adagrad
102     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
103                                 [dWxh, dWhh, dWhy, dbh, dby],
104                                 [mWxh, mWhh, mWhy, mbh, mby]):
105         mem += dparam * dparam
106         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
107
108     p += seq_length # move data pointer
109     n += 1 # iteration counter

```

## Main loop

```

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

```

## Loss function

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

```

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 some 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 wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whv = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 wb = 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

```

```

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(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(whv, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     dwh, dwhv, dwhy = np.zeros_like(wh), np.zeros_like(whv), 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         dby += np.dot(dy, hs[t].T)
52         dby += dy
53         dh = np.dot(why, T, dy) + dhnext # backprop into h
54         dhrw = (1 - hs[t]**2) * dh # dh * backprop through tanh nonlinearity
55         dbh += dhrw
56         dwhv += np.dot(dhrw, hs[t].T)
57         dwhb += np.dot(dhrw, hs[t].T)
58         dwhmax = np.dot(whh, T, dhrw)
59         for dparam in [wh, dwh, dbh, dwhy, dbh, dby]:
60             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwh, dwhv, dwhy, dbh, dby, hs[len(inputs)-1]

```

```

62 def sample(next_ix, y):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     x = np.zeros(vocab_size, 1)
68     x[seed_ix] = 1
69     ixes = []
70     for i in xrange(10):
71         h = np.tanh(np.dot(wh, x) + np.dot(whh, h) + bh)
72         y = np.dot(whv, h) + by
73         p = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=p, rand=True)
75         x = np.zeros(vocab_size, 1)
76         x[ix] = 1
77         ixes.append(ix)
78     return ixes
79
80 # R, P, S, O
81 rnn, dwh, dwhv = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
82 rnnh, dby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
83 unroll_loss = np.log(1.0/vocab_size)/seq_length # loss at iteration 0
84 while True:
85     # prepare inputs (we're sampling from left to right in steps seq_length long)
86     if pprev_length is len(data) or n == 0:
87         hprev = np.zeros(hidden_size, 1) # reset rnn memory
88         p = 0 # 0 for first char of data
89         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
90         targets = [char_to_ix[ch] for ch in data[p+seq_length:p+seq_length+1]]
91
92     # sample from the model row and then
93     if n % 100 == 0:
94         sample_ix = sample(targets, inputs[0], 0)
95         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
96         print '....%8d %s....' % (n, txt)
97
98     # forward seq_length characters through the net and fetch gradients
99     loss, dwh, dwhv, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
100     unroll_loss = unroll_loss + (loss - unroll_loss) * 0.999
101     if n % 100 == 0: print 'iter %d, loss %f' % (n, unroll_loss) # print progress
102
103     # perform parameter update with Adagrad
104     for param, dparam in zip([wh, whv, whb, bh, by], [dwh, dwhv, dwhy, dbh, dby]):
105         rnn += dparam * dparam
106         param -= learning_rate * dparam / np.sqrt(rnn + 1e-8) # adagrad update
107
108     p += seq_length # move data pointer
109     n += 1 # iteration counter

```

```

27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = [], [], [], []
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros(vocab_size, 1) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     dWxh, dWwh, 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         dhrw = (1 - hs[t]**2) * dh # dh * backprop through tanh nonlinearity
55         dbh += dhrw
56         dWxh += np.dot(dhrw, xs[t].T)
57         dWwh += np.dot(dhrw, hs[t-1].T)
58         dhnext = np.dot(Whh, T, dhrw)
59     for dparam in [dWxh, dWwh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dWxh, dWwh, dWhy, dbh, dby, hs[len(inputs)-1]

```



# min-char-rnn.py gist

```
1 """
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD license
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be some 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 wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 ww = 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
```

```
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(whx, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backward
45     dhs, dwh, dww, dwhy = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(why)
46     dby = np.zeros_like(by)
47     dbh = np.zeros_like(bh)
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(ps[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dby += np.dot(dy, why)
52         dht = dy
53         dh = np.dot(why.T, dy) + dbh # backprop into h
54         dhrav = (-1 - h[t]**2) * dh # dh = backprop through tanh nonlinearity
55         dwh += np.dot(dhrav, xs[t])
56         dww += np.dot(dhrav, hs[t-1])
57         dhw = np.dot(wh.T, dhrav)
58         for param in [wh, dwh, dww, dwhy, dbh, dby]:
59             np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
60     return loss, dwh, dww, dwhy, dh, dby, hs[len(inputs)-1]
```

```
61 # sample
62 """
63 sample a sequence of integers from the model
64 h is hidden state, seed_ix is seed letter for first time step
65 """
66 x = np.zeros((vocab_size, 1))
67 s = [seed_ix]
68 hses = []
69 for t in xrange(10):
70     xs = np.zeros((vocab_size, 1))
71     xs[s[-1]] = 1
72     h, p = lossFun(xs, s, h)
73     s = np.dot(why, h) + by
74     p = np.exp(p) / np.sum(np.exp(p))
75     ix = np.random.choice(vocab_size, 1, p=p.ravel())
76     x = np.zeros((vocab_size, 1))
77     x[ix] = 1
78     hses.append(x)
79     return hses
80
81 #, p = 0, 0
82 dwh, dww, dwhy = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(why)
83 dbh, dby = np.zeros_like(bh), np.zeros_like(by) # memory variables for adagrad
84 smooth_loss = -np.log(1.0/vocab_size)/seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sampling from left to right in steps seq_length long)
87     if prev_length is len(chars) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset rnn memory
89         p = 0 # no prev state of data
90         inputs = [char_ix[ix] for ix in data[:prev_length]]
91         targets = [char_ix[ix] for ix in data[prev_length:]]
92
93     # sample from the model row and then
94     if n % 100 == 0:
95         sample_ix = sampleFromProb(dwh, dww, dwhy, dbh, dby, hses, targets, hprev)
96         text = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '....%8d %s' % (n, text)
98
99     # forward seq length characters through the rnn and fetch gradients
100     loss, dwh, dww, dwhy, dh, dby, hprev = lossFun(inputs, targets, hprev)
101     smooth_loss = smooth_loss * 0.99 + loss * 0.01
102     if n % 100 == 0: print 'iter %d, loss %f' % (n, smooth_loss) # print progress
103
104     # perform parameter update with adagrad
105     for param, dparam, nnn in zip([dwh, dww, why, wh, bh, by],
106                                 [dwh, dww, dwhy, dbh, dby],
107                                 [dwh, dww, dwhy, dbh, dby]):
108         nnn += dparam * dparam
109         param -= learning_rate * dparam / np.sqrt(nnn + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
```

```
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(whx, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

# min-char-rnn.py gist

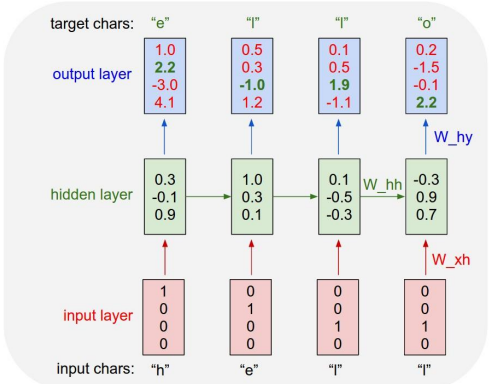
```
1 """
2 Minimal character-level Vanilla RNN model, written by Andrej Karpathy (@karpathy)
3 BSD license
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be some plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print('data 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 wh = np.random.randn(hidden_size, hidden_size) # input to hidden
22 wh = np.random.randn(hidden_size, hidden_size) # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size) # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
```

```
27 def lossfun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nxi array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hxs, yts, os = [], [], [], []
34     hxi[0] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xi[t] = np.zeros((vocab_size,1)) # encode in 1-of-K representation
39         xi[xi[inputs[t]]] = 1
40         hxi[t] = np.tanh(np.dot(wh, xi[t]) + np.dot(whh, hxi[t-1]) + bh) # hidden state
41         yts[t] = np.dot(why, hxi[t]) + by # unnormalized log probabilities for next chars
42         pi[t] = np.exp(yts[t]) / np.sum(np.exp(yts[t])) # probabilities for next chars
43         loss += -np.log(pi[t][targets[t],0]) # softmax cross-entropy loss
44     # backward pass: compute gradients going backwards
45     dbh, dwh, dwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
46     dh, dby = np.zeros_like(h), np.zeros_like(by)
47     dhnext = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(pi[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dwhy += np.dot(dy, hs[t].T)
52         dbh += dwhy
53         dh = np.dot(Why.T, dy) + dhnext # backprop into h
54         dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += dhrw
56         dwxh += np.dot(dhraw, xs[t].T)
57         dwhh += np.dot(dhraw, hs[t-1].T)
58         dhnext = np.dot(Whh.T, dhraw)
59     for dparam in [dwxh, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dbh, dwh, dwhy, dbh, dby, hxi[len(inputs)-1]
```

```
def weights_and_biases():
    """
    sample a sequence of integers from the model
    h is memory state, seed_ix is seed letter for first time step
    """
    n = np.zeros((vocab_size, 1))
    h = [0]
    for i in xrange(1):
        xi = np.zeros((vocab_size, 1))
        xi[seed_ix] = 1
        for t in xrange(1):
            xi[xi[inputs[t]]] = 1
            hxi[t] = np.tanh(np.dot(wh, xi[t]) + np.dot(whh, hxi[t-1]) + bh) # hidden state
            yts[t] = np.dot(why, hxi[t]) + by # unnormalized log probabilities for next chars
            pi[t] = np.exp(yts[t]) / np.sum(np.exp(yts[t])) # probabilities for next chars
            loss += -np.log(pi[t][targets[t],0]) # softmax cross-entropy loss
            n += pi[t]
        hxi.append(hxi[-1])
        return loss, n
    return loss, n
n, p = 0, 0
mh, mwh, mwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
mh, mwh, mwhy = np.zeros_like(h), np.zeros_like(h), np.zeros_like(h)
smooth_loss = -np.log(pi[0][targets[0],0]) # loss at iteration 0
while True:
    # prepare inputs (we're unrolling from left to right in steps seq_length long)
    if p == 0:
        hprev = np.zeros((hidden_size, 1)) # reset new memory
        p = 0 # go from start of data
        inputs = [char_to_ix[ch] for ch in data[:p+seq_length]]
        targets = [char_to_ix[ch] for ch in data[p+seq_length+1]]
    # sample from the model now and then
    if n % 100 == 0:
        sample_ix = sampleFromProb(pi[0][0], 100)
        text = ''.join(ix_to_char[ix] for ix in sample_ix)
        print('....%s\n' % text)
    # forward seq_length characters through the net and fresh gradients
    loss, dbh, dwh, dwhy, dbh, dby, hprev = lossfun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.99 + loss * 0.01
    if n % 100 == 0:
        print('iter %d, loss %f' % (n, smooth_loss)) # print progress
    # perform parameter update with Adagrad
    for param, dparam, new in zip((wh, whh, why, bh, by),
                                  (dbh, dwh, dwhy, dbh, dby),
                                  (mh, mwh, mwhy, mbh, mby)):
        new += dparam
        param -= learning_rate * dparam / np.sqrt(new + 1e-8) # Adagrad update
    p += seq_length # move data pointer
    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(pi[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     dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55     dbh += dhrw
56     dwxh += np.dot(dhraw, xs[t].T)
57     dwhh += np.dot(dhraw, hs[t-1].T)
58     dhnext = np.dot(Whh.T, dhraw)
59 for dparam in [dwxh, dwhh, dwhy, dbh, dby]:
60     np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61 return loss, dwxh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

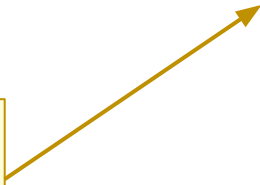
recall:



# min-char-rnn.py gist

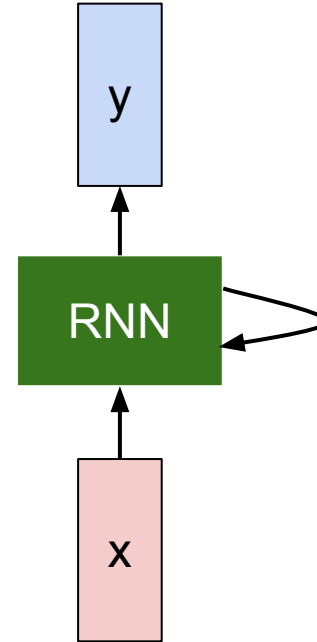
```
1 """
2 Minimal character-level Vanilla RNN model, written by Andrej Karpathy (@karpathy)
3 BSD license
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be some plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print '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 wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 wh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossfun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     w, wh, wht, w = [], [], [], []
34     h[1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         x[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         x[t][inputs[t]] = 1
40         h[t] = np.tanh(np.dot(wh, x[t]) + np.dot(whh, h[t-1]) + bh) # hidden state
41         y[t] = np.dot(why, h[t]) + by # unnormalized log probabilities for next chars
42         p[t] = np.exp(y[t]) / np.sum(np.exp(y[t])) # probabilities for next chars
43         loss += -np.log(p[t][targets[t],0]) # softmax (cross-entropy) loss
44     # backward pass: compute gradients going backward
45     dwh, dwhh, dwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
46     dh, dh1 = np.zeros_like(h), np.zeros_like(h)
47     dhnext = np.zeros_like(h[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(p[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dht = np.dot(dy, h[t].T)
52         dh1 = dy
53         dh = np.dot(why.T, dy) + dhnext # backprop into h
54         ddraw = (1 - h[t]**2) * dh1 # dh a backprop through tanh nonlinearity
55         dwh += np.dot(ddraw, x[t].T)
56         dwhh += np.dot(ddraw, h[t].T)
57         dhnext = np.dot(wh.T, ddraw)
58     for dparam in [dwh, dwhh, dwhy, dh, dh1]:
59         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dwh, dwhh, dwhy, dh, dh1, h[len(inputs)-1]
61
62 def sample(h, seed_ix, n):
63     """
64     sample a sequence of integers from the model
65     h is memory state, seed_ix is seed letter for first time step
66     """
67     x = np.zeros((vocab_size, 1))
68     x[seed_ix] = 1
69     ixes = []
70     for t in xrange(n):
71         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
72         y = np.dot(Why, h) + by
73         p = np.exp(y) / np.sum(np.exp(y))
74         ix = np.random.choice(range(vocab_size), p=p.ravel())
75         x = np.zeros((vocab_size, 1))
76         x[ix] = 1
77         ixes.append(ix)
78     return ixes
79
80 # main
81 """
82 """
83 # sample a sequence of integers from the model
84 h is memory state, seed_ix is seed letter for first time step
85 """
86 x = np.zeros((vocab_size, 1))
87 ixes = []
88
89 for t in xrange(10):
90     x = np.zeros((vocab_size, 1))
91     x[seed_ix] = 1
92     ixes = []
93     for t in xrange(n):
94         h = np.tanh(np.dot(wh, x) + np.dot(whh, h) + bh)
95         y = np.dot(why, h) + by
96         p = np.exp(y) / np.sum(np.exp(y))
97         ix = np.random.choice(range(vocab_size), p=p.ravel())
98         x = np.zeros((vocab_size, 1))
99         x[ix] = 1
100         ixes.append(ix)
101     return ixes
102
103 # main
104 """
105 """
106 # sample from the model one and then
107 if n % 100 == 0:
108     sample_ix = sample(hprev, inputs[0], 200)
109     txt = ''.join(ix_to_char[ix] for ix in sample_ix)
110     print '----%s%s-----' % (txt, ' ')
111
112 # a few long length characters through the net and fresh gradients
113 loss, dwh, dwhh, dwhy, dh, dh1, hprev = lossfun(inputs, targets, hprev)
114 smooth_loss = smooth_loss + loss * 0.999 + loss * 0.001
115 if n % 100 == 0: print 'iter %d, loss %f' % (n, smooth_loss) # print progress
116
117 # perform parameter update with Adagrad
118 for dparam, dparam_smooth in zip([dwh, wh, why, dh, dh1], [dwh, dwhh, dwhy, dh, dh1]):
119     param_smooth += dparam * dparam / np.sqrt(param_smooth + 1e-8) # adagrad update
120     param = param_smooth - learning_rate * dparam / np.sqrt(param_smooth + 1e-8) # adagrad update
121
122 # = seq_length # move data pointer
123 # = 1 # iteration counter
```

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



«p class="clear» Products: Laser Printers The fundamental everyday requirement for mono and colour laser printing throughout today's office is perfectly met with the extensive Epson laser printer range. The latest AcU-Laser printer range offers users exceptionally

Epson AcU-Laser C1900 Networked compact colour laser printer for professional enterprises. Businesses have been denied simple and affordable colour laser printing for far too long. The traditionally high costs and poor speeds of colour lasers has left many offices looking a bit, well, grey. But not any more: with the Epson AcU-Laser C1900, Epson brings both colour and monochrome laser printing together at a black and white price. more Where to Buy Support Epson AcU-Laser C3000 The fastest colour laser printer in its class. The perfect printer for small businesses and work groups, the Epson AcU-Laser C3000 prints high volumes in black and white and vibrant colour, at high speed and with low running costs. more Where to Buy High quality resolution: 2400dpi R7† Large paper capacity: 600 sheets, expandable up to 1,000 sheets† Compatible Windows and Mac. High speed USB and EpsonNet 10/100 Base-TX Ethernet interfaces as standard† \* Epson AcU-Laser Resolution Improvement Technology,†EpsonNet 10/100 Base-TX Ethernet standard with Epson AcU-Laser C3000† model only. AcU-Laser C1900 64MB Memory, 100 sheet MP Tray, 500 sheet cassette, Duplex printing as standard AcU-Laser C3000 64MB Memory, 100 sheet MP Tray, 500 sheet cassette, Duplex printing, 10/100Base-TX Ethernet Interface Networked compact colour laser printer for professional enterprises. Businesses have been denied simple and affordable colour laser printing for far too long. The traditionally high costs and poor speeds of colour lasers has left many offices looking a bit, well, grey. But not any more: with the Epson AcU-Laser C1900, Epson brings both colour and monochrome laser printing together at a black and white price. Key features cost effective mono printing for day to day business needs and a vivid variable colour when required. Search Epson UK Epson AcU-Laser C900 Outstanding professional colour printing for business. Add colour to your business with the Epson AcU-Laser C900 from Epson. Its perfect for the smaller workgroup, being a compact and cost effective laser printing workhorse that offers amazing colour output as well as high performance black and white production. more Where to Buy Support As cost efficient to run as a mono-only laser printer. Paper capacity of 700 sheets from two media sources. Easy to operate with advanced print driver. Memory expansion from 32MB to 1024MB. Pre-configured models available with Wireless 802.11b, Adobe® PostScript® 3 Level 3™ and two-sided printing. The AcU-Laser C1900 is available in 5 configurations. - AcU-Laser C1900: with 200B, 200 Sheet MP Tray, 10/100Base-TX Networking. -AcU-Laser C1900: with 320B, 200 Sheet MP Tray, 500 Sheet Cassette, 10/100Base-TX Networking. Support Epson AcU-Laser C4100 High performance colour laser for all your business printing needs. The Epson AcU-Laser C4100 provides businesses with a high performance colour and monochrome printing solution. It adds crucial colour to your business, while producing high quality monochrome output at lower costs than many monochrome-only printers, and is just as easy to operate. So now there's no reason to buy two printers, because perfect monochrome and colour solutions are available in one. more Where to Buy Support Epson AcU-Laser C5600 Professional high performance A3W colour laser printer. Epson AcU-Laser C5600 is the perfect professional printing solution for users who require exceptional quality colour and mono output on a range of media formats from C5 up to A3W in size. The Epson AcU-Laser C5600 is able to achieve superb print quality by utilising a combination of Epson's exclusive AcU-Laser Colour Laser Technologies. more Where to Buy Support - AcU-Laser C1900PS: with Acobe® PostScript® 3™, 96MB, 200 Sheet MP Tray, 500 Sheet Cassette, 10/100Base-TX Networking. - AcU-Laser C1900: with Duplex unit (two sided printing) 96MB, 200 Sheet MP Tray, 500 Sheet Cassette, 10/100Base-TX Networking. - AcU-Laser C1900 WFi with 320B, 200 Sheet MP Tray, 500 Sheet Cassette, Wireless Networking facility. Add colour to your business with the Epson AcU-Laser C900 from Epson. Its perfect for the smaller workgroup, being a compact and cost effective laser printing workhorse that offers amazing colour output as well as high. Support Epson AcU-Laser C400 High performance colour laser. The Epson AcU-Laser C400 provides businesses with high performance colour and monochrome printing solutions. more Where to Buy Epson AcU-Laser C8100 High speed laser printer. Why have separate black and white and colour printers when you can have the Epson AcU-Laser C8100? Epson has taken the lead in laser technology to deliver a complete high-performance solution for all your colour and mono printing needs. Support. EPL-6200, high performance A4 mono laser professional printers. The Epson EPL-6200 and EPL-6200L are the ideal printing solutions for small to medium workgroups and personal users. They deliver professional performance quickly, easily, reliably and cost-effectively, and are perfect for users who need high levels of laser quality and productivity at a low investment. more Where to Buy Support. EPL-6200 high performance A4 mono laser professional printers. The Epson EPL-6200 and EPL-6200L are the ideal printing solutions for small to medium workgroups and personal users. They deliver professional performance quickly, easily, reliably and cost-effectively, and are perfect for users who need high levels of laser quality and productivity at a low investment. more performance black and white production. For the first time, you can now bring the power of high quality colour to your documents without suffering the high costs or low speeds traditionally associated with colour





## 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 'nhtnee e  
plia tklrqd t o idoe ns,smtt h ne etie h,hregtrs niglike,aoaenns lng

↓  
train more

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

↓  
train more

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

↓  
train more

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

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 nudes 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:

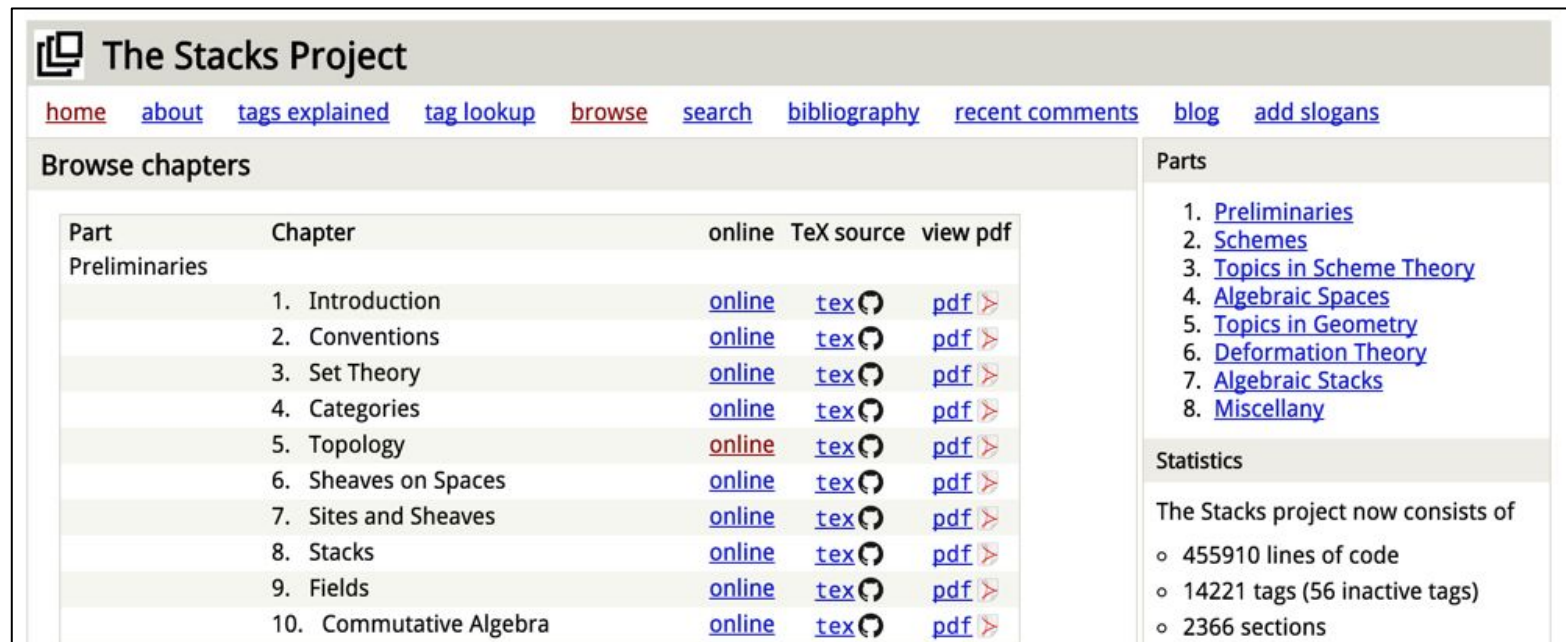
Why, Salisbury must find his flesh and thought  
That which I am not apt, not a man and in fire,  
To show the reining of the raven and the wars  
To grace my hand reproach within, and not a fair are hand,  
That Caesar and my goodly father's world;  
When I was heaven of presence and our fleets,  
We 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	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	2. Conventions	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	3. Set Theory	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	4. Categories	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	5. Topology	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	6. Sheaves on Spaces	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	7. Sites and Sheaves	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	8. Stacks	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	9. Fields	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>
	10. Commutative Algebra	<a href="#">online</a>	<a href="#">tex</a>	<a href="#">pdf</a>

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  $Sh(G)$  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  $GL_{S'}(x'/S'')$  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

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

and

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

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  $X_{spaces, \acute{e}tale}$  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  $\text{Proj}_X(\mathcal{A}) = \text{Spec}(B)$  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  $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{X, \dots, 0}$ .

**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  $i_0 \leq \mathfrak{p}$  is a subset of  $\mathcal{J}_{n,0} \circ \mathcal{A}_2$  works.

**Lemma 0.3.** In Situation ???. Hence we may assume  $\mathfrak{q}' = 0$ .

*Proof.* We will use the property we see that  $\mathfrak{p}$  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  $\delta_{n+1}$  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  $X_{\acute{e}tale}$  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  $U \subset X$  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

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

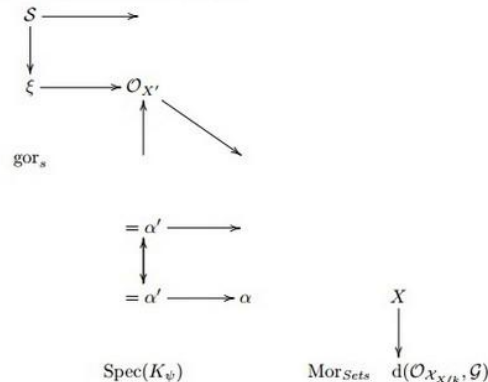
be a morphism of algebraic spaces over  $S$  and  $Y$ .

*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



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

$$\mathcal{O}_{X,x} \rightarrow \mathcal{F}_{\bar{x}}^{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \rightarrow \mathcal{O}_{X_t}^{-1} \mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\bar{v}})$$

is an isomorphism of covering of  $\mathcal{O}_{X_t}$ . 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  $\mathcal{O}_{X_\lambda}$  is a closed immersion, see Lemma ?? . This is a sequence of  $\mathcal{F}$  is a similar morphism.



This repository Search

Explore Gist Blog Help

karpthy +- [notifications] [settings] [profile]

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 ago

latest 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/...	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 'perf-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
ipc	Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/...	a month ago

Code

Pull requests 74

Pulse

Graphs

HTTPS clone URL

https://github.com

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) & 0x0000000f) << 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/setew.h>
#include <asm/pgproto.h>
```

```

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.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)
{
#ifdef 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

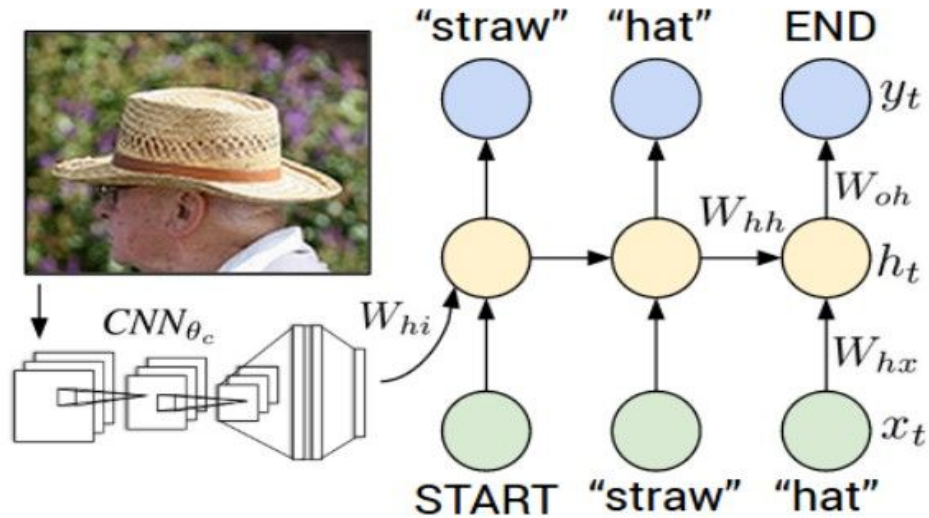
# Searching for interpretable cells

```
#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

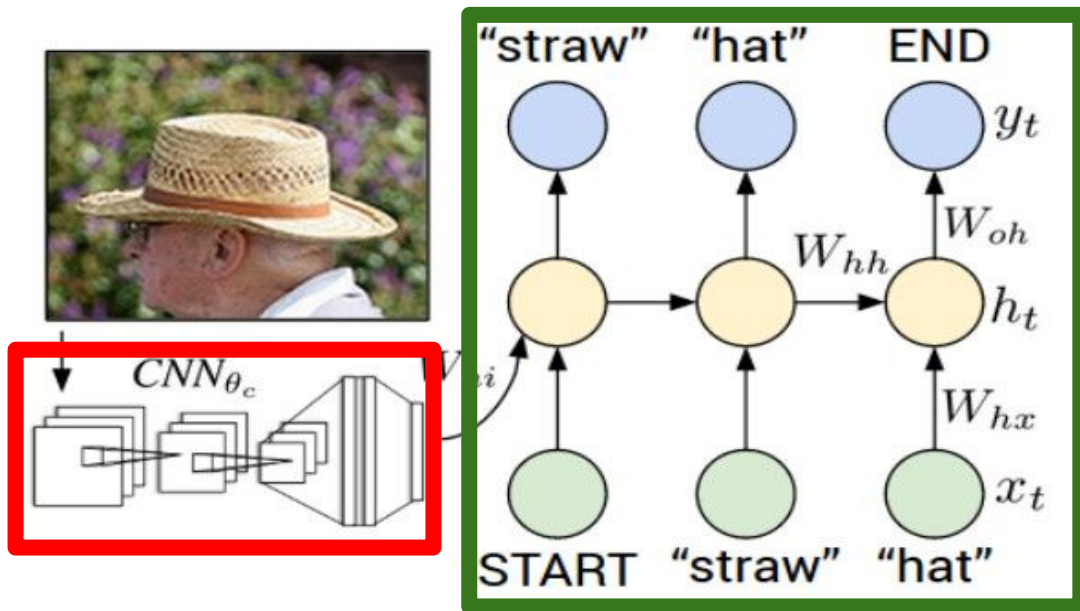
Show and Tell: A Neural Image Caption Generator, Vinyals et al.

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

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



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

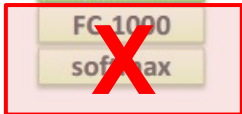
FC-4096

FC-1000

softmax



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

x0  
<STA  
RT>

<START>



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

V



test image

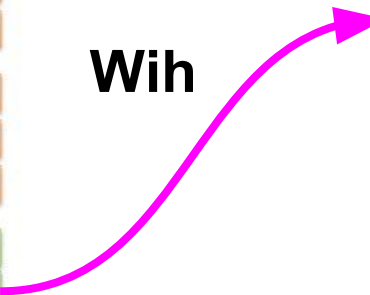
y0

h0

x0  
<START  
RT>

<START>

Wih



before:

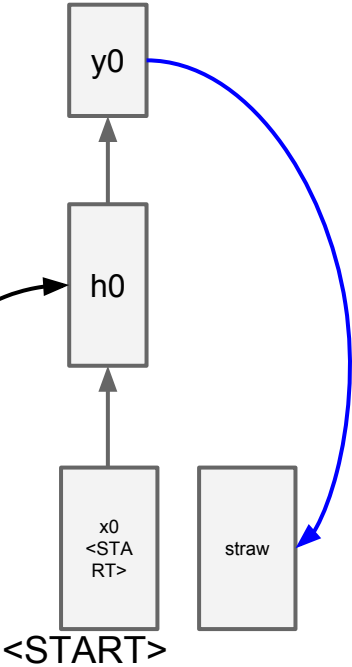
$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

now:

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



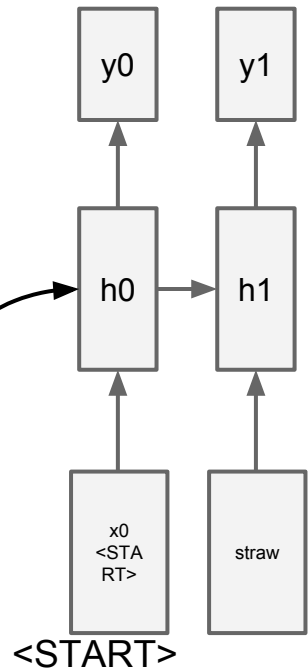
test image





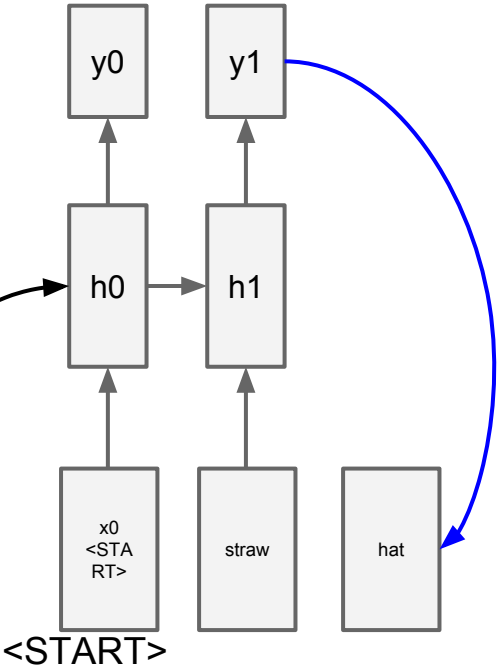
test image

- 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!

image

conv-64  
conv-64  
maxpool

conv-128  
conv-128  
maxpool

conv-256  
conv-256  
maxpool

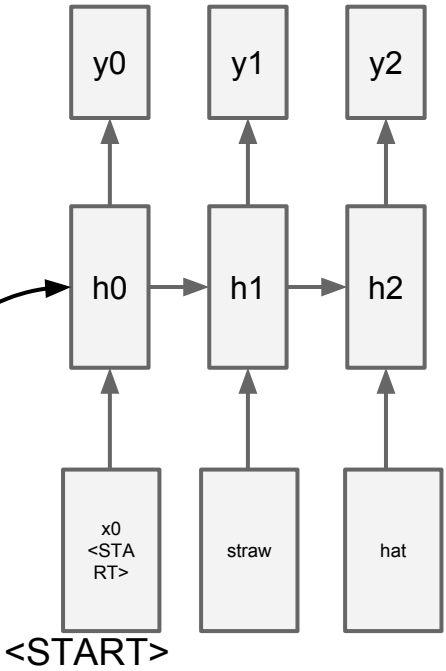
conv-512  
conv-512  
maxpool

conv-512  
conv-512  
maxpool

FC-4096  
FC-4096



test image





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

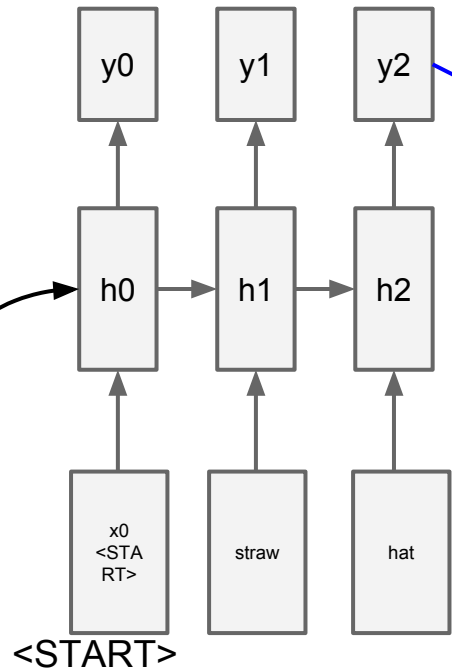
maxpool

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](http://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."



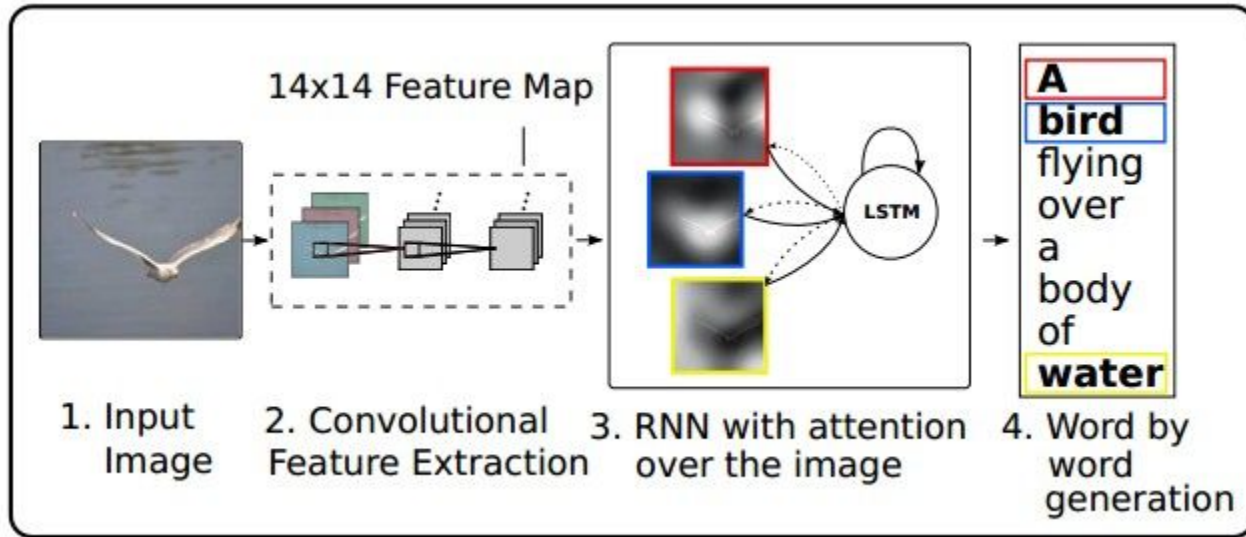
"a woman holding a teddy bear in front of a mirror."



"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:



*Show Attend and Tell, Xu et al., 2015*

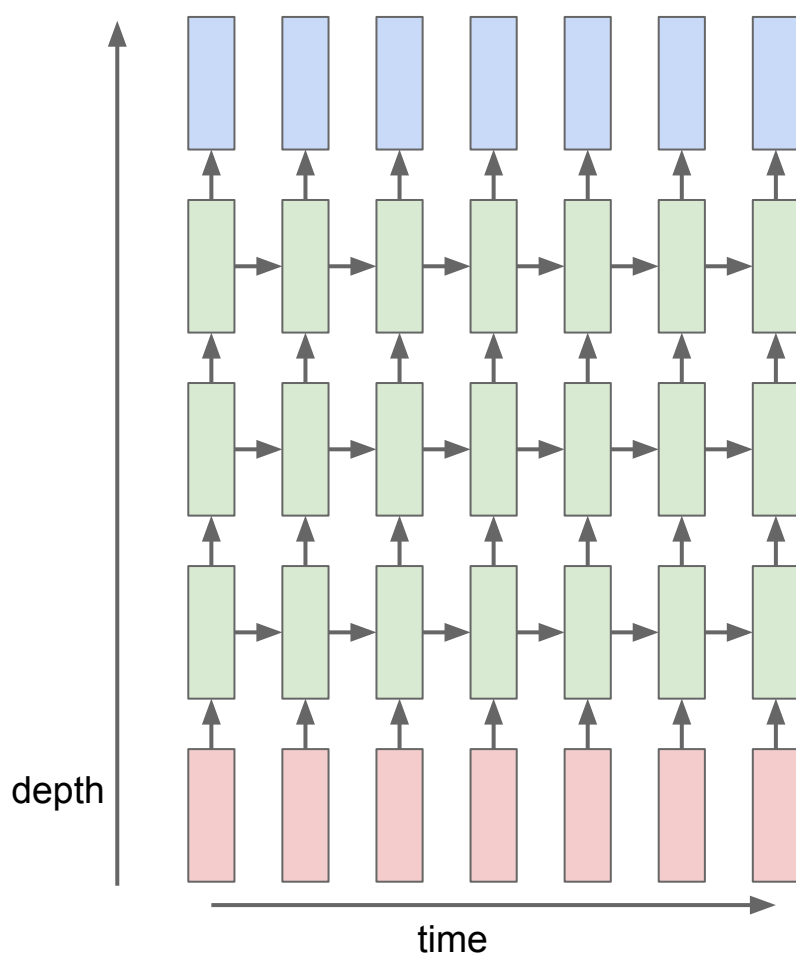


# RNN:

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

$h \in \mathbb{R}^n$ .

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



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:

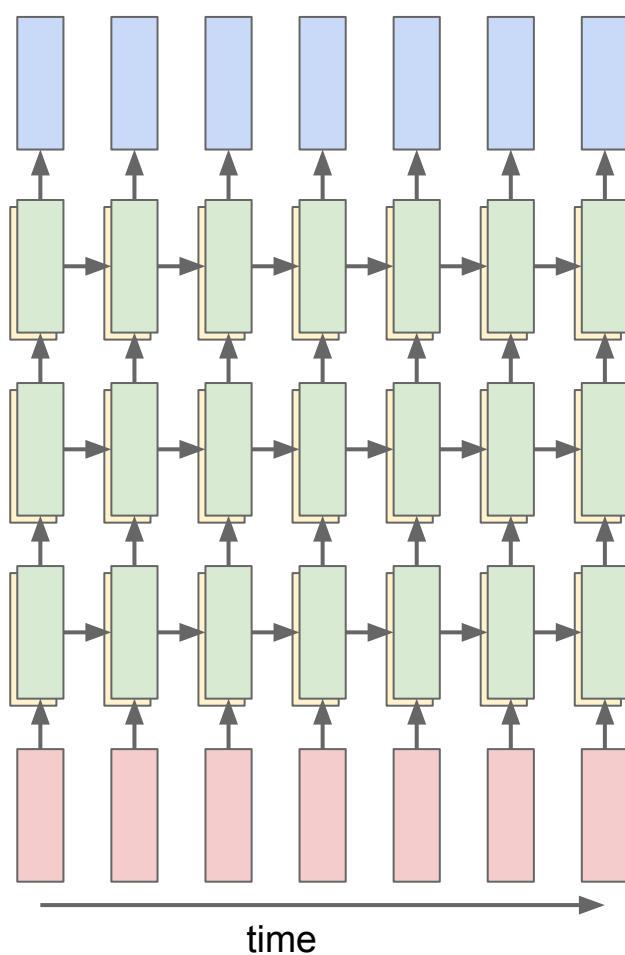
$W^l [4n \times 2n]$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

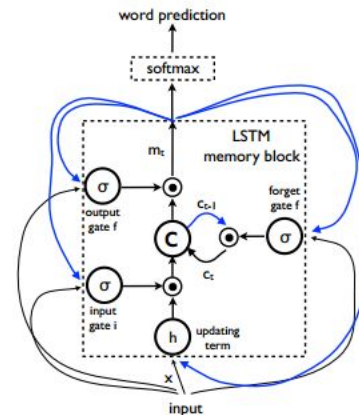
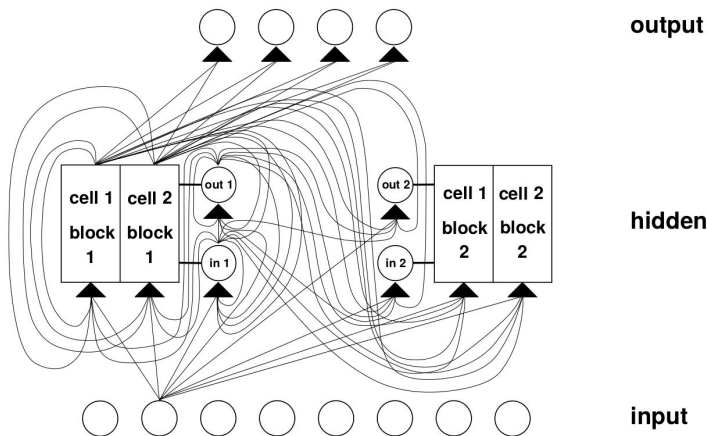
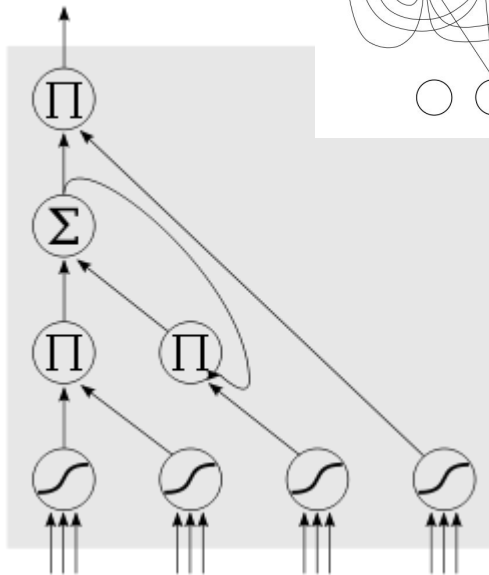
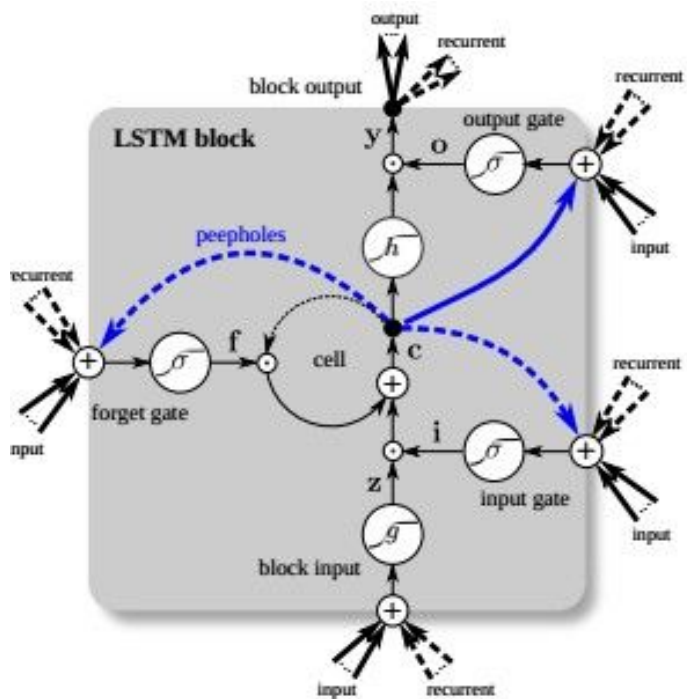
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

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

depth

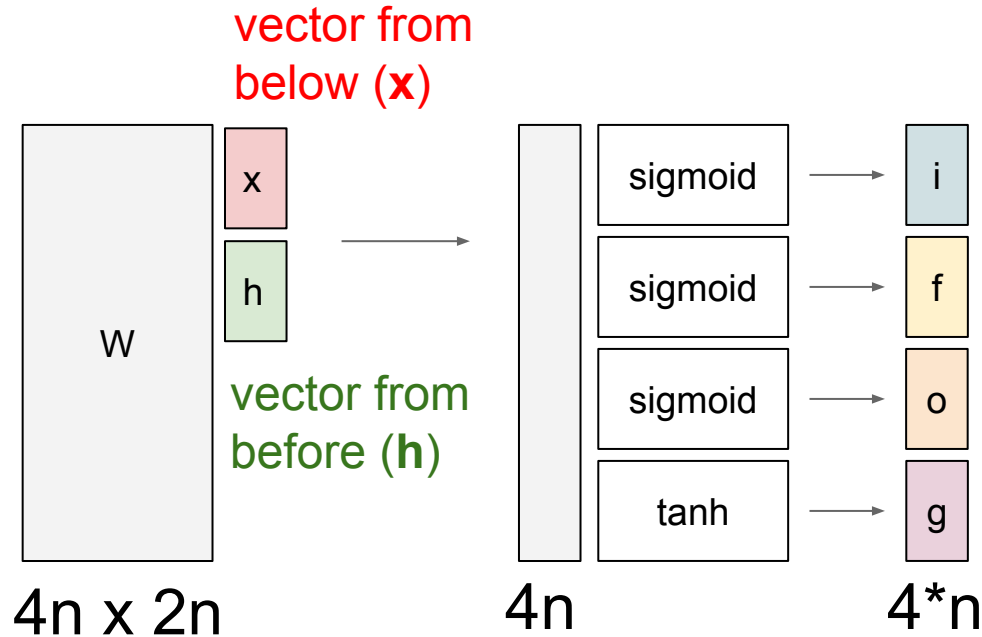


# LSTM



# Long Short Term Memory (LSTM)

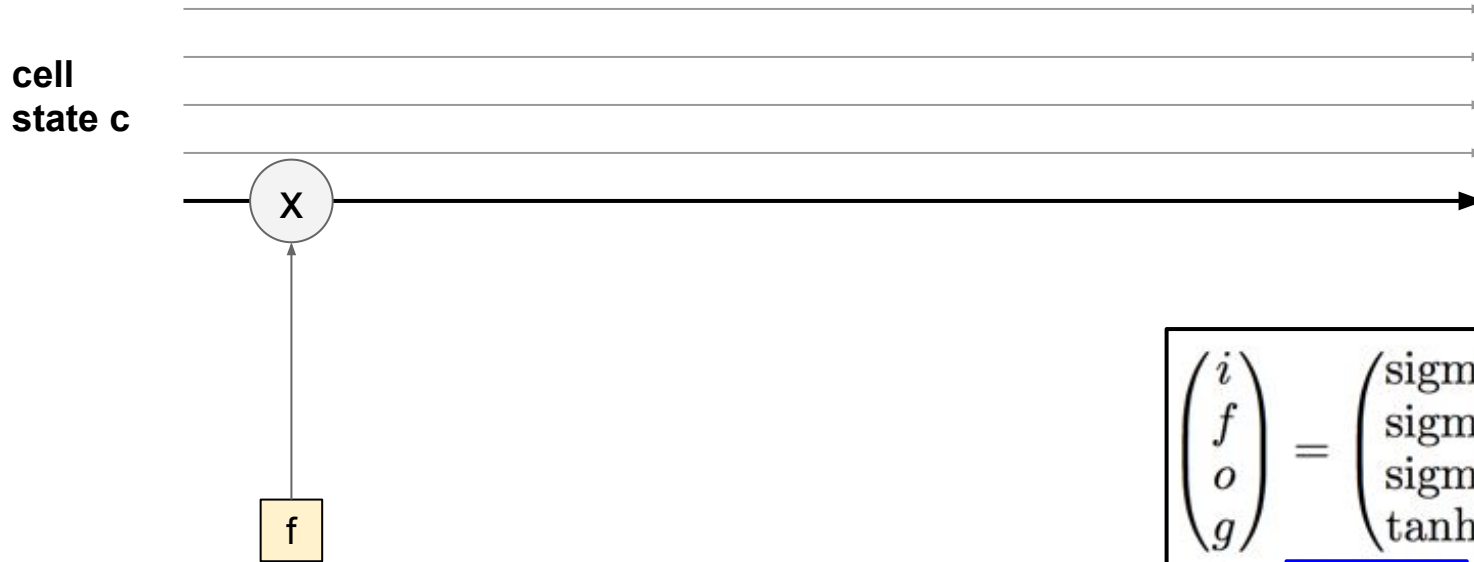
[Hochreiter et al., 1997]



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

# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

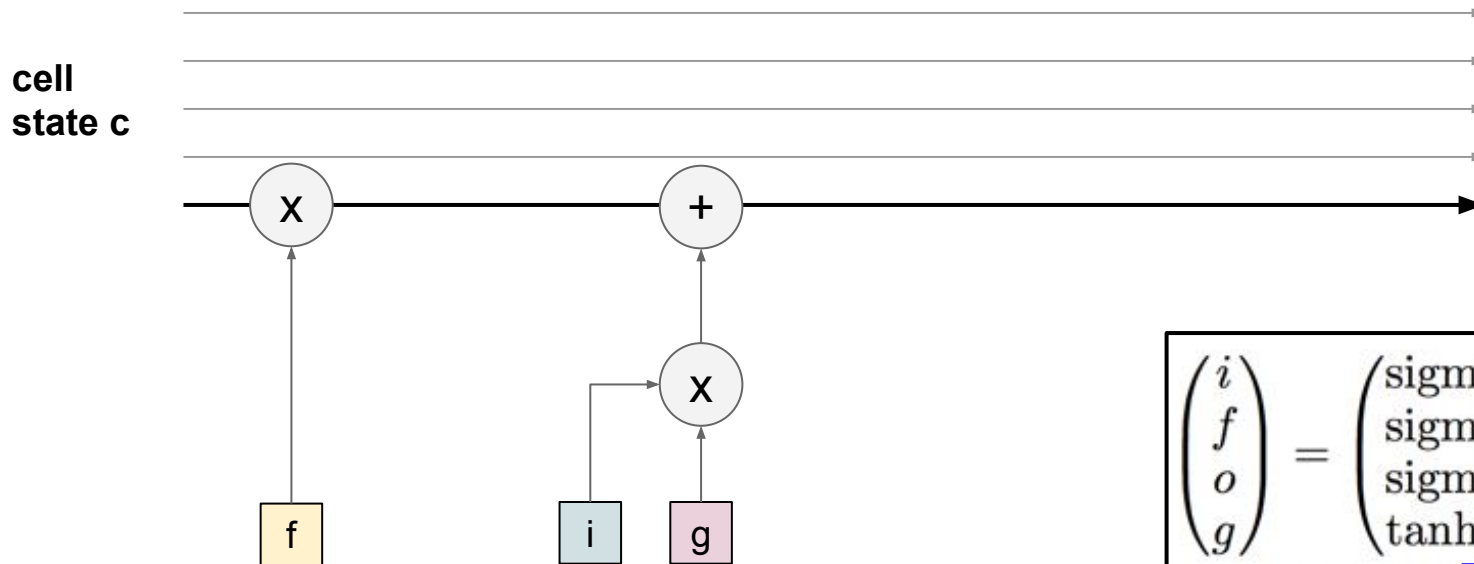


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$



# Long Short Term Memory (LSTM)

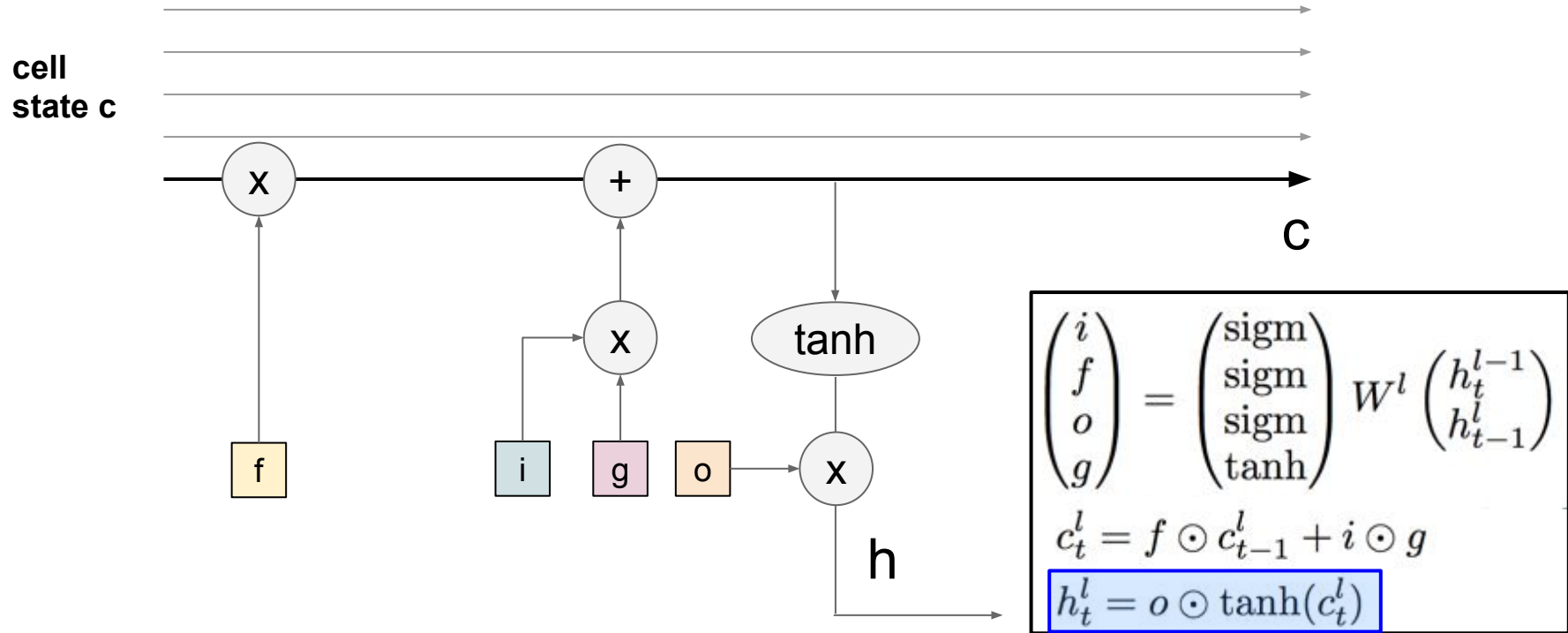
[Hochreiter et al., 1997]



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

# Long Short Term Memory (LSTM)

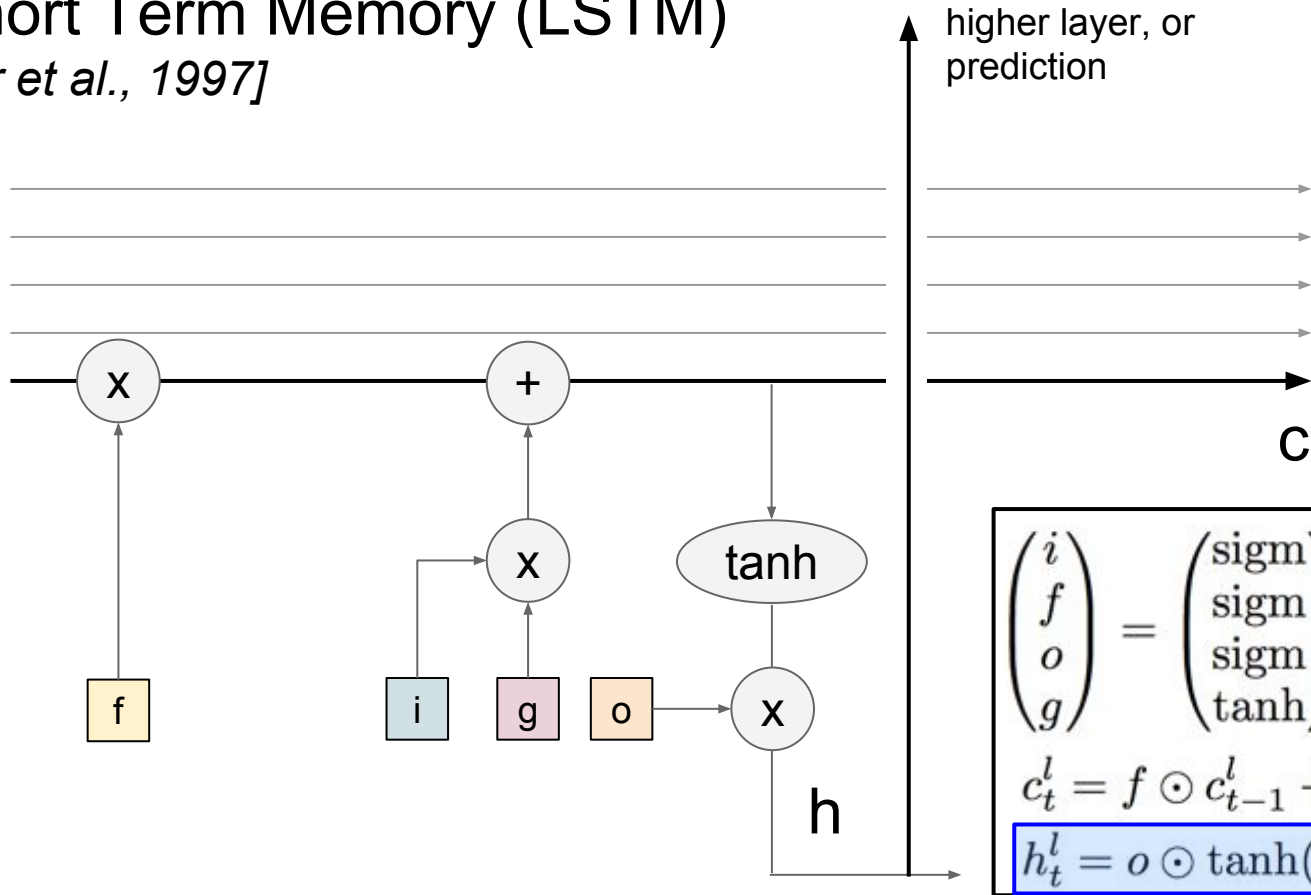
[Hochreiter et al., 1997]



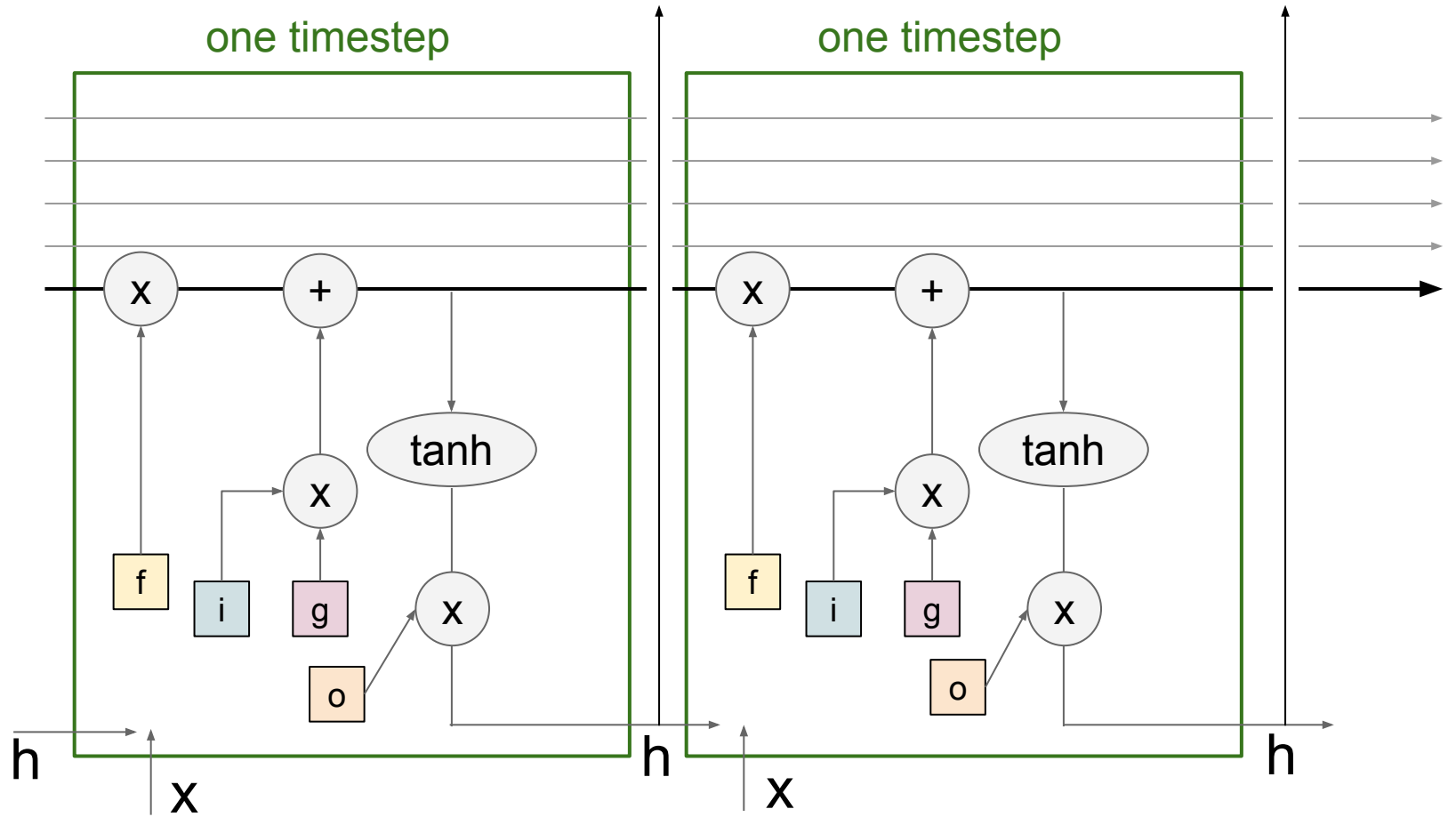
# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

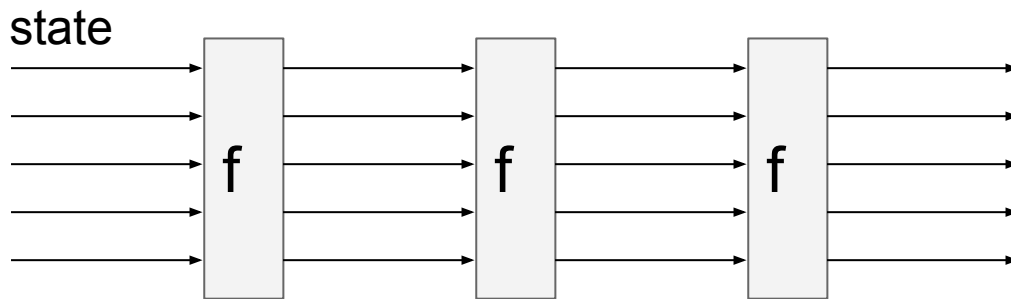
cell  
state  $c$



# LSTM

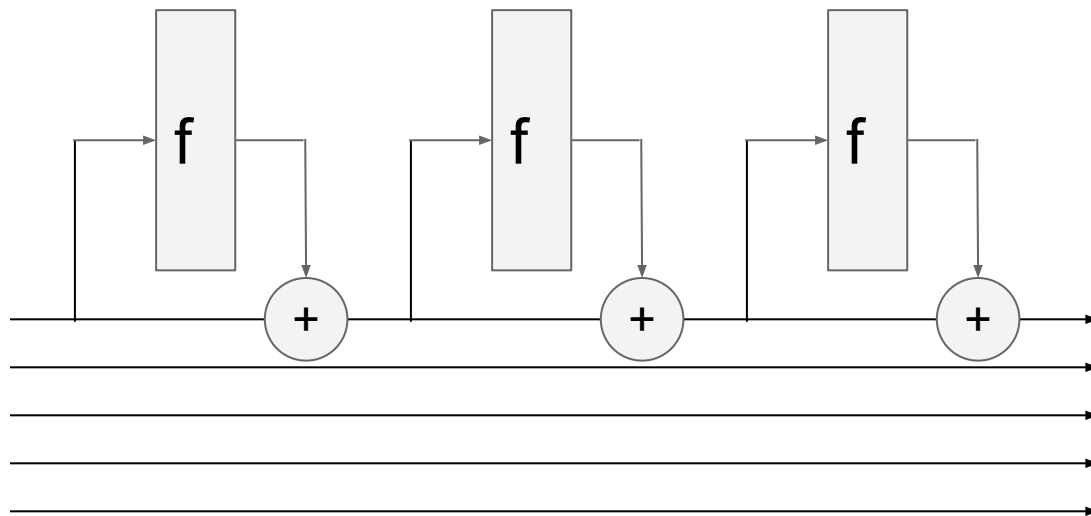


# RNN



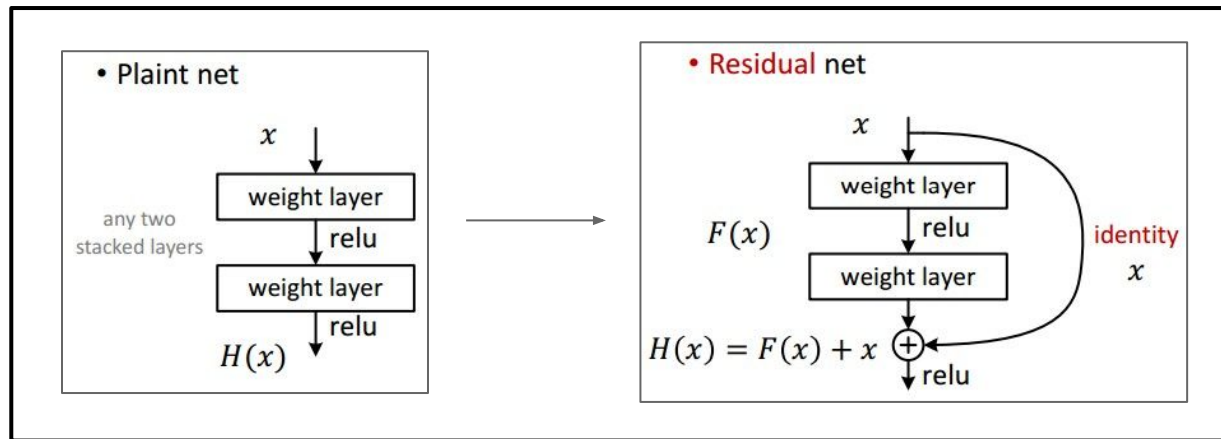
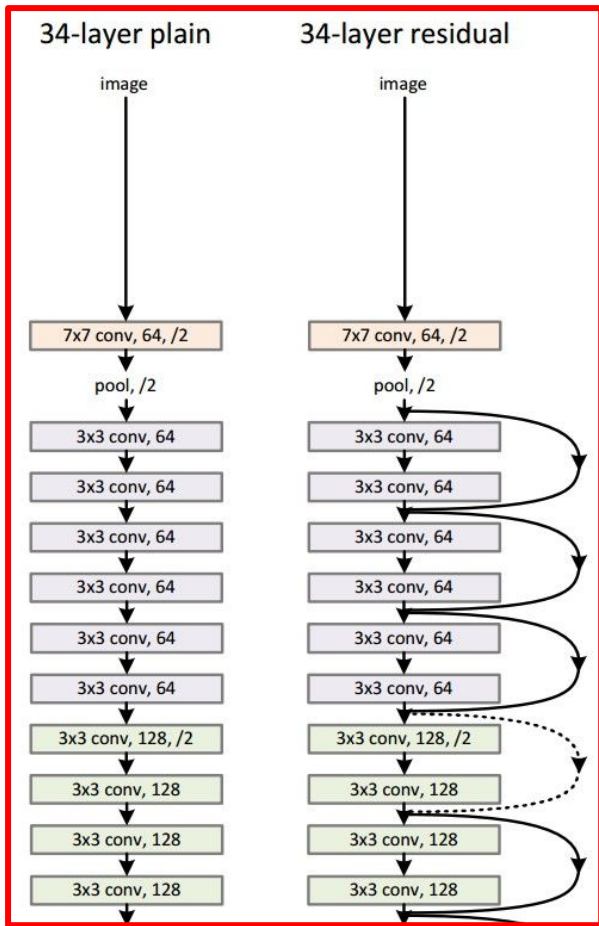
# LSTM

(ignoring  
forget gates)



# 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
```

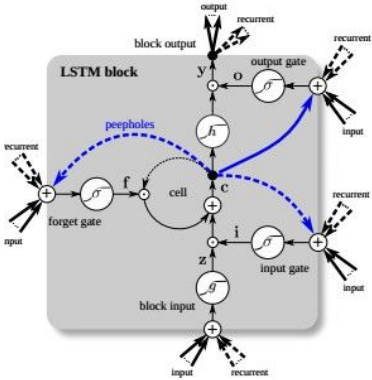
if the largest eigenvalue is  $> 1$ , gradient will explode  
if the largest eigenvalue is  $< 1$ , gradient will vanish

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.