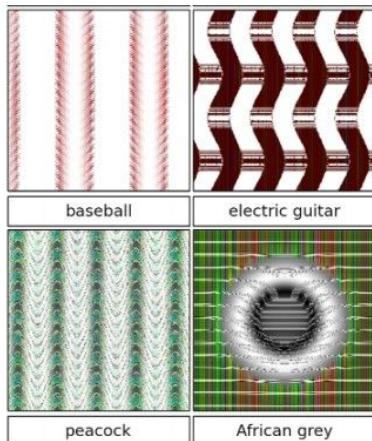
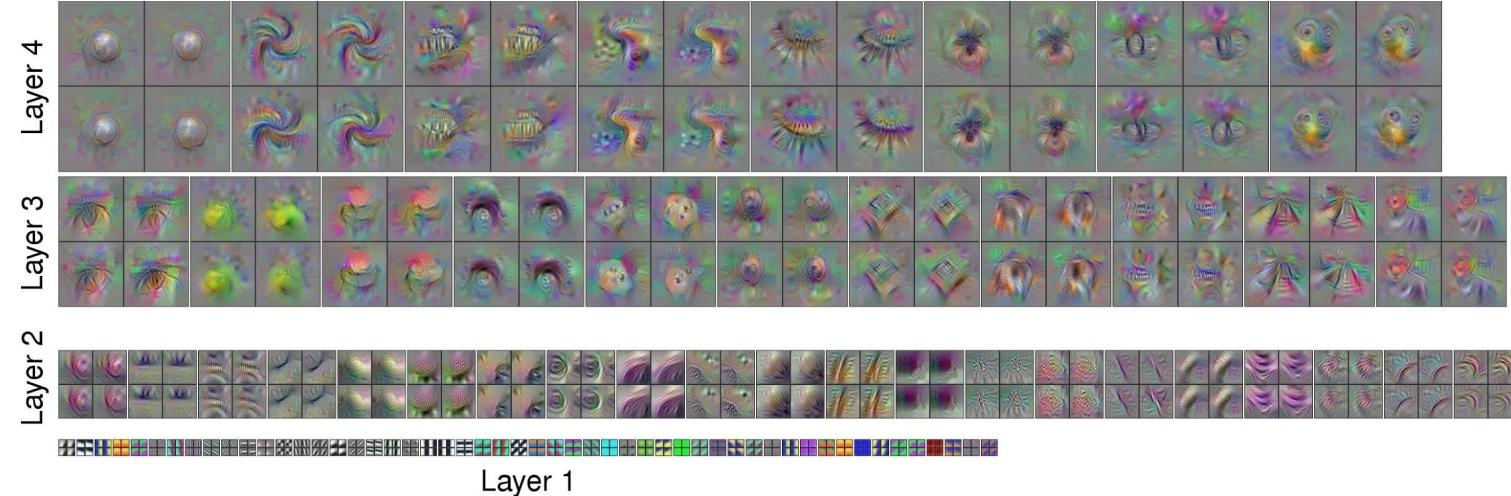
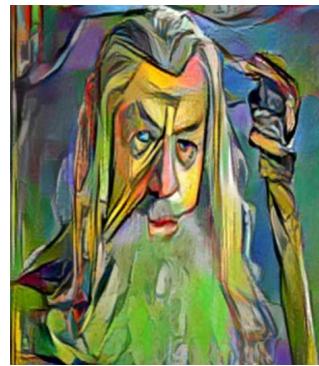
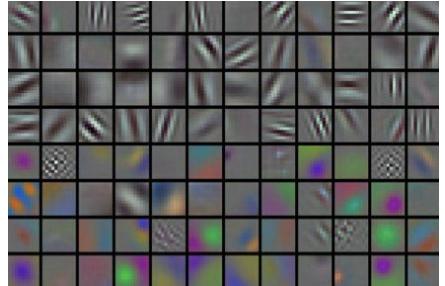


Lecture 10:

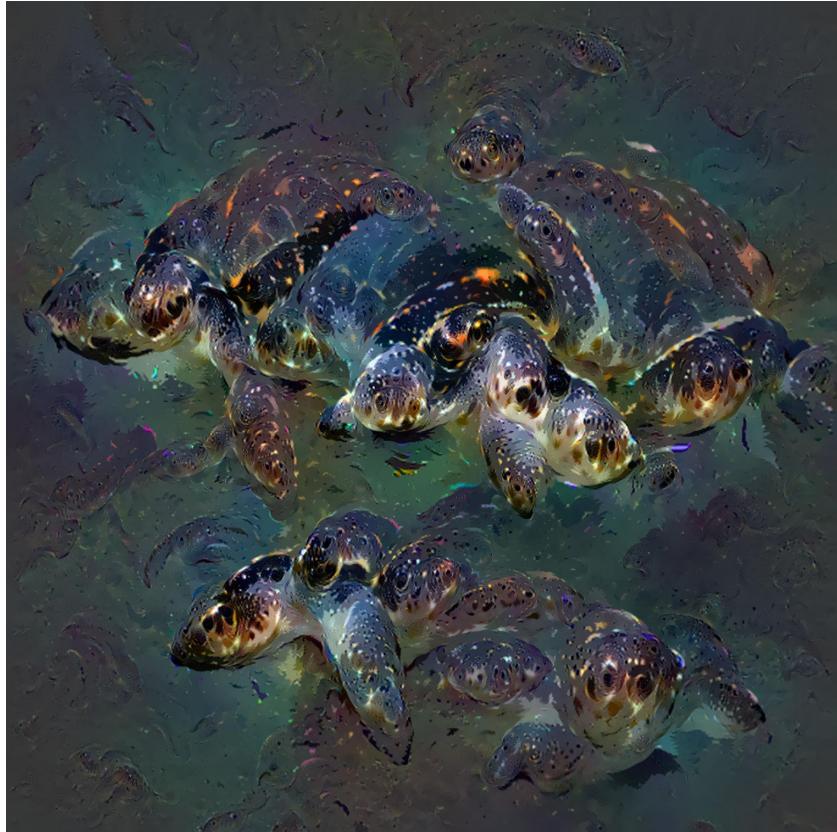
Recurrent Neural Networks

Administrative

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



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

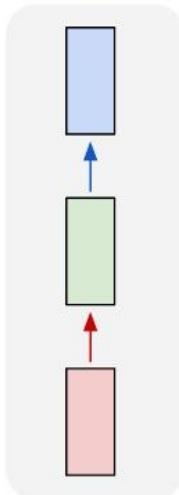


<http://mtyka.github.io/deepdream/2016/02/05/bilateral-class-vis.html>

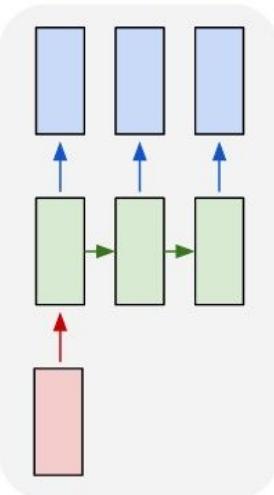


Recurrent Networks offer a lot of flexibility:

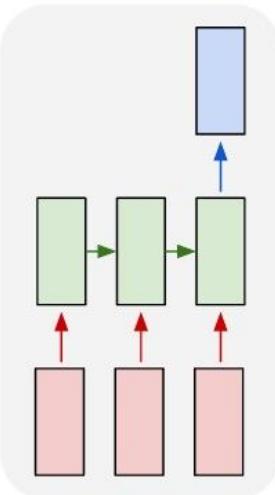
one to one



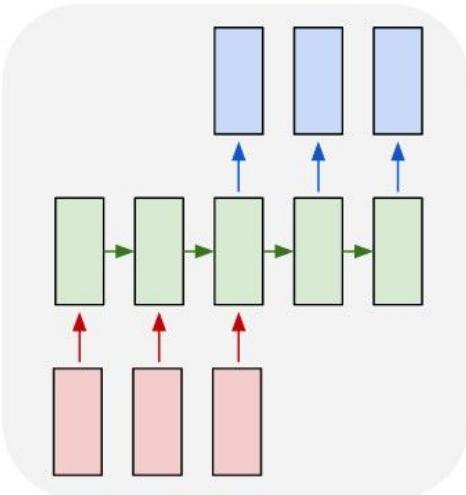
one to many



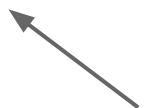
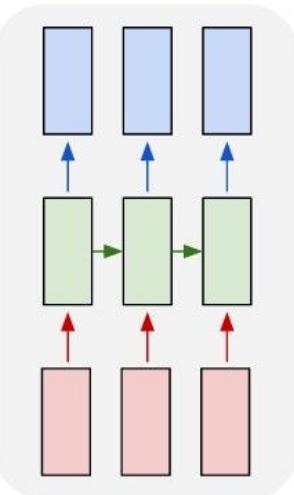
many to one



many to many



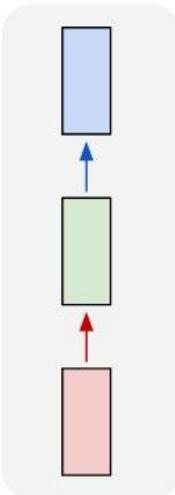
many to many



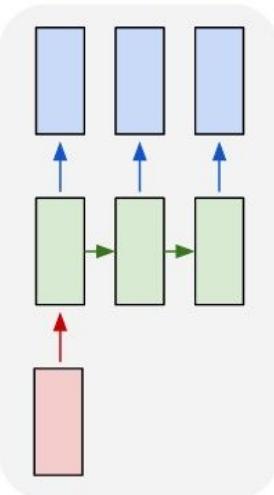
Vanilla Neural Networks

Recurrent Networks offer a lot of flexibility:

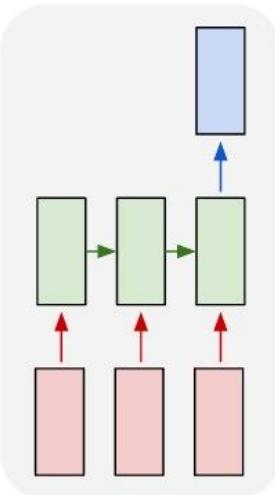
one to one



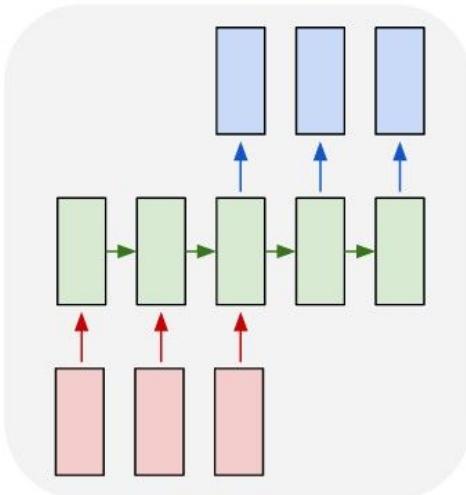
one to many



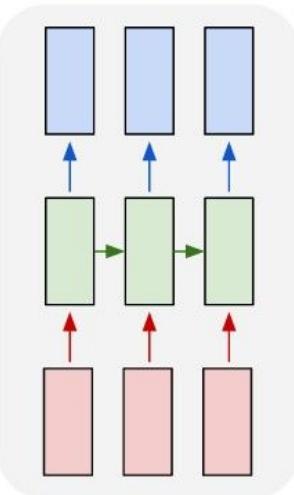
many to one



many to many



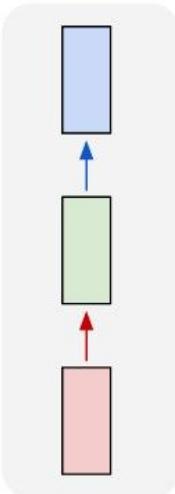
many to many



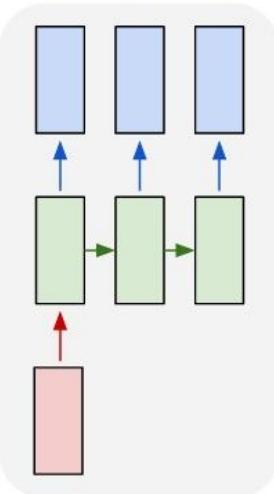
→ e.g. **Image Captioning**
image -> sequence of words

Recurrent Networks offer a lot of flexibility:

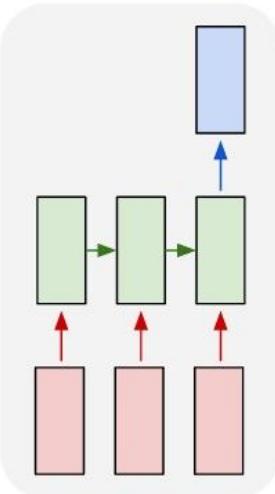
one to one



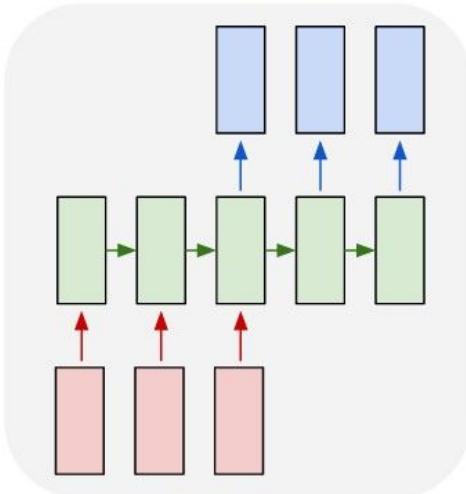
one to many



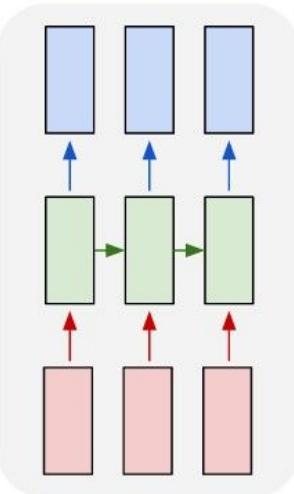
many to one



many to many



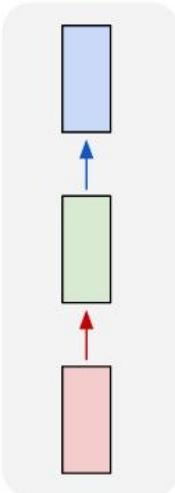
many to many



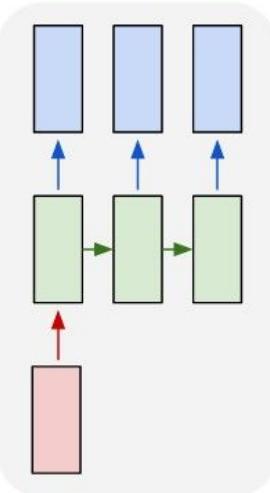
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

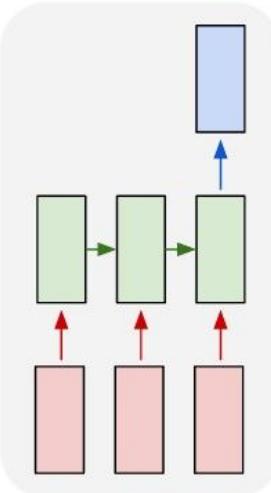
one to one



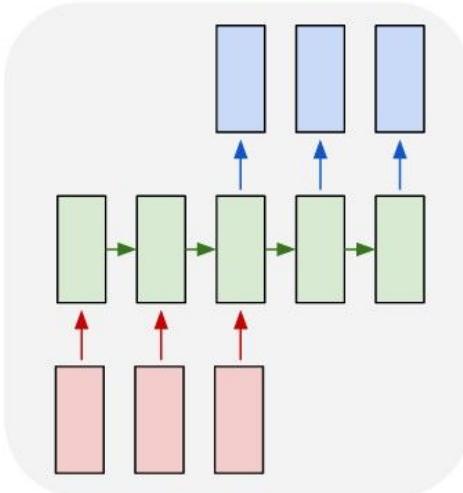
one to many



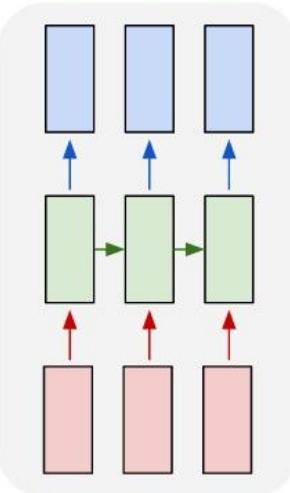
many to one



many to many



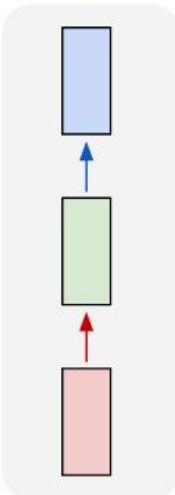
many to many



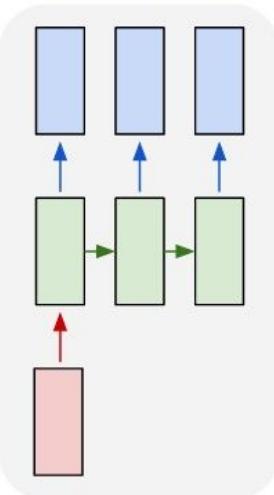
↑
e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Networks offer a lot of flexibility:

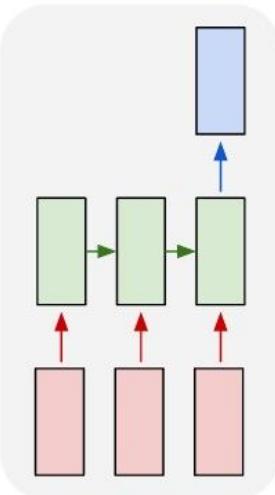
one to one



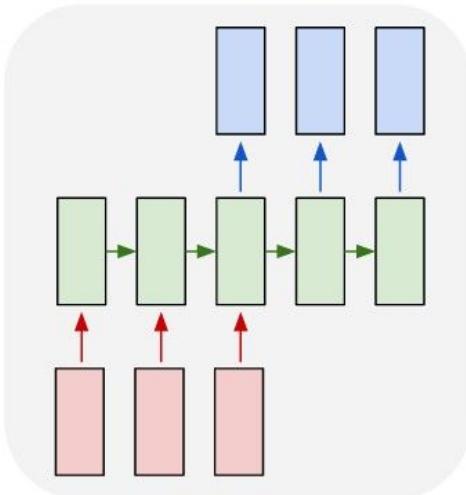
one to many



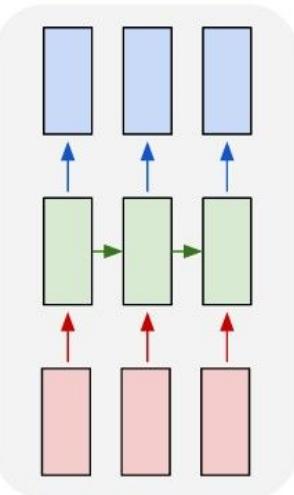
many to one



many to many



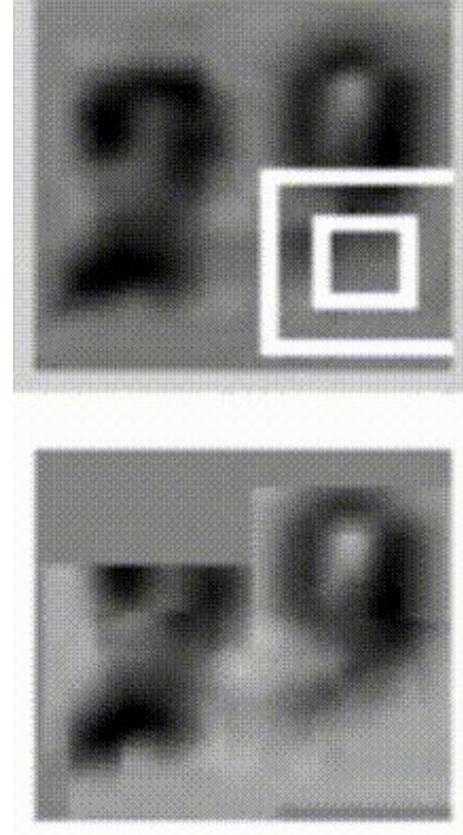
many to many



e.g. Video classification on frame level

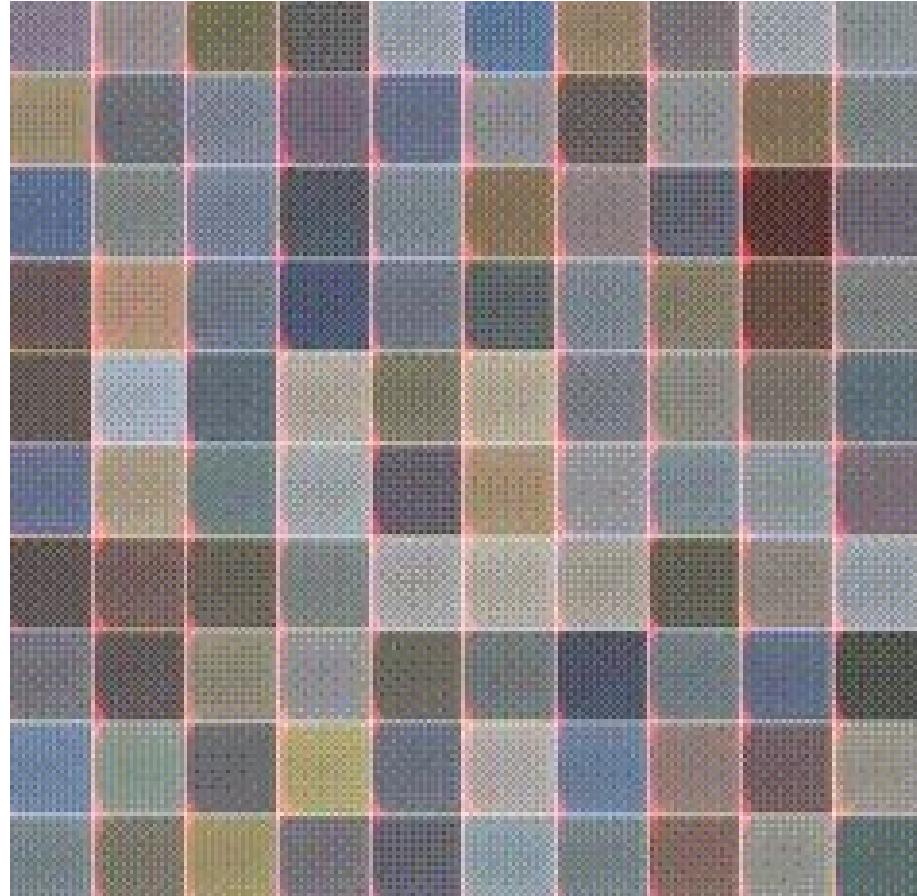
Sequential Processing of fixed inputs

Multiple Object Recognition with
Visual Attention, Ba et al.

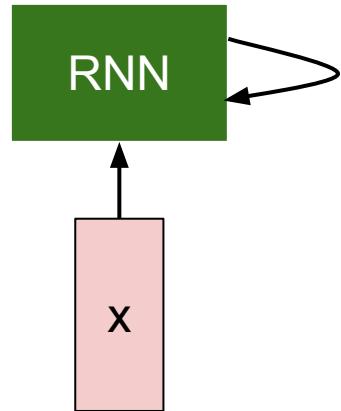


Sequential Processing of fixed outputs

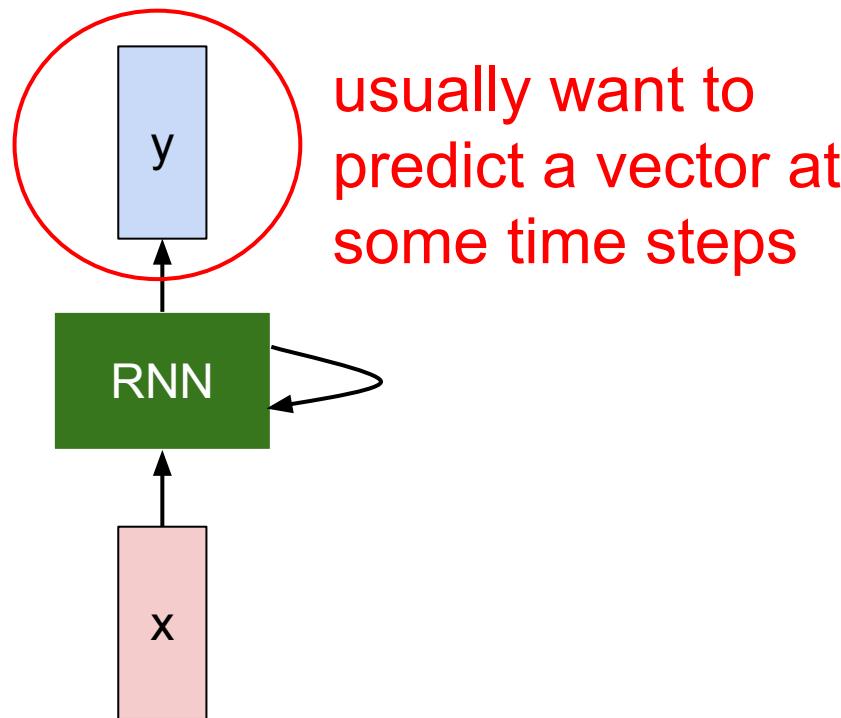
DRAW: A Recurrent
Neural Network For
Image Generation,
Gregor et al.



Recurrent Neural Network



Recurrent Neural Network

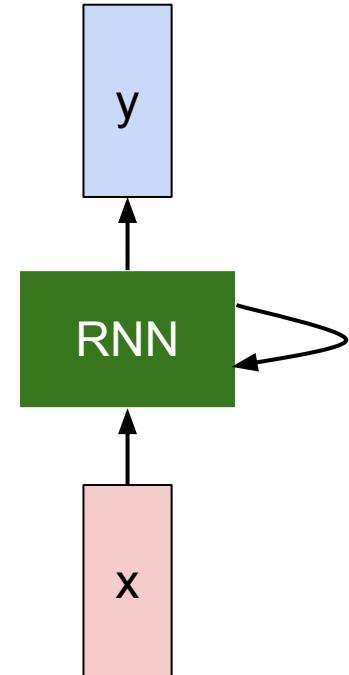


Recurrent Neural Network

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

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

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

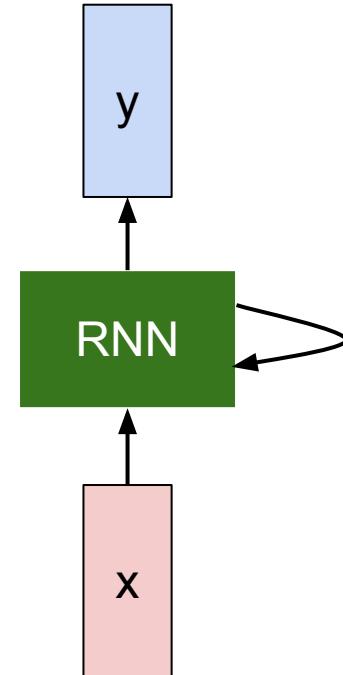


Recurrent Neural Network

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

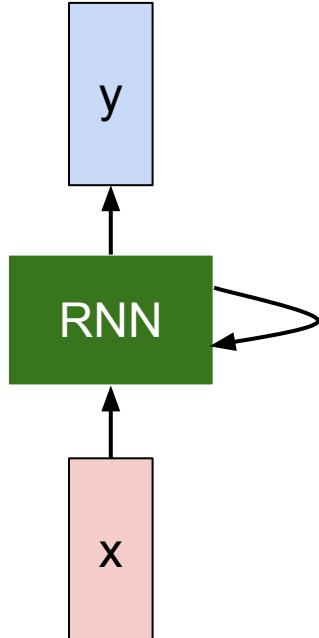
$$h_t = f_W(h_{t-1}, x_t)$$

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



(Vanilla) Recurrent Neural Network

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



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



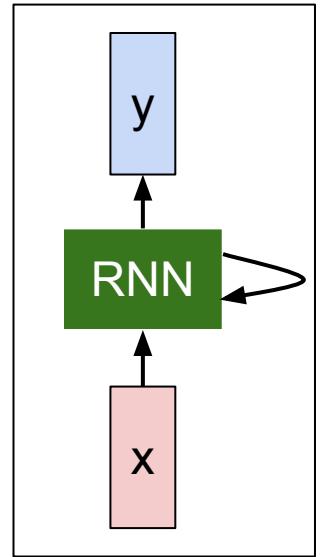
$$\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$$

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

Character-level language model example

Vocabulary:
[h,e,l,o]

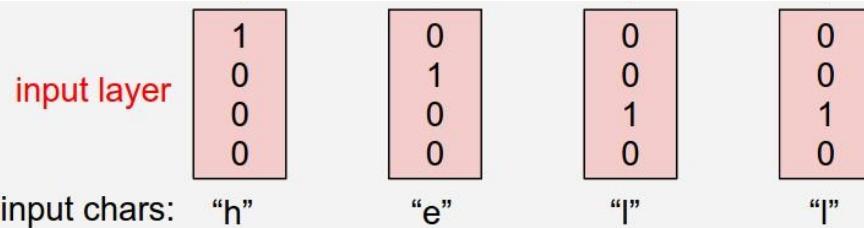
Example training
sequence:
“hello”



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

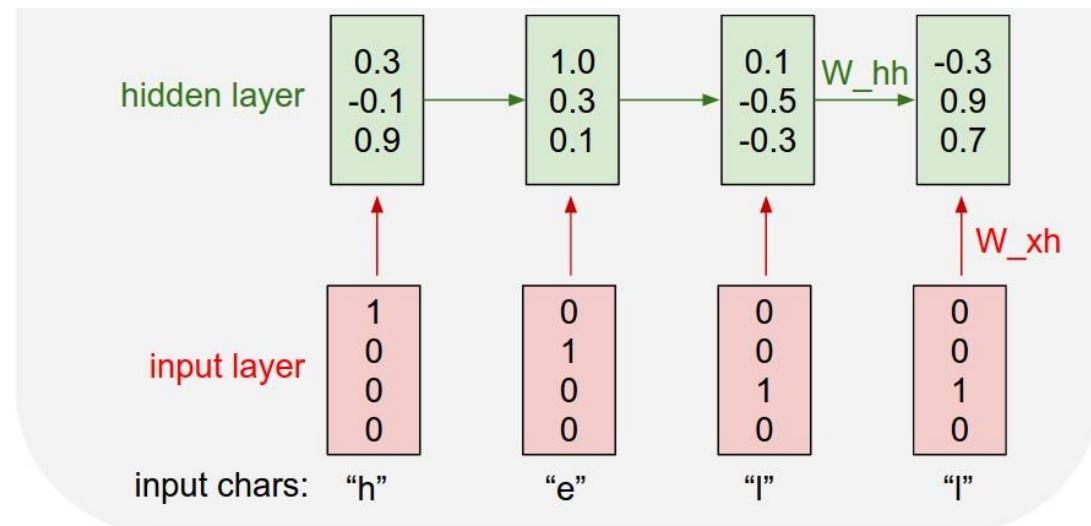


Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

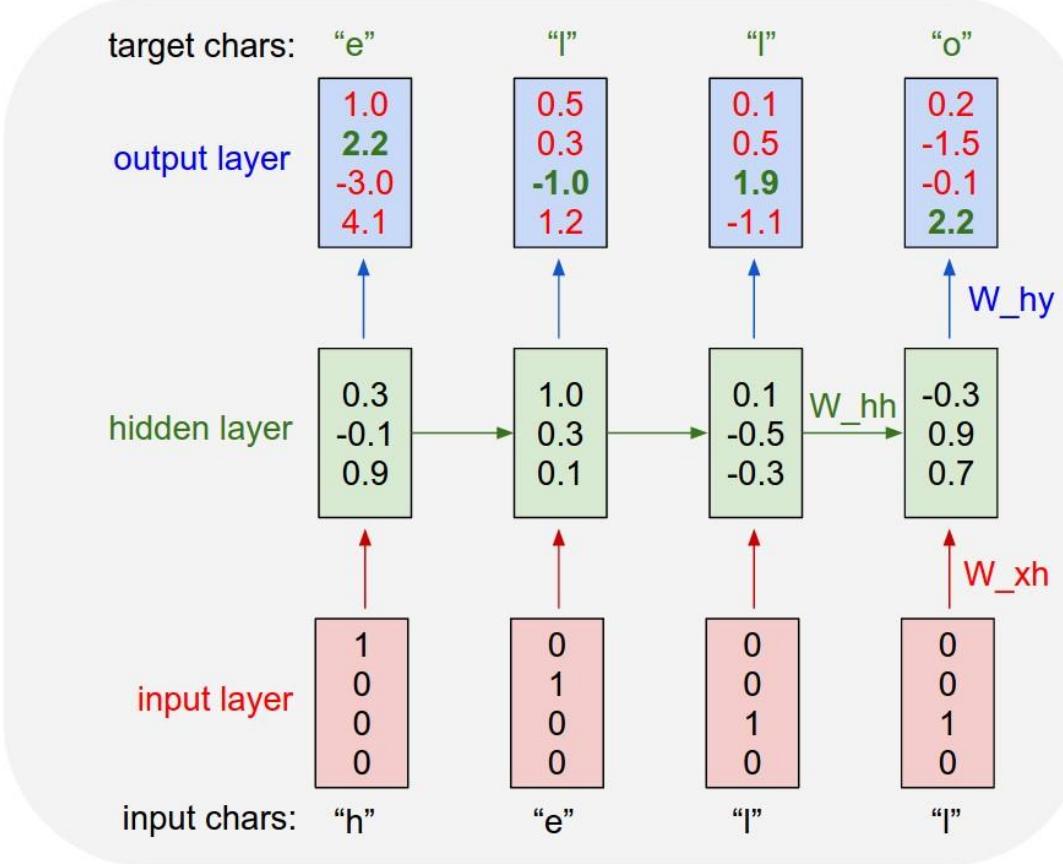
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”



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

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

```
63 def sample(h, seed_ix, n):
64     """
65     sample a sequence of integers from the model
66     h is memory state, seed_ix is seed letter for first time step
67     """
68     x = np.zeros((vocab_size, 1))
69     x[seed_ix] = 1
70     ixes = []
71     for t in xrange(n):
72         h = np.tanh(np.dot(wkh, x) + np.dot(whh, h) + bh)
73         y = np.dot(why, h) + by
74         p = np.exp(y) / np.sum(np.exp(y))
75         ix = np.random.choice(range(vocab_size), p=p.ravel())
76         x = np.zeros((vocab_size, 1))
77         x[ix] = 1
78         ixes.append(ix)
79
80     return ixes
81
82 n, p = 0, 0
83 mxwh, mwhh, mwhy = np.zeros_like(wkh), np.zeros_like(whh), np.zeros_like(why)
84 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
85 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
86 while True:
87     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size,1)) # reset RNN memory
90         p = 0 # go from start of data
91     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print('----\n%s\n----' % (txt, ))
99
100     # forward seq_length characters through the net and fetch gradient
101     loss, dwhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102     smooth_loss = smooth_loss * .999 + loss * .001
103     if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
104
105     # perform parameter update with Adagrad
106     for param, dparam, mem in zip([wkh, whh, why, bh, by],
107                                   [dwhx, dwhh, dwhy, dbh, dby],
108                                   [mxwh, mwhh, mwhy, mbh, mby]):
109         mem += dparam * dparam
110         param -= -learning_rate * param / np.sqrt(mem + 1e-8) # adagrad update
111
112     p += seq_length # move data pointer
113     n += 1 # iteration counter
```

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

min-char-rnn.py gist

```

1  #!/usr/bin/python
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)) # len(chars) = number of unique characters
10  data_size, vocab_size = len(data), len(chars)
11  print 'data has %d characters, %d unique.' % (data_size, vocab_size)
12  char_to_ix = { ch:i for i,ch in enumerate(chars) }
13  ix_to_char = { i:ch for i,ch in enumerate(chars) }

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

18  # Model parameters
19  #w = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
20  #wh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
21  #wy = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
22  #bh = np.zeros(hidden_size, 1) # hidden bias
23  #by = np.zeros(vocab_size, 1) # output bias

24  def lossf(inputs, targets, hprev):
25      # inputs,targets are both lists of integers.
26      # hprev is Hx1 array of initial hidden state
27      # returns the loss, gradients on model parameters, and last hidden state
28      xs, hs, ys, ps = {}, {}, {}, {}
29      hs[-1] = np.copy(hprev)
30      loss = 0.0
31      for t in range(len(inputs)):
32          x = np.zeros((vocab_size, 1)) # encode in 1-of-K representation
33          x[inputs[t], 0] = 1.0
34          wh_t = np.tanh(np.dot(wh, x[t]) + np.dot(wb, hs[t-1]) + bh) # hidden state
35          wy_t = np.tanh(np.dot(wy, wh_t) + np.dot(by, wh_t) + bh) # output state
36          ps[t] = np.exp(wy_t) / np.sum(np.exp(wy_t)) # probabilities for next chars
37          loss += -np.log(ps[t][targets[t]] or 0.01) # softmax (cross-entropy loss)
38          # backprop through tanh nonlinearity
39          dprev_t = np.zeros_like(hs[t-1])
40          dw, db, dhy, dby = np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(wy)
41          dwh, dby = np.zeros_like(wy), np.zeros_like(by)
42          dnext_t = np.zeros_like(hs[t])
43          for i in range(vocab_size):
44              dy = np.zeros(vocab_size)
45              dy[targets[t]] = -1 # backprop into y
46              db += dy * np.sum(dby * np.eye(vocab_size))
47              dw += np.dot(dy * np.eye(vocab_size), x[t].T)
48              dhy += np.dot(dy * np.eye(vocab_size), wh.T)
49              dby += np.sum(dw * np.eye(hidden_size), 1)
50          dw = dw / seq_length
51          dhy = dhy / seq_length
52          dby = dby / seq_length
53          dh = np.dot(wy.T, dy) + dnext_t # backward into h
54          dprev_t = np.tanh(dh) * (1 - dh * dh) # backward through tanh nonlinearity
55          dw += np.dot(dprev_t, x[t].T)
56          dprev_t *= dw # gradient flow into wh
57          dw += np.dot(dprev_t, x[t].T)
58          dprev_t *= dwh # gradient flow into wy
59          dw = dw / seq_length
60          dhy += np.dot(dprev_t, wh.T)
61          dprev_t *= dhy # gradient flow into wy
62          dhy = dhy / seq_length
63          dby += np.sum(dhy * np.eye(hidden_size), 1)
64          dw = dw / seq_length
65          dhy = dhy / seq_length
66          dby = dby / seq_length
67          dby = dby / seq_length
68          dwh += dhy
69          db += dby
70          dw += dhy
71          dhy += dby
72          dby += dhy
73      dw = dw / seq_length
74      dhy = dhy / seq_length
75      dby = dby / seq_length
76      dhy = dhy / seq_length
77      return loss, np.tanh(dw), np.tanh(dhy), np.tanh(dby), np.zeros((hidden_size, 1))

78  # Sample a sequence of integers from the model
79  def sample(h, seed_ix, n):
80      h = np.dot(w, h) + b
81      for t in range(n):
82          x = np.zeros(vocab_size)
83          x[seed_ix] = 1.0
84          h = np.tanh(np.dot(w, x) + np.dot(b, h) + b)
85          p = np.exp(h) / np.sum(np.exp(h))
86          ix = np.random.choice(range(vocab_size), p=p.ravel())
87          seed_ix = ix
88      return ix

89  n, p = 0, 0
90  max_i, min_i = np.zeros(hidden_size), np.zeros(hidden_size), np.zeros(hidden_size)
91  max_o, min_o = np.zeros(vocab_size), np.zeros(vocab_size), np.zeros(vocab_size)
92  smooth_loss = 0.0
93  smooth_loss_n = 0.0
94  while True:
95      if p == seq_length:
96          #print 'resetting memory' #print 'resetting memory'
97          if p+seq_length > len(data):
98              p = 0
99          hprev = np.zeros((hidden_size, 1))
100         #print 'resetting memory' #print 'resetting memory'
101         #print 'resetting memory' #print 'resetting memory'
102      inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
103      targets = [char_to_ix[ch] for ch in data[p+seq_length:]]
104
105      a = np.tanh(dw, now=True)
106      b = np.tanh(dhy, now=True)
107      c = np.tanh(dby, now=True)
108
109      for i in range(seq_length):
110          if p+seq_length+i > len(data):
111              p = 0
112          hprev = np.zeros((hidden_size, 1))
113          inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
114          targets = [char_to_ix[ch] for ch in data[p+seq_length:]]
115
116          # forward pass: compute pre-activations through the net and fetch gradient
117          loss, dh, dw, dbh, dhy, dby, hprev = lossf(inputs, targets, hprev)
118
119          smooth_loss = smooth_loss * 0.999 + loss * 0.001
120
121          if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
122
123      # backpropagation step
124      for parname, param in zip(['wh', 'wy', 'wb', 'by'],
125                               [wh, wy, wb, by],
126                               [dwh, dhy, dby, dby],
127                               [dbh, dbh, dbh, dbh]):
128          parname += 'grad' # gradient
129          parname += 'param' # parameter
130          param += 'param' # parameter
131          np.clip(dw, -5, 5, out=parname) # clip to mitigate exploding gradients
132          np.clip(dhy, -5, 5, out=parname)
133          np.clip(dby, -5, 5, out=parname)
134          np.clip(dbh, -5, 5, out=parname)
135
136      dw = dw / seq_length
137      dhy = dhy / seq_length
138      dby = dby / seq_length
139      dbh = dbh / seq_length
140
141      # update parameters via Adam
142      for parname, param in zip(['wh', 'wy', 'wb', 'by'],
143                               [wh, wy, wb, by],
144                               [dwh, dhy, dby, dby],
145                               [dbh, dbh, dbh, dbh]):
146          parname += 'param'
147          param += 'param'
148
149      p += seq_length # move data pointer
150
151      n += 1 # iteration counter

```

Data I/O

```

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

13
14  """
15  Mininal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
16  BSD License
17  """
18
19
20  import numpy as np
21
22  # data I/O
23  data = open('input.txt', 'r').read() # should be simple plain text file
24  chars = list(set(data))
25  data_size, vocab_size = len(data), len(chars)
26  print 'data has %d characters, %d unique.' % (data_size, vocab_size)
27  char_to_ix = { ch:i for i,ch in enumerate(chars) }
28  ix_to_char = { i:ch for i,ch in enumerate(chars) }

29
30
31

```

min-char-rnn.py gist

```

1 /**
2  * Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  * BSD License
4  */
5
6 import numpy as np
7
8 # Data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print('data has %d characters.' % data_size)
13 char_to_ix = {ch:i for i, ch in enumerate(chars)}
14 ix_to_char = {i:ch for ch in enumerate(chars)}
15
16 n_h = 100
17 hidden_size = n_h + size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 whh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 who = 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     """ inputs,targets are both lists of integers.
29         hprev is RNN array of initial hidden state
30         returns the loss, gradients on model parameters, and last hidden state
31     """
32     xs, hs, ys, ps = O, O, O, O
33     hprev = np.copy(hprev)
34     loss = 0
35
36     for t in range(len(inputs)):
37         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
38         x[inputs[t], 1] = 1
39         h1t1 = np.tanh(np.dot(whh, x1t1) + np.dot(who, hs[-1]) + bh) # hidden state
40         y1t1 = np.dot(why, h1t1) / np.sum(np.exp(y1t1)) = softmax (cross-entropy loss)
41         loss += -np.exp(y1t1) * np.log(inputs[t])
42         dx1t1 = np.exp(y1t1) / np.sum(np.exp(y1t1)) = probabilities for next chars
43         loss -= np.exp(y1t1) * np.log(inputs[t])
44         dy1t1 = np.zeros_like(y1t1) # backprop into y
45         dh1t1 = np.zeros_like(h1t1) # backprop into h
46         dwhh = np.zeros_like(whh) # backprop into whh
47         dwho = np.zeros_like(who) # backprop into who
48         dwhy = np.zeros_like(why) # backprop into why
49         dbh = np.zeros_like(bh) # backprop into bh
50         dby = np.zeros_like(by) # backprop into by
51
52         dnext = np.zeros_like(h1t1)
53         for l in range(len(inputs)-t-1):
54             dy1t1 = np.dot(why, dnext) # backprop into y
55             dh1t1 = np.dot(who, dy1t1) # backprop through tanh nonlinearity
56             dh1t1 += dh1t1 # backprop through tanh nonlinearity
57             dwhh += np.dot(dy1t1, x1t1.T) # backprop into whh
58             dwho += np.dot(dy1t1, hs[-1].T) # backprop into who
59             dwhy += np.dot(dy1t1, dh1t1.T) # backprop into why
60             dbh += dy1t1 # backprop into bh
61             dby += dy1t1 # backprop into by
62
63             for opname in [dwhh, dwho, dbh, dby]:
64                 np.clip(opname, -5, 5, out=opname) = clip to mitigate exploding gradients
65             if np.isnan(dwhh) or np.isnan(dwho) or np.isnan(dbh) or np.isnan(dby):
66                 raise ValueError("NaN in gradient computation for %s!" % opname)
67             else:
68                 pass
69
70         if t == 0:
71             h1 = np.zeros(hidden_size, 1)
72             h1 = np.tanh(np.dot(whh, x) + np.dot(who, h) + bh)
73             h1 = np.zeros(hidden_size, 1)
74         else:
75             h1 = np.tanh(np.dot(whh, x) + np.dot(who, h) + bh)
76             h1 = np.zeros(hidden_size, 1)
77
78         x[seed_ix, 1] = 1
79
80        for t in range(p+seqLength):
81            h = np.tanh(np.dot(whh, x) + np.dot(who, h) + bh)
82            p = np.argmax(h) # highest probability
83            x = np.zeros(vocab_size, 1)
84            x[p, 1] = 1
85            h = np.zeros(hidden_size, 1)
86        x[seed_ix, 1] = 1
87
88    return loss
89
90 n, p = 0
91
92 mean, std = np.mean(data), np.std(data)
93 m, s = np.zeros(vocab_size), np.zeros(vocab_size)
94
95 for i in range(len(data)):
96     m[i] = mean*(m[i] + (data[i] - mean)*(1 - s[i])) / seq_length
97     s[i] = std*(s[i] + ((data[i] - mean)**2)*(1 - s[i])/seq_length)
98
99 if p+seqLength > len(data) or p == 0:
100     hprev = np.zeros(hidden_size)
101
102 inputs = [char_to_ix[ch] for ch in data[p:p+seqLength]]
103 targets = [char_to_ix[ch] for ch in data[p+seqLength-1:p+seqLength]]
104
105 n += 1
106
107 # Clip from the model, now that we have gradients
108 if n < 100:
109     sample_ix = sample(hprev, inputs[0], 200)
110     sample_ix = jacobian_to_charsample[1] for ix in sample_ix]
111     print("Sampling from first sequence")
112
113 # Forward pass, unrolled characters through the net and fetch gradient
114 loss, dh1, dwhh, dwho, dbh, dby, hprev = lossfun(inputs, targets, hprev)
115 smooth_loss = smooth_loss * 0.999 + loss * 0.001
116
117 if n % 100 == 0:
118     print('iter %d, loss: %f, smooth loss: %f' % (n, loss, smooth_loss))
119
120 # Compute gradients, unrolled characters through the net and fetch gradient
121 for params, dparams, name in zip([whh, who, why, bh, by],
122                                 [dwhh, dwho, dwhy, dbh, dby],
123                                 [w_hh, w_who, w_why, w_bh, w_by]):
124     dparams = np.dot(dy1t1, h1t1.T)
125     np.einsum('...jk,...ki->...ji', dparams, x1t1, out=dparams)
126
127     eee += dparams * dparams
128     dparams /= learning_rate
129     dparams = np.sqrt(eee + 1e-8) = adapted update
130
131 p = seqLength % data pointer
132 n = 1 + iteration counter

```

15

```

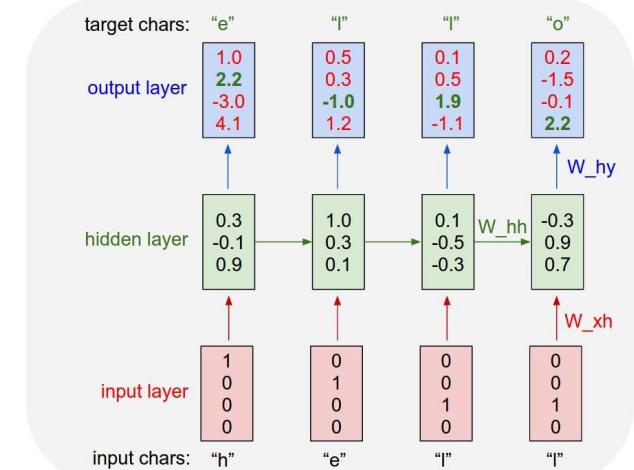
# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros(hidden_size, 1) # hidden bias
by = np.zeros(vocab_size, 1) # output bias

```

Initializations

recall:



min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size = len(data)
12 print("data has %d characters, %d unique." % (data_size, vocab_size))
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # dict to map from index to character
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def init_randomhidden(state, vocab_size):
24     wh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
25     bh = np.random.rand(hidden_size, vocab_size)*0.01 # hidden to hidden
26     why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
27     by = np.zeros((vocab_size, 1)) # hidden bias
28     bi = np.zeros((vocab_size, 1)) # output bias
29
30     return wh, bh, why, by, bi
31
32 def lossFun(inputs, targets, hprev):
33
34     inputs,targets = both lists of integers.
35
36     hprev is Hx1 array of initial hidden state
37     returns the loss, gradients on model parameters, and last hidden state
38
39     xs, hs, ys, ps = O, O, O, O
40     h0 = -1 * np.copy(hprev)
41
42     for t in xrange(seq_length):
43         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
44         x[inputs[t]:] = 1 # x[t] = 1; all other values 0
45
46         h1 = 1 / np.tanh(np.dot(wh, x) + np.dot(bh, hs[-1]) + bh) # hidden state
47         ps = np.exp(h1) / np.sum(np.exp(h1)) # probabilities for next chars
48
49         loss += -np.log(ps[targets[t]:][0]) # softmax (cross-entropy loss)
50
51         # backprop through tanh nonlinearity
52         dprev, dh0, dwh, dbh, dwhy, dby = np.zeros_like(hprev), np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(why)
53         dby = np.zeros_like(by)
54         dnext = np.zeros_like(h1)
55
56         for i in range(len(inputs)):
57             dy = np.copy(ps[i])
58             dy[targets[t]] -= 1 # backprop into y
59             dh = np.dot(dy, wh) # backprop into h
60             dwh += np.dot(dy.T, h0) # dh * backprop through tanh nonlinearity
61             dbh += np.dot(dy, h0)
62             dprev = np.dot(dy, wh.T) # backprop into previous h
63             dnext = np.dot(dy, wh) # backprop into next h
64
65             for dparam in [dwh, dbh, dwhy, dby]:
66                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
67             if dparam != 0:
68                 dparam *= 0.01 # gradient clipping
69
70         if t > 0:
71             h0 = h1 # update previous hidden state
72
73     smooth_loss = smooth_loss * 0.999 + loss * 0.001
74
75     return loss, smooth_loss
76
77
78 # sample a sequence of integers from the model
79 h0 = memory_state # seed h0
80 h1 = memory_state # seed h1 is used later for first time step
81
82 x = np.zeros((vocab_size, 1))
83 x[seed_ix] = 1
84
85 for t in xrange(n):
86     h = np.tanh(np.dot(wh, x) + np.dot(bh, h0) + bh)
87     p = np.exp(h) / np.sum(np.exp(h))
88     ix = np.random.choice(range(vocab_size), p=p.ravel())
89     x[0] = np.zeros((vocab_size, 1))
89     x[1] = np.zeros((vocab_size, 1))
90
91     return ix
92
93
94 n, p = 0, 0
95 mem, mem0, mwhy = np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(why)
96 mem0, mbh, mby = np.zeros_like(bh), np.zeros_like(by), np.zeros_like(why)
97 smooth_loss = 0.0
98 loss = 0.0
99
100 while True:
101
102     if p+seq_length+1 >= len(data) or n == 0:
103         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
104         p = 0 # go from start of data
105         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
106         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
107
108     # sample from the model now and then
109     if n % 100 == 0:
110         sample_ix = sample(hprev, inputs[0], 200)
111         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
112         print '----\n %s \n----' % (txt, )
113
114     # forward seq_length characters through the net and fetch gradient
115     loss, dwxh, dwhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
116     smooth_loss = smooth_loss * 0.999 + loss * 0.001
117     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
118
119     # perform parameter update with Adagrad
120     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
121                                   [dWxh, dWhh, dWhy, dbh, dby],
122                                   [mWxh, mWhh, mWhy, mbh, mby]):
123
124         mem += dparam * dparam
125         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
126
127         p += seq_length # move data pointer
128
129         n += 1 # iteration counter
```

Main loop



min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))}
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # Model parameters
22 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
23 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
24 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
25
26 while True:
27     def lossFun(inputs, targets, hprev):
28         """ Inputs, targets are both lists of integers.
29             hprev is RNN array of initial hidden state
30             returns the loss, gradients on model parameters, and last hidden state
31         """
32         xs, hs, ys, ps = [], [], [], []
33         hprev = np.copy(hprev)
34         for t in xrange(seq_length):
35             x = np.zeros(vocab_size).astype(np.float) # one-hot encoding for next chars
36             x[ix_to_ix[inputs[t]]] = 1
37             hprev = np.tanh(np.dot(wxh, x[t]) + np.dot(whh, hs[-1]) + bh)
38             y = np.exp(hprev)
39             ps.append(ps)
40             ys.append(ix_to_ix[np.argmax(y)])
41             loss += -np.log(y[targets[t]]) / np.sum(np.exp(y[t])) # softmax + cross-entropy loss
42             # backprop into hidden state
43             dxh = np.zeros_like(x[t])
44             dwh = np.zeros_like(whh)
45             dbh = np.zeros_like(bh)
46             dby = np.zeros_like(by)
47             dnext = np.zeros_like(hs[-1])
48             for param in [dWxh, dWhh, dWhy, dbh, dby]:
49                 param += np.outer(dxh, dparam)
50             dy = np.copy(ps[t])
51             dy[targets[t]] -= 1 # backprop into y
52             dh = np.dot(dy, dbh) * dtanh(hs[-1])
53             dby += dy
54             dwh += np.outer(dh, dxh) * dtanh(hs[-1])
55             dbh += np.dot(dtanh(hs[-1]), dh)
56             dnext = np.dot(dtanh(hs[-1]), dwh)
57             for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
58                 dparam += np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
59             if t >= seq_length-1:
60                 dh = np.zeros_like(hs[-1])
61             else:
62                 dh = dnext
63             dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs[t+1:], targets[t+1:])
64             if smooth_loss is None:
65                 smooth_loss = loss
66             else:
67                 smooth_loss = smooth_loss * 0.999 + loss * 0.001
68             if n % 100 == 0: print '----\n%d %f' % (n, smooth_loss)
69
70     # Sample a sequence of integers from the model
71     h = np.zeros((hidden_size,)) # hidden state, seed_ix is used later for first time step
72     sample_ix = sample(hprev, inputs[0], 200)
73     txt = ''.join(ix_to_char[ix] for ix in sample_ix)
74     print '----\n%s' % (txt, )
75
76 for t in xrange(n):
77     h = np.tanh(np.dot(wxh, x) + np.dot(whh, h) + bh)
78     y = np.exp(h)
79     p = np.array([y[i]/np.sum(np.exp(y)) for i in range(vocab_size)])
80     ix = np.random.choice(range(vocab_size), p=p.ravel())
81     x = np.zeros(vocab_size)
82     x[ix] = 1
83     inputs.append(ix)
84
85     # Sample from the model, now and then
86     sample_ix = sample(hprev, inputs[0], 200)
87     txt = ''.join(ix_to_char[ix] for ix in sample_ix)
88     print '----\n%s' % (txt, )
89
90     # Forward pass: compute gradients through the net and fetch gradient
91     loss, dxh, dwh, dbh, dby, hprev = lossFun(inputs, targets, hprev)
92     smooth_loss = smooth_loss * 0.999 + loss * 0.001
93     if t >= seq_length-1: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
94
95     # Perform parameter update with Adagrad
96     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
97                                   [dWxh, dWhh, dWhy, dbh, dby],
98                                   [mWxh, mWhh, mWhy, mbh, mby]):
99         mem += dparam * dparam
100        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
101
102    p += seq_length # move data pointer
103    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' % (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 simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # dict to map indices back to characters
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def init_random_params():
24     wh = np.random.rand(hidden_size, size=vocab_size)*0.01 # input to hidden
25     bh = np.random.rand(hidden_size, size=hidden_size)*0.01 # hidden to hidden
26     why = np.random.rand(vocab_size, size=hidden_size)*0.01 # hidden to output
27     by = np.zeros((vocab_size, size)) # hidden bias
28     bb = np.zeros((vocab_size, size)) # output bias
29
30     return wh, bh, why, by, bb
31
32 def lossFun(inputs, targets, hprev):
33
34     inputs,targets = both lists of integers.
35
36     hprev is Hx1 array of initial hidden state
37     returns the loss, gradients on model parameters, and last hidden state
38
39     xs, hs, ys, ps = O, O, O, O
40     hprev = np.copy(hprev)
41     loss = 0
42
43     for t in xrange(seq_length):
44         # forward pass
45         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
46         x[inputs[t]:] = 1
47
48         hprev = np.tanh(np.dot(hprev, wh) + np.dot(x, wI[t]) + bh) # hidden state
49         ps = np.exp(hprev) / np.sum(np.exp(hprev)) = softmax = probabilities for next chars
50         loss += -np.exp(y[t]) / np.sum(np.exp(y[t])) = cross-entropy loss
51
52         dy = np.zeros_like(ps)
53         dy[y[t]] = 1 # a backward pass into y
54         dh = np.dot(dy, wh.T) # dh = backprop into h
55         dwh = np.dot(x.T, dh) # dh = backprop through tanh nonlinearity
56         dbh = np.sum(dh, axis=0, keepdims=True) # dh = backprop through tanh nonlinearity
57         dby = np.dot(dy.T, np.ones((vocab_size, 1))) # dy = backprop into y
58         dby = np.sum(dy, axis=0, keepdims=True) # dy = backprop through softmax
59
60         for dparam in [dwh, dbh, dby]:
61             np.clip(dparam, -5, 5, out=dparam) = clip to mitigate exploding gradients
62
63         dh = np.dot(dy, wh.T) # dh = backprop into h
64         dwh = np.dot(x.T, dh) # dh = backprop through tanh nonlinearity
65         dbh = np.sum(dh, axis=0, keepdims=True) # dh = backprop through tanh nonlinearity
66         dby = np.dot(dy.T, np.ones((vocab_size, 1))) # dy = backprop into y
67         dby = np.sum(dy, axis=0, keepdims=True) # dy = backprop through softmax
68
69         if t > 0:
70             hprev = np.tanh(np.dot(hprev, wh) + np.dot(x, wI[t]) + bh)
71
72     return loss
73
74 # sample a sequence of integers from the model
75 # h is memory state, seed_ix is seed integer for first time step
76
77 def sample(hprev, inputs[0], 200):
78
79     x = np.zeros((vocab_size, 1))
80     x[seed_ix] = 1
81
82     for t in xrange(200):
83         h = np.tanh(np.dot(hprev, wh) + np.dot(inputs[t], wI[t]) + bh)
84         ps = np.exp(h) / np.sum(np.exp(h))
85         ix = np.random.choice(range(vocab_size), p=ps.ravel())
86         x[0] = 0
87         x[i] = 1
88
89     return ix
90
91
92 n, p = 0, 0
93 mem, mewh, mbhh, mwhy = np.zeros_like(Whh), np.zeros_like(Whh), np.zeros_like(Why)
94 mem, mbh, mby = np.zeros_like(bh), np.zeros_like(by), np.zeros_like(Why)
95 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
96 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
97
98 while True:
99
100     # sample from the model now and then
101     if n % 100 == 0:
102         sample_ix = sample(hprev, inputs[0], 200)
103         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
104         print '----\n %s ----' % (txt, )
105
106     # forward seq_length characters through the net and fetch gradient
107     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
108     smooth_loss = smooth_loss * 0.999 + loss * 0.001
109
110     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
111
112     # perform parameter update with Adagrad
113     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
114                                   [dWxh, dWhh, dWhy, dbh, dby],
115                                   [mWxh, mWhh, mWhy, mbh, mby]):
116
117         mem += dparam * dparam
118         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
119
120         p += seq_length # move data pointer
121
122         n += 1 # iteration counter
```

Main loop

min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # dict to map indices to characters
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # Model parameters
22 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
23 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
24 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
25
26 while True:
27     # Prepare inputs (we're sweeping from left to right in steps seq_length long)
28     if p+seq_length+1 >= len(data) or n == 0:
29         hprev = np.zeros((hidden_size,1)) # reset RNN memory
30         p = 0 # go from start of data
31         inputs = [c for c in data[p:p+seq_length]] # fetch sequence of inputs
32         targets = [c for c in data[p+1:p+seq_length+1]] # fetch sequence of targets
33
34     xs, hs, ys, ps = np.zeros((4, hidden_size, 1))
35     hprev = np.copy(hprev)
36     loss, smooth_loss = 0.0, 0.0
37
38     for t in range(seq_length):
39         # Unpack and reformat the previous hidden state as current context
40         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
41         x[inputs[t]] = 1.0
42         hprev = np.tanh(np.dot(Whh, hprev) + np.dot(Wxh, x) + bh) # hidden state
43         y = np.exp(hprev) # softmax output
44         ps = np.sum(np.exp(y)) # probabilities for next chars
45         y = np.exp(y)/ps # softmax output
46         loss += -np.log(ps[targets[t]])/ps # softmax (cross-entropy loss)
47         smooth_loss = smooth_loss * 0.999 + loss * 0.001
48
49         dy = np.copy(ps)*0.01
50         dy[targets[t]] -= 1.0 # backprop into y
51         dh = np.dot(Why, dy) # backprop through Why
52         dh += dy # backprop into h
53         dh = np.tanh(dh) * (1 - dh**2) # backprop through tanh nonlinearity
54         dWhh, dWxh, dbh = np.dot(dy, hprev.T) # dh backprop through tanh nonlinearity
55         dWhh += dWhh # gradient accumulation
56         dWxh += dWxh # gradient accumulation
57         dbh += dbh # gradient accumulation
58         dhnext = np.zeros((hidden_size, 1)) # initialize next hidden state
59
60     for param in [dWxh, dWhh, dbh, dWhy]:
61         np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
62
63     for param, mem in zip([dWxh, dWhh, dbh, dWhy], [mWxh, mWhh, mbh, mby]):
64         mem += param * param # memory variable for Adagrad
65
66     for t in range(seq_length):
67         x = np.zeros((vocab_size, 1))
68         x[inputs[t]] = 1.0
69         x[seed_ix] = 1.0
70
71     for t in range(seq_length):
72         x = np.tanh(np.dot(Whh, h) + np.dot(Wxh, x) + bh)
73         y = np.exp(x) / np.sum(np.exp(x))
74         ix = np.random.choice(range(vocab_size), p=y.ravel())
75         x[seed_ix] = 1.0
76         x[inputs[t]] = 0.0
77         h = hnext
78
79     return loss
80
81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85
86 while True:
87     # forward pass
88     if p+seq_length+1 >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size,1)) # reset RNN memory
90         p = 0 # go from start of data
91         inputs = [c for c in data[p:p+seq_length]] # fetch sequence of inputs
92         targets = [c for c in data[p+1:p+seq_length+1]] # fetch sequence of targets
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print '----\n%s\n----' % (txt, )
99
100
101     # forward seq_length characters through the net and fetch gradient
102     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
103     smooth_loss = smooth_loss * 0.999 + loss * 0.001
104
105     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
106
107
108     # perform parameter update with Adagrad
109     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
110                                   [dWxh, dWhh, dWhy, dbh, dby],
111                                   [mWxh, mWhh, mWhy, mbh, mby]):
112         mem += dparam * dparam
113         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
114
115     p += seq_length # move data pointer
116     n += 1 # iteration counter
```



Main loop

min-char-rnn.py gist

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # 14
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def init_randomhidden(state, vocab_size):
24     wh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
25     bh = np.random.rand(hidden_size, vocab_size)*0.01 # hidden to hidden
26     why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
27     by = np.zeros((vocab_size, 1)) # hidden bias
28     mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
29
30     return wh, bh, why, by, mbh, mby
31
32 def lossFun(inputs, targets, hprev):
33
34     inputs, targets = both lists of integers.
35
36     hprev is Hx1 array of initial hidden state
37     returns the loss, gradients on model parameters, and last hidden state
38
39     xs, hs, ys, ps = O, O, O, O
40     h1 = np.copy(hprev)
41
42     for t in xrange(seq_length):
43         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
44         x[0][char_to_ix[inputs[t]]] = 1
45
46         h1 = np.tanh(np.dot(wh, x1) + np.dot(bh, h1[-1]) + bh) # hidden state
47         ps1 = np.exp(h1) / np.sum(np.exp(h1)) # probabilities for next chars
48         y1 = np.argmax(ps1) # next char predicted
49         loss += -np.log(ps1[y1]) / np.sum(np.exp(h1)) # softmax (cross-entropy loss)
50
51         # backprop into hidden state
52         dprev_dh = np.zeros((hidden_size, 1))
53         dprev_dh1 = np.zeros((hidden_size, 1))
54         dprev_dy = np.zeros((vocab_size, 1))
55
56         dprev_dh += np.dot(why.T, dy) + dhnext # backward into h
57         dh += np.dot(why, dy) # backprop through tanh nonlinearity
58         dh += dprev_dh1 # backward into previous h
59         dprev_dy += np.dot(bh, dy) # backprop through bias
60         dy = np.copy(dy[1:])
61
62         dy[targets[t]] -= 1 # backprop into y
63         dhnext = np.dot(why, dy) # backprop into next hidden state
64
65         dx = np.dot(wh.T, dy) # backprop through tanh nonlinearity
66         dbh = np.sum(dx * dy, axis=0, keepdims=True) # backprop into bias
67         dwh = np.sum(dx * np.expand_dims(x1, 1), axis=0, keepdims=True) # backprop into weights
68         dprev_dx = np.dot(wh, x1) # backprop into previous x
69
70         for dparam in [dwh, dbh, ddy]:
71             if dparam != 0:
72                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
73
74         if np.isnan(dbh) or np.isnan(dwh) or np.isnan(ddy):
75             raise ValueError("Nan error")
76
77         dhnext = np.dot(why, dy) # backprop into next hidden state
78
79     if np.isnan(loss):
80         raise ValueError("Nan error")
81
82     # sample from the model now and then
83     if n % 100 == 0:
84         sample_ix = sample(hprev, inputs[0], 200)
85         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
86         print '----\n %s \n----' % (txt, )
87
88     # forward seq_length characters through the net and fetch gradient
89     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
90     smooth_loss = smooth_loss * 0.999 + loss * 0.001
91
92     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
93
94
95     # perform parameter update with Adagrad
96     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
97                                   [dWxh, dWhh, dWhy, dbh, dby],
98                                   [mWxh, mWhh, mWhy, mbh, mby]):
99
100        mem += dparam * dparam
101
102        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
103
104
105        p += seq_length # move data pointer
106
107        n += 1 # iteration counter
108
109
110
111
112
```

Main loop



min-char-rnn.py gist

```
***  
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)  
BSD License  
***  
Import numpy as np  
  
# Data I/O  
data = open('input.txt', 'r').read() # should be simple plain text file  
chars = list(set(data))  
data_size, vocab_size = len(data), len(chars)  
print 'data has %d characters, %d unique.' % (data_size, vocab_size)  
char_to_ix = {ch:i for i, ch in enumerate(chars)}  
ix_to_char = {i:ch for ch in enumerate(chars)}  
  
# Hyperparameters  
hidden_size = 100 # size of hidden layer of neurons  
seq_length = 20 # number of steps to unroll the RNN for  
learning_rate = 1e-1  
  
# Model parameters  
wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden  
bh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden  
why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output  
bh0 = np.zeros(hidden_size, 1) # hidden bias  
by = np.zeros(vocab_size, 1) # output bias  
  
def lossFun(inputs, targets, hprev):  
  
    inputs, targets are both list of integers.  
    hprev is Hx1 array of initial hidden state  
    returns the loss, gradients on model parameters, and last hidden state  
    ***  
  
    xs, hs, ys, ps = {}, {}, {}, {}  
    hs[-1] = np.copy(hprev)  
    loss = 0  
  
    # forward pass  
    for t in xrange(len(inputs)):  
        xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation  
        xs[t][inputs[t]] = 1  
        wht = np.tanh(np.dot(wh, xs[t]) + np.dot(bh, hs[t-1]) + bh) # hidden state  
        whyt = np.dot(why, wht) + by # unnormalized log probabilities for next chars  
        ps[t] = np.exp(whyt) / np.sum(np.exp(whyt)) # softmax (cross-entropy loss)  
        loss += -np.log(ps[t][targets[t],0])  
        # backward pass through tanh nonlinearity  
        dwhy = np.zeros_like(why)  
        dbh = np.zeros_like(bh)  
        dhnext = np.zeros_like(hs[0])  
        for i in range(len(xs[t])):  
            dy = np.copy(ps[t])[i]  
            dy[targets[t]] -= 1 # backprop into y  
            dy *= dy.T  
            dy *= dy  
            dh = np.dot(dy.T, dy) + dhnext # backprop into h  
            dh *= np.dot(dy, xs[t]) # backprop through tanh nonlinearity  
            dwh = np.dot(dy, xs[t].T)  
            dbh += np.sum(dh, axis=0, keepdims=True)  
            dhnext = np.dot(why, dh) # backprop into hnext  
        for dparam in [dwh, dbh, dwhy]:  
            np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
        for dparam in [dwh, dbh, dwhy]:  
            dparam *= learning_rate  
            dparam /= np.sqrt(np.sum(dparam**2))  
  
    def sample(hx, ix):  
  
        # sample a sequence of integers from the model  
        h is memory state, seed_ix is seed integer for first time step  
        h = np.zeros((hidden_size,1))  
        x = np.zeros((vocab_size, 1))  
        x[seed_ix] = 1  
        for t in xrange(n):  
            h = np.tanh(np.dot(wh, x) + np.dot(bh, h) + bh)  
            p = np.exp(why * np.dot(h, why))  
            ix = np.argmax(np.random.multinomial(1, p))  
            x = np.zeros((vocab_size, 1))  
            x[ix] = 1  
            if ix == 0: break  
        return ix  
  
    n, p, r, b  
    mean, stdv, why = np.zeros_like(why), np.zeros_like(why), np.zeros_like(why)  
    mean, stdv, why = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(bh)  
    smooth_loss = -np.inf  
    for i in range(seq_length):  
        # forward pass: compute loss over the next seq_length length  
        if p+seq_length > len(data) or r == 0:  
            p = np.zeros((hidden_size,1)) # reset RNN memory  
            r = 1  
        inputs = char_to_ix[x] # get char  
        targets = char_to_ix[x] # get char  
        for ch in data[p+seq_length-1]:  
            targets.append(ch)  
  
        # forward pass: compute loss over the next seq_length length  
        loss, dh0, dwh, dbh, dwhy, dby, dprev = lossFun(inputs, targets, p)  
        smooth_loss = smooth_loss * 0.999 + loss * 0.001  
        if r == 100: print 'iter %d, loss: %f' % (i, smooth_loss) # print progress  
  
        # backward pass: compute gradients going backwards  
        dwhh, dwhyh, dwhy = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)  
        dbhh, dbyh, dby = np.zeros_like(bh), np.zeros_like(by)  
        dhnext = np.zeros_like(hs[0])  
        for t in reversed(xrange(len(inputs))):  
            dy = np.copy(ps[t])  
            dy[targets[t]] -= 1 # backprop into y  
            dwhyh += np.dot(dy, hs[t].T)  
            dbhh += dy  
            dhnext = np.dot(why, dhnext) # backprop into h  
            dwhh = (1 - hs[t] * hs[t]) * dhnext # backprop through tanh nonlinearity  
            dbhh += dwhh  
            dwhh += np.dot(dhnext, xs[t].T)  
            dhnext = np.dot(why.T, dby) + dhnext # backprop into h  
  
        ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity  
        dbhh += ddraw  
        dwhh += np.dot(ddraw, xs[t].T)  
        dhnext = np.dot(why.T, ddraw) # backprop into hnext  
        for dparam in [dwhh, dwhyh, dwhy, dbhh, dbyh, dby]:  
            np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
        for dparam in [dwhh, dwhyh, dwhy, dbhh, dbyh, dby]:  
            dparam *= learning_rate  
            dparam /= np.sqrt(np.sum(dparam**2))  
  
    p = seq_length  
    n = 1 # iteration counter
```

Loss function

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

```
27 def lossFun(inputs, targets, hprev):  
28     """  
29         inputs,targets are both list of integers.  
30         hprev is Hx1 array of initial hidden state  
31         returns the loss, gradients on model parameters, and last hidden state  
32         """  
33  
34     xs, hs, ys, ps = {}, {}, {}, {}  
35     hs[-1] = np.copy(hprev)  
36     loss = 0  
37     # forward pass  
38     for t in xrange(len(inputs)):  
39         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation  
40         xs[t][inputs[t]] = 1  
41         hs[t] = np.tanh(np.dot(wxh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state  
42         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars  
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars  
44     loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)  
45     # backward pass: compute gradients going backwards  
46     dwxh, dwhh, dwhy = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)  
47     dbh, dby = np.zeros_like(bh), np.zeros_like(by)  
48     dhnext = np.zeros_like(hs[0])  
49     for t in reversed(xrange(len(inputs))):  
50         dy = np.copy(ps[t])  
51         dy[targets[t]] -= 1 # backprop into y  
52         dwhy += np.dot(dy, hs[t].T)  
53         dbh += dy  
54         dh = np.dot(why.T, dy) + dhnext # backprop into h  
55         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity  
56         dbh += ddraw  
57         dwhh += np.dot(ddraw, xs[t].T)  
58         dhnext = np.dot(why.T, ddraw)  
59     for dparam in [dwxh, dwhh, dwhy, dbh, dby]:  
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
61     return loss, dwxh, dwhh, 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
6 import numpy as np
7
8 # Data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = {ch:i for i in xrange(len(chars))}
14 ix_to_char = {i:ch for ch in xrange(len(chars))}

15 # Hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # Model parameters
21 Wxh = np.random.rand(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.rand(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros(hidden_size, 1) # hidden bias
25 by = np.zeros(vocab_size, 1) # output bias
26 by = 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
34     xs, hs, ys, ps = {}, {}, {}, {}
35     hs[-1] = np.copy(hprev)
36     loss = 0
37     # forward pass
38     for t in xrange(len(inputs)):
39         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
42         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)

45     # backward pass: compute gradients going backwards
46     dch, dhb, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
47     dbh, dy = np.zeros_like(by), np.zeros_like(by)
48     dnext = np.zeros_like(hs[0])
49     for t in reversed(xrange(len(inputs))):
50         dy = np.copy(xs[t])
51         dy[targets[t]] -= 1 # backprop into y
52         dhy = np.dot(Why.T, dy) # backprop through Why
53         dbh += np.dot(dy.T, dhb) # backprop through tanh nonlinearity
54         dch += np.dot(dhy.T, dch) # backprop through tanh nonlinearity
55         dxs = np.dot(Wxh.T, dch) # backprop through Wxh
56         dxs += np.dot(Whh.T, dch) # backprop through Whh
57         dxs += np.zeros((vocab_size, 1)) # backprop through bias
58         dnext = np.dot(Wxh, dnext) # backprop through Wxh
59         dnext = np.tanh(dnext) # backprop through tanh
60         dch = np.dot(Wxh.T, dnext) + dch # backprop into h
61         dhb += np.dot(dch.T, dhb) # backprop through tanh nonlinearity
62         dbh += dch # backprop through tanh nonlinearity
63         dhy += dch # backprop through Why
64     for opname in [dch, dhb, dhy, dbh, ddy]:
65         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
66         np.clip(dbh, -5, 5, out=dbh) # clip to mitigate exploding gradients
67         np.clip(dhy, -5, 5, out=dhy) # clip to mitigate exploding gradients
68
69     def sample(hseed, ix, n):
70         """"
71         sample a sequence of integers from the model
72         h is memory state, seed_ix is seed for first time step
73         """
74         x = np.zeros((vocab_size, 1))
75         x[seed_ix] = 1
76
77         for t in xrange(n):
78             h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
79             p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
80             ix = np.argmax(np.random.ranf(vocab_size), np.ravel())
81             x[0] = 0
82             x[i] = 1
83
84         return ix
85
86     n, p, r, b, m, mch, mhy = np.zeros(vocab_size), np.zeros_like(bh), np.zeros_like(why)
87     mem, memh, memy = np.zeros_like(bh), np.zeros_like(by), np.zeros_like(why)
88     smooth_loss = 0
89     seq_length = len(data)/seq_length + 1
90     while True:
91         if p == 0 or r == 0 or b == 0:
92             p = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
93             r = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
94             b = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
95             m = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
96             mch = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
97             mhy = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
98
99         inputs = [char_to_ix[ch] for ch in data[p:seq_length]]
100         targets = [char_to_ix[ch] for ch in data[p+1:seq_length+1]]
101
102         # forward pass: compute scores, through the net and fetch gradient
103         loss, dch, dhb, dhy, dbh, ddy, hprev = lossFun(inputs, targets, hprev)
104         smooth_loss = smooth_loss * 0.999 + loss * 0.001
105         if r < 100: print 'iter %d, loss: %e' % (r, smooth_loss) # print progress
106
107         # backward pass: compute gradients, through the net and fetch gradient
108         for param, dparam, mem in zip([Wxh, Whh, why, bh, by],
109                                     [dch, dhb, dhy, dbh, ddy],
110                                     [mem, memh, memy, mem, memy]):
111             dparam += learning_rate * dparam / np.sqrt(m + 1e-8) # adaptive update
112             param += dparam
113
114         p = seq_length * n # move data pointer
115         n = 1 # iteration counter

```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

```

27 def lossFun(inputs, targets, hprev):
28     """"
29     inputs,targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33
34     xs, hs, ys, ps = {}, {}, {}, {}
35     hs[-1] = np.copy(hprev)
36     loss = 0
37     # forward pass
38     for t in xrange(len(inputs)):
39         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
42         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)

45     # backward pass: compute gradients going backwards
46     dch, dhb, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
47     dbh, dy = np.zeros_like(by), np.zeros_like(by)
48     dnext = np.zeros_like(hs[0])
49     for t in reversed(xrange(len(inputs))):
50         dy = np.copy(xs[t])
51         dy[targets[t]] -= 1 # backprop into y
52         dhy = np.dot(Why.T, dy) # backprop through Why
53         dbh += np.dot(dy.T, dhb) # backprop through tanh nonlinearity
54         dch += np.dot(dhy.T, dch) # backprop through tanh nonlinearity
55         dxs = np.dot(Wxh.T, dch) # backprop through Wxh
56         dxs += np.dot(Whh.T, dch) # backprop through Whh
57         dxs += np.zeros((vocab_size, 1)) # backprop through bias
58         dnext = np.dot(Wxh, dnext) # backprop through Wxh
59         dnext = np.tanh(dnext) # backprop through tanh
60         dch = np.dot(Wxh.T, dnext) + dch # backprop into h
61         dhb += np.dot(dch.T, dhb) # backprop through tanh nonlinearity
62         dbh += dch # backprop through tanh nonlinearity
63         dhy += dch # backprop through Why
64     for opname in [dch, dhb, dhy, dbh, ddy]:
65         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
66         np.clip(dbh, -5, 5, out=dbh) # clip to mitigate exploding gradients
67         np.clip(dhy, -5, 5, out=dhy) # clip to mitigate exploding gradients
68
69     def sample(hseed, ix, n):
70         """"
71         sample a sequence of integers from the model
72         h is memory state, seed_ix is seed for first time step
73         """
74         x = np.zeros((vocab_size, 1))
75         x[seed_ix] = 1
76
77         for t in xrange(n):
78             h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
79             p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
80             ix = np.argmax(np.random.ranf(vocab_size), np.ravel())
81             x[0] = 0
82             x[i] = 1
83
84         return ix
85
86     n, p, r, b, m, mch, mhy = np.zeros(vocab_size), np.zeros_like(bh), np.zeros_like(why)
87     mem, memh, memy = np.zeros_like(bh), np.zeros_like(by), np.zeros_like(why)
88     smooth_loss = 0
89     seq_length = len(data)/seq_length + 1
90     while True:
91         if p == 0 or r == 0 or b == 0:
92             p = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
93             r = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
94             b = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
95             m = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
96             mch = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
97             mhy = np.zeros(vocab_size) / np.sum(np.zeros(vocab_size))
98
99         inputs = [char_to_ix[ch] for ch in data[p:seq_length]]
100         targets = [char_to_ix[ch] for ch in data[p+1:seq_length+1]]
101
102         # forward pass: compute scores, through the net and fetch gradient
103         loss, dch, dhb, dhy, dbh, ddy, hprev = lossFun(inputs, targets, hprev)
104         smooth_loss = smooth_loss * 0.999 + loss * 0.001
105         if r < 100: print 'iter %d, loss: %e' % (r, smooth_loss) # print progress
106
107         # backward pass: compute gradients, through the net and fetch gradient
108         for param, dparam, mem in zip([Wxh, Whh, why, bh, by],
109                                     [dch, dhb, dhy, dbh, ddy],
110                                     [mem, memh, memy, mem, memy]):
111             dparam += learning_rate * dparam / np.sqrt(m + 1e-8) # adaptive update
112             param += dparam
113
114         p = seq_length * n # move data pointer
115         n = 1 # iteration counter

```

min-char-rnn.py gist

```

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

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

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

25 def lossFun(inputs, targets, hprev):
26     """ inputs,targets are both lists of integers.
27     hprev is Hx1 array of initial hidden state
28     returns the loss, gradients on model parameters, and last hidden state
29     """
30
31     xs, hs, ys, ps = [], [], [], []
32     h0 = np.copy(hprev)
33     loss = 0
34     for t in xrange(len(inputs)):
35         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
36         x[inputs[t], 1] = 1
37         h0 = np.tanh(np.dot(h0, wh) + np.dot(x, xT) + bh) # hidden state
38         ys[t] = np.exp(np.dot(h0, why)) / np.sum(np.exp(ys[t])) # next chars
39         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
40         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
41
42         dnh, dwh, dbh, dhy = np.zeros_like(h0), np.zeros_like(wh), np.zeros_like(bh),
43         dnx, dby = np.zeros_like(h0), np.zeros_like(by)
44         dhnext = np.zeros_like(h0)
45         for i in range(len(inputs[:t])):
46             dy = np.copy(ps[i])
47             dy[targets[i]] -= 1 # backprop into y
48             dnx += np.dot(dy, x[i].T)
49             dwh += np.dot(dy, h0.T)
50             dbh += dy
51             dhy += dy
52
53             dh = np.dot(why.T, dy) + dhnext # backprop through tanh nonlinearity
54             ddraw = (1 - h0 * h0) * dh # backprop through tanh nonlinearity
55             dbh += ddraw
56             dwh += np.dot(ddraw, h0.T)
57             dhnext = np.dot(wh.T, ddraw)
58
59             dnx += np.dot(dy, x[i].T)
60             dwh += np.dot(dy, h0.T)
61             dbh += dy
62
63             dnx += np.dot(dy, x[i].T)
64             dwh += np.dot(dy, h0.T)
65             dbh += dy
66             dhnext = np.dot(wh.T, ddraw)
67
68         for dparam in [dnh, dwh, dbh, dhy]:
69             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
70
71         dhnext = np.zeros_like(h0)
72         dnx = np.zeros_like(h0)
73         dwh = np.zeros_like(wh)
74         dbh = np.zeros_like(bh)
75         dhy = np.zeros_like(why)
76
77     return loss, hs[-1], ps

78 def sample(h0, seed_ix, n):
79     """ sample a sequence of integers from the model
80     h is memory state, seed_ix is seed integer for first time step
81     """
82
83     x = np.zeros((vocab_size, 1))
84     x[seed_ix] = 1
85
86     for t in xrange(n):
87         h = np.tanh(np.dot(h0, wh) + np.dot(x, xT) + bh)
88         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
89         ix = np.random.choice(range(vocab_size), p=p.ravel())
90         x[0, ix] = 1
91         h0 = h
92
93     return ix

94 n, p = 0
95 mean, std = 0, 0.01
96 mem, memh, memy = np.zeros_like(h0), np.zeros_like(wh), np.zeros_like(why)
97 mem0, memh0, memy0 = np.zeros_like(h0), np.zeros_like(wh), np.zeros_like(why)
98 smooth_loss = 0
99 seqLength = 1000
100 seqLength = len(data) / seqLength + 1
101 while True:
102     if p == seqLength or (p == 0 and n >= seqLength):
103         hprev = np.zeros(hidden_size, 1) # reset RNN memory
104         inputs = [char_to_ix[ch] for ch in data[p:seqLength]]
105         targets = [char_to_ix[ch] for ch in data[p+seqLength-1:p]]
106
107         # Forward pass: compute activations through the net and fetch gradient
108         loss, dh0, dwh, dbh, dhy, hprev = lossFun(inputs, targets, hprev)
109         smooth_loss = smooth_loss * 0.999 + loss * 0.001
110         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss)
111
112         # Backward pass: compute gradients going backwards
113         dnh, dwhh, dwhy = np.zeros_like(h0), np.zeros_like(wh), np.zeros_like(why)
114         dbh, dby = np.zeros_like(h0), np.zeros_like(by)
115         dhnext = np.zeros_like(h0)
116
117         for t in reversed(xrange(len(inputs))):
118             dy = np.copy(ps[t])
119             dy[targets[t]] -= 1 # backprop into y
120             dwhh += np.dot(dy, h0.T)
121             dbh += dy
122             dhy += dy
123
124             dh = np.dot(why.T, dy) + dhnext # backprop into h
125             ddraw = (1 - h0 * h0) * dh # backprop through tanh nonlinearity
126             dbh += ddraw
127             dwhh += np.dot(ddraw, h0.T)
128             dhnext = np.dot(wh.T, ddraw)
129
130             dnx += np.dot(dy, x[t].T)
131             dwhh += np.dot(dy, h0.T)
132             dbh += dy
133
134             dnx += np.dot(dy, x[t].T)
135             dwhh += np.dot(dy, h0.T)
136             dbh += dy
137             dhnext = np.dot(wh.T, ddraw)
138
139         for dparam in [dnh, dwhh, dwhy, dbh, dby]:
140             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
141
142         dhnext = np.zeros_like(h0)
143         dnx = np.zeros_like(h0)
144         dwhh = np.zeros_like(wh)
145         dbh = np.zeros_like(bh)
146         dhy = np.zeros_like(why)
147
148     n += 1
149
150 p = seqLength
151 n = 1 # iteration counter

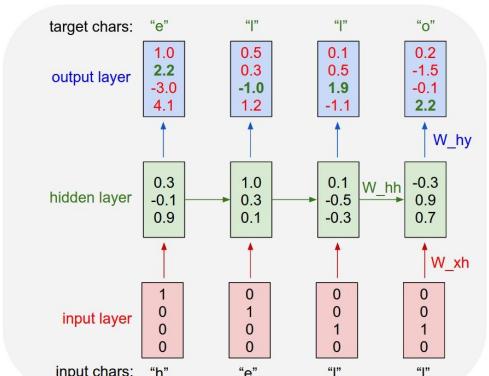
```

```

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

```

recall:



min-char-rnn.py gist

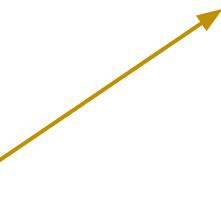
```
1 /**
2  * Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  * BSD License
4  */
5
6 import numpy as np
7
8 # Data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)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

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

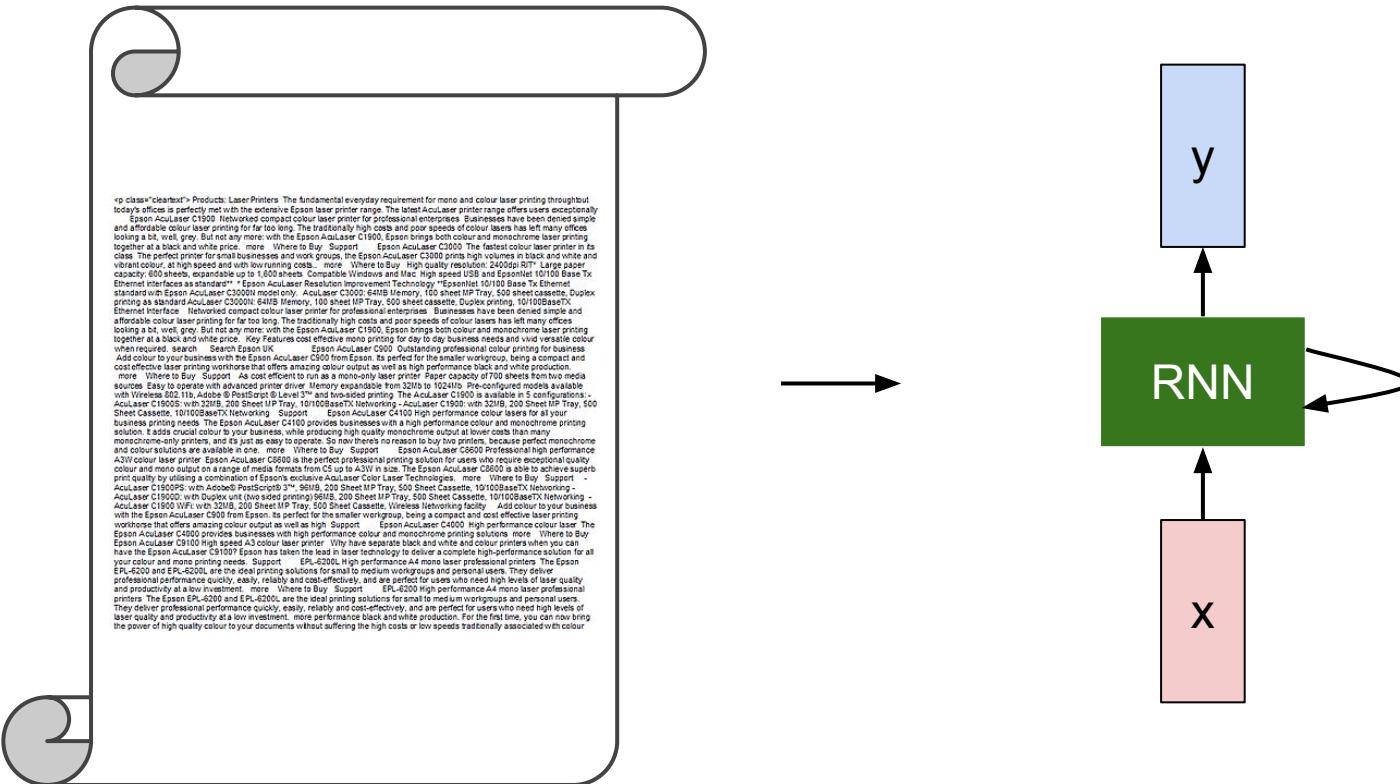
19 # Model parameters
20 dnh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
21 wh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
22 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
23 bh = np.zeros(hidden_size, 1) # hidden bias
24 by = np.zeros(vocab_size,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

25 def lossFun(inputs, targets, hprev):
26     """ Inputs, targets are both lists of integers.
27     hprev is h0 array of initial hidden state
28     returns the loss, gradients on model parameters, and last hidden state
29     """
30
31     xs, hs, ys, ps = np. zeros( (0, 0, 0, 0)
32     h0 = np.copy(hprev)
33     loss = 0
34
35     for t in xrange(len(inputs)):
36         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
37         x[inputs[t], 1] = 1
38
39         h1 = np.tanh(np.dot(wxh, x) + np.dot(whh, h0) + bh) # hidden state
40         y1 = np.dot(why, h1) + by # output
41         ps1 = np.exp(y1) / np.sum(np.exp(y1)) = softmax # probabilities for next chars
42         loss += -np.log(ps1[targets[t], 0]) = cross-entropy loss
43
44         # backward pass
45         dxh = np.zeros_like(x) # we backprop through tanh nonlinearity
46         dwh = np.zeros_like(wxh)
47         dwhy = np.zeros_like(why)
48         dbh = np.zeros_like(bh)
49         dnext = np.zeros_like(h1)
50
51         for i in range(len(inputs)-t, len(inputs)):
52             dy = np.copy(ps1[i])
53             dy[targets[i], 0] = -1 # backprop into y
54             dh = np.dot(dy, why.T)
55             dxh += dy # backprop into x
56             dwh += np.dot(dxh, x.T)
57             dwhy += np.sum(dxh * np.exp(y1), axis=0, keepdims=True)
58             dbh += np.sum(dxh, axis=0, keepdims=True)
59
60             for opname in [dnh, dwh, dwhy, dbh, ddy]:
61                 np.clip(opname, -5, 5, out=opname) = clip to mitigate exploding gradients
62             if t > 0:
63                 dnh, dwh, dwhy, dbh, ddy, h0 = hnext
64
65     return loss, hnext, ps1, ix

66 # sample a sequence of integers from the model
67 # h is memory state, seed_ix is seed letter for first time step
68
69 def sample(h, seed_ix, n):
70     """
71     sample a sequence of integers from the model
72     h is memory state, seed_ix is seed letter for first time step
73     """
74
75     x = np.zeros((vocab_size, 1))
76     x[seed_ix] = 1
77
78     ixes = []
79
80     for t in xrange(n):
81         h = np.tanh(np.dot(wxh, x) + np.dot(whh, h) + bh)
82         y = np.dot(why, h) + by
83         p = np.exp(y) / np.sum(np.exp(y))
84         ix = np.random.choice(range(vocab_size), p=p.ravel())
85         x[seed_ix] = 0
86         ixes.append(ix)
87
88     return ixes
89
90
91 # Main loop
92 inputs = np.zeros((seq_length, vocab_size))
93 targets = np.zeros((seq_length, vocab_size))
94 smooth_loss = 0.0
95
96 for i in range(seq_length):
97     inputs[i, :] = np.copy(chars[i])
98
99     if i > 0:
100        hprev = hnext
101
102    hnext, dxh, dwh, dwhy, dbh, ddy, hprev = lossFun(inputs, targets, hprev)
103
104    smooth_loss = smooth_loss * 0.999 + loss * 0.001
105
106    if i % 100 == 0:
107        print 'iter %d, loss: %f' % (i, smooth_loss)
108
109    for param, dparam, mem in zip([wh, why, bh, by],
110                                 [dwh, dwhy, dbh, ddy],
111                                 [dnh, dwh, dwhy, dbh]):
112        mem += dparam
113        param += -learning_rate * mem / np.sqrt(mem + 1e-8) = integrated update
114
115    p = np.argmax(hnext)
116
117    inputs[0, :] = np.zeros(vocab_size)
118
119    hnext = np.zeros(hidden_size)
120
121    n = 1 + iteration counter
```



```
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```



Sonnet 116 – Let me not ...

by William Shakespeare

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

at first:

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

↓ train more

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

↓ train more

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

↓ train more

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

PANDARUS:

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

Second Senator:

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

DUKE VINCENTIO:

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

Second Lord:

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

Clown:

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

VIOLA:

I'll drink it.

VIOLA:

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

KING LEAR:

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

open source textbook on algebraic geometry

The Stacks Project

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

Browse chapters

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

Parts

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

Statistics

The Stacks project now consists of

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

Latex source

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

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

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

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

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

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

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

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

is a unique morphism of algebraic stacks. Note that

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

and

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

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

Proof. See discussion of sheaves of sets. \square

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. Omitted. □

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

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

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

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

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

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

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

Proof. See Spaces, Lemma ??.

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

The following to the construction of the lemma follows.

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

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

be a morphism of algebraic spaces over S and Y .

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Linux kernel source tree

520,037 commits

1 branch

420 releases

5,039 contributors

branch: master · [linux](#) / +

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

 torvalds authored 9 hours agolatest commit 4b1786927d  Documentation

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

6 days ago

 arch

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

a day ago

 block

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

9 days ago

 crypto

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

10 days ago

 drivers

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

9 hours ago

 firmware

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

2 months ago

 fs

vfs: read file_handle only once in handle_to_path

4 days ago

 include

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

a day ago

 init

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

a month ago

 io

bio: bio_start: New bio_start and multi-block bio_start methods introduced

a month ago

 Code Pull requests
74 Pulse Graphs

HTTPS clone URL

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

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << i))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 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/seteew.h>
#include <asm/pgproto.h>
```

```

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

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

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

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

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

```

Searching for interpretable cells

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

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

Searching for interpretable cells

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

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

quote detection cell

Searching for interpretable cells

Cell sensitive to position in line:

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

line length tracking cell

Searching for interpretable cells

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

if statement cell

Searching for interpretable cells

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

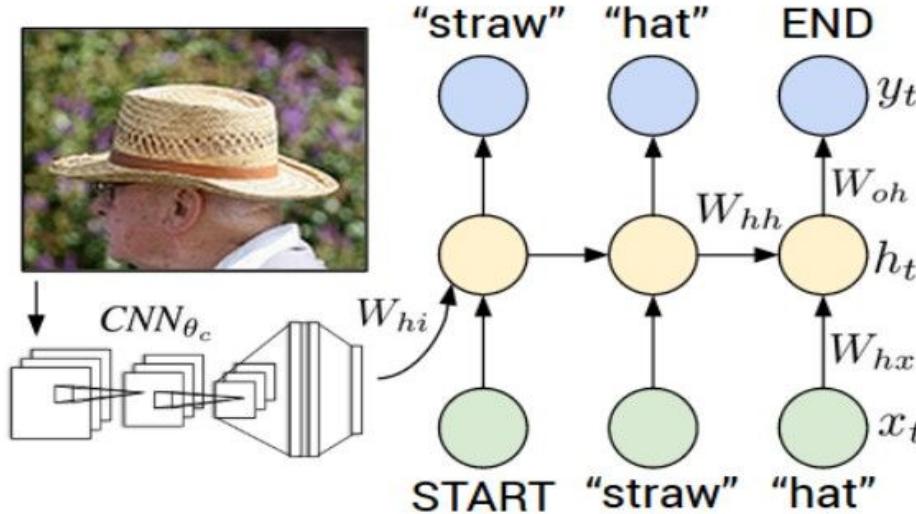
quote/comment cell

Searching for interpretable cells

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

code depth cell

Image Captioning



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

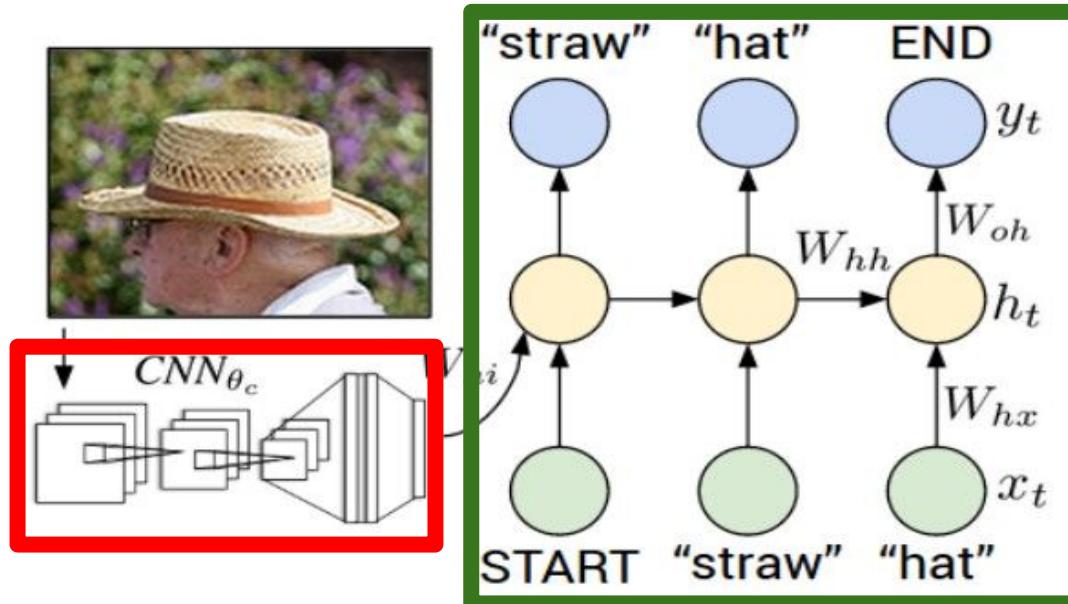
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

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

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

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

Recurrent Neural Network



Convolutional Neural Network

test image



image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

X

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

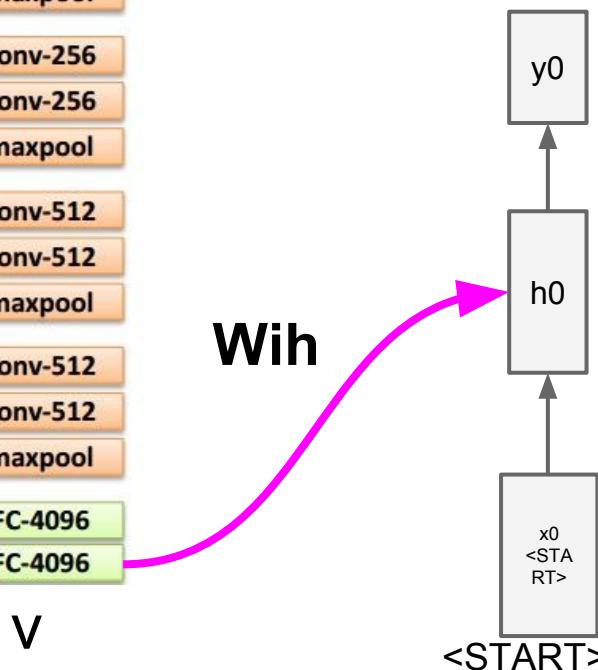
FC-4096



<START>



test image



before:

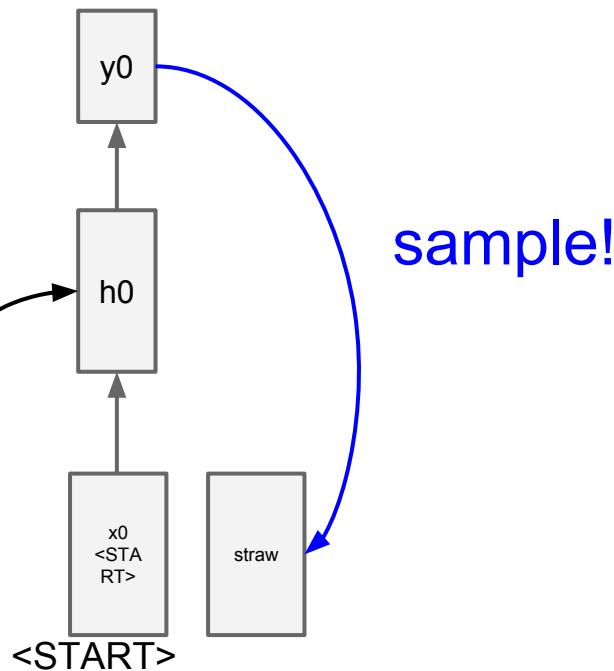
$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

now:

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



test image



image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

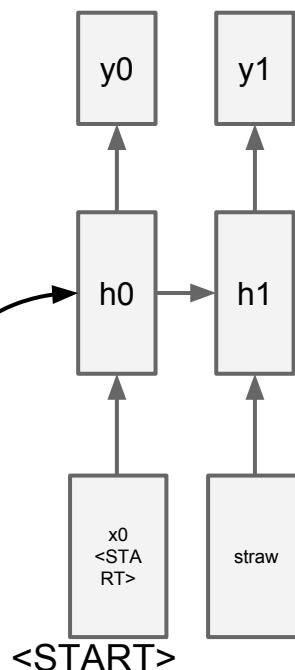
conv-512

conv-512

maxpool

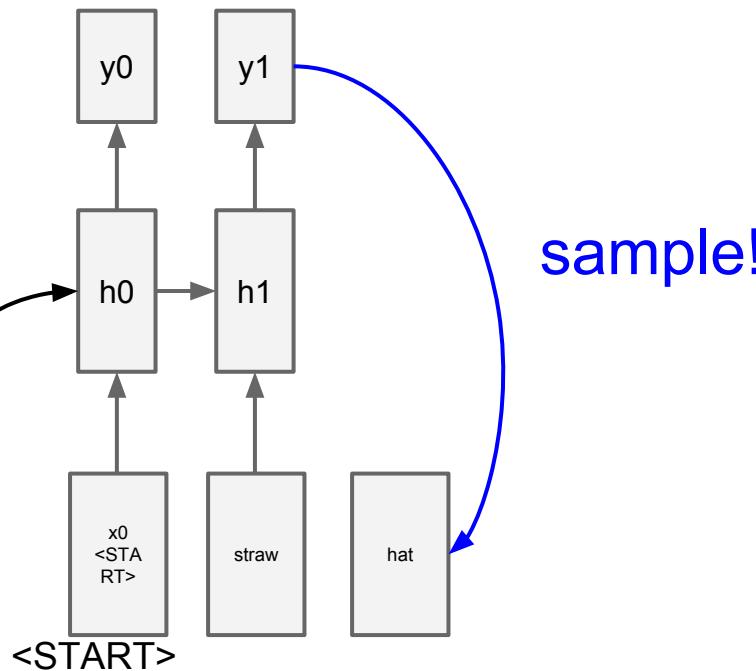
FC-4096

FC-4096





test image



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

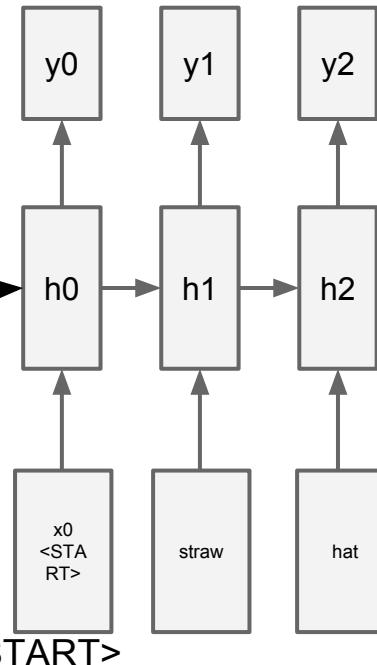
conv-512

conv-512

maxpool

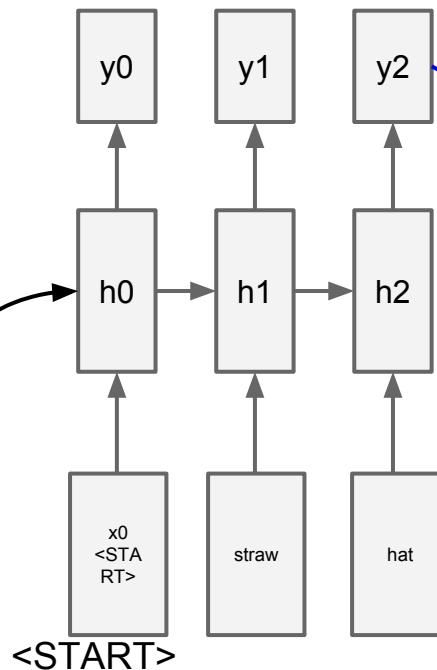
FC-4096

FC-4096





test image



sample
<END> token
=> finish.

Image Sentence Datasets

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



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

currently:
~120K images
~5 sentences each



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



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



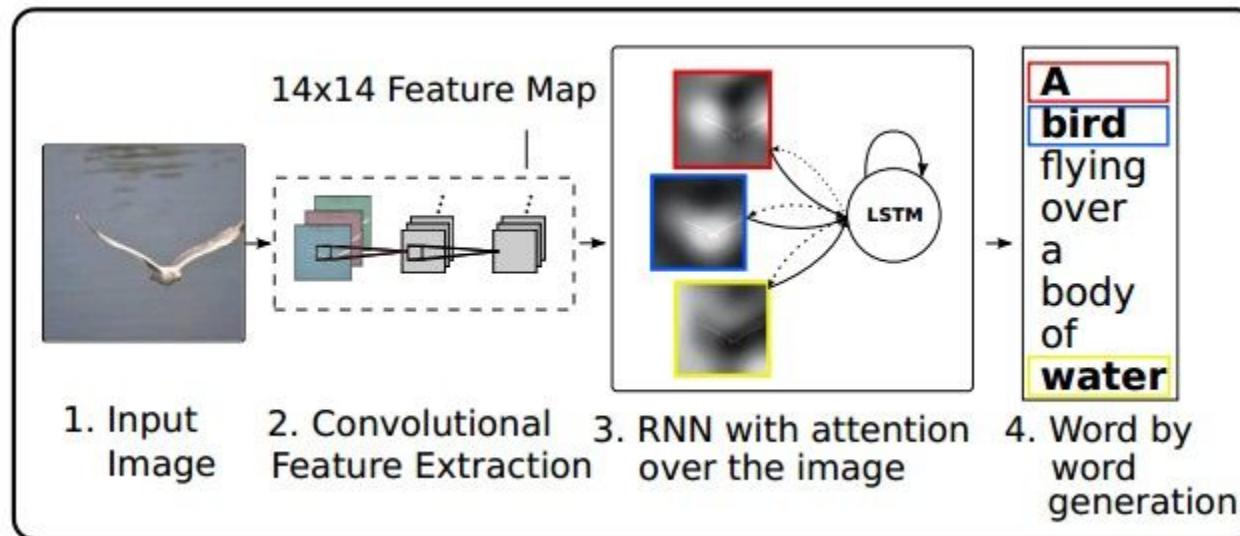
"a woman holding a teddy bear in front of a mirror."



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

Preview of fancier architectures

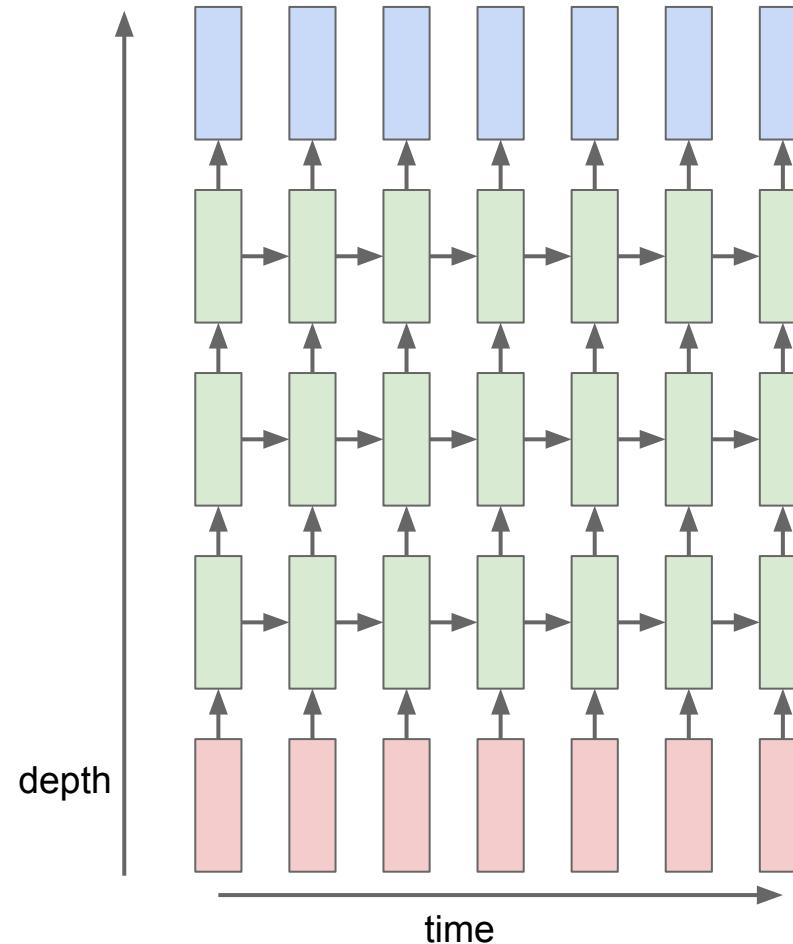
RNN attends spatially to different parts of images while generating each word of the sentence:



RNN:

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

$h \in \mathbb{R}^n$. W^l [n × 2n]



RNN:

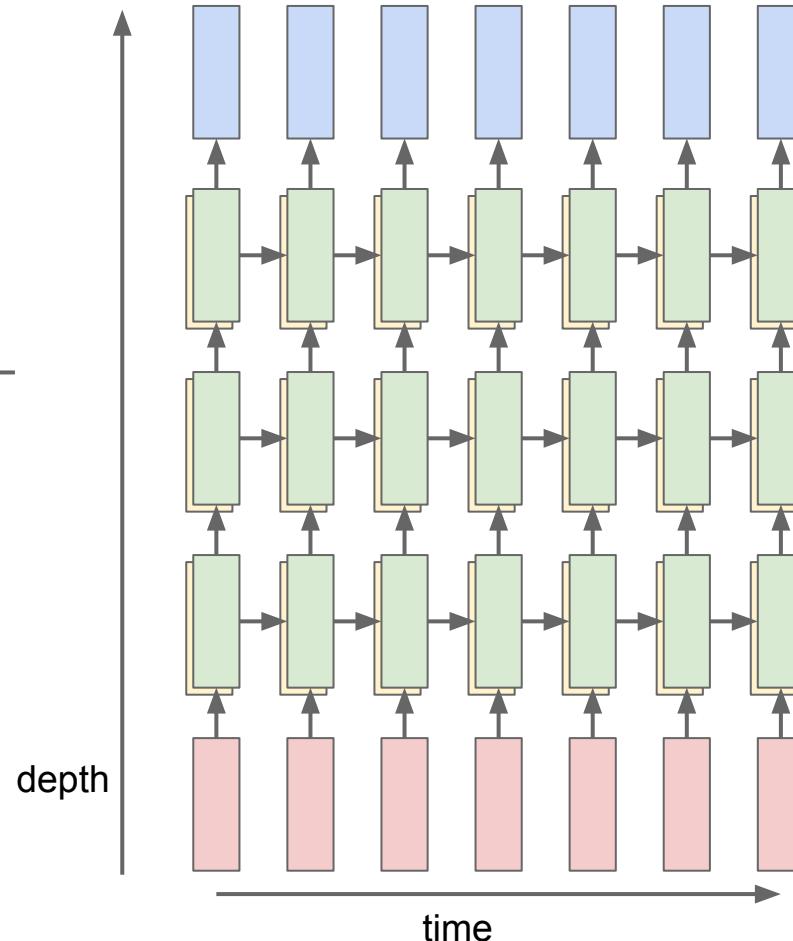
$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

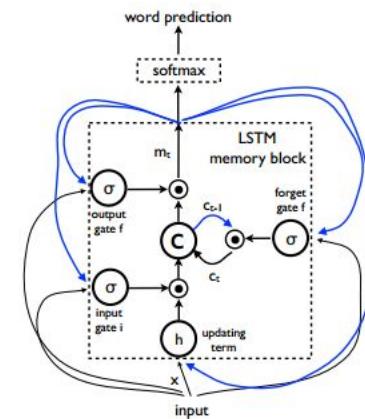
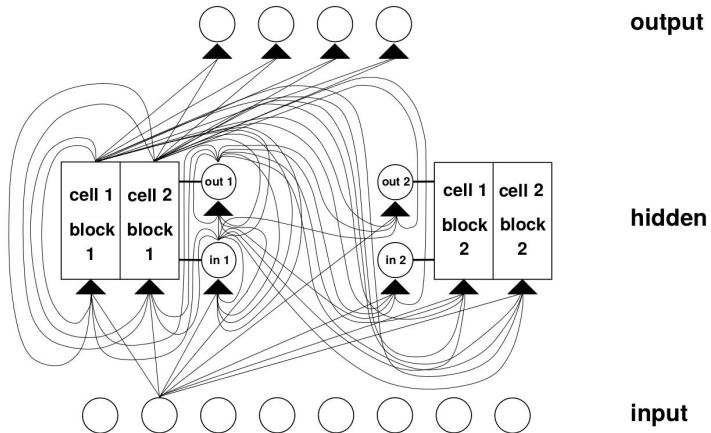
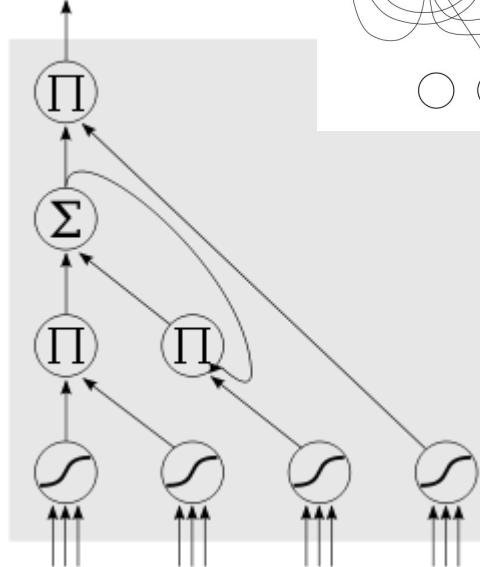
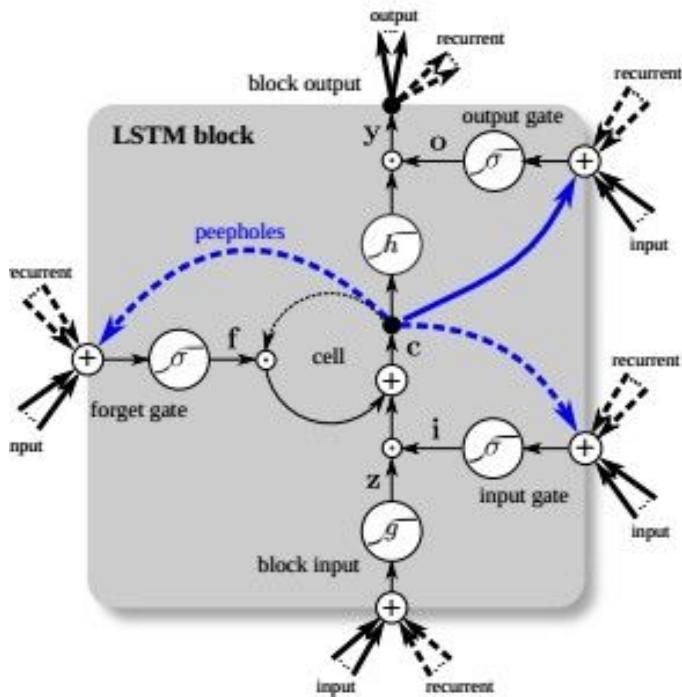
LSTM:

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

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

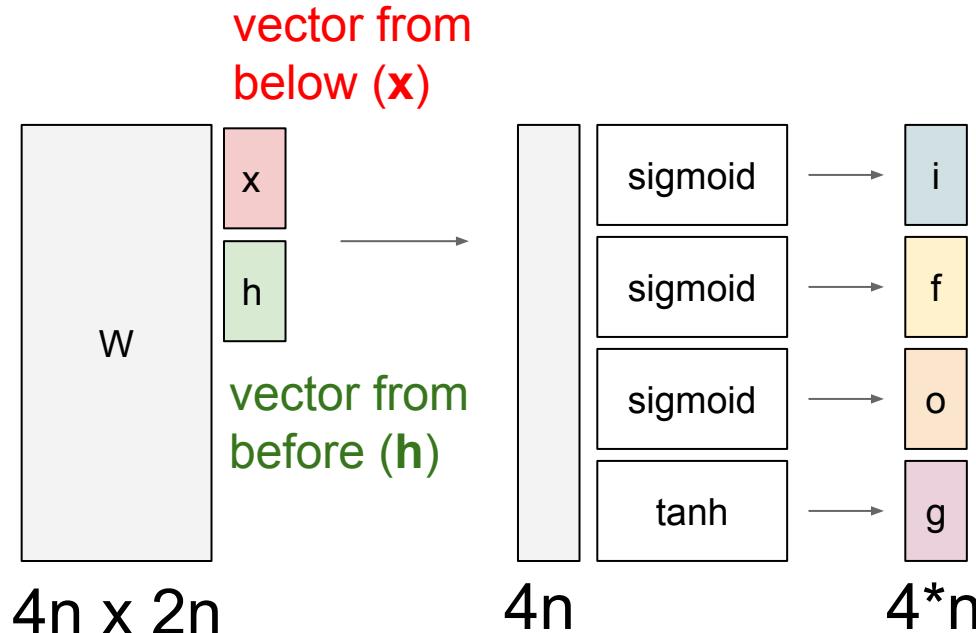


LSTM



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

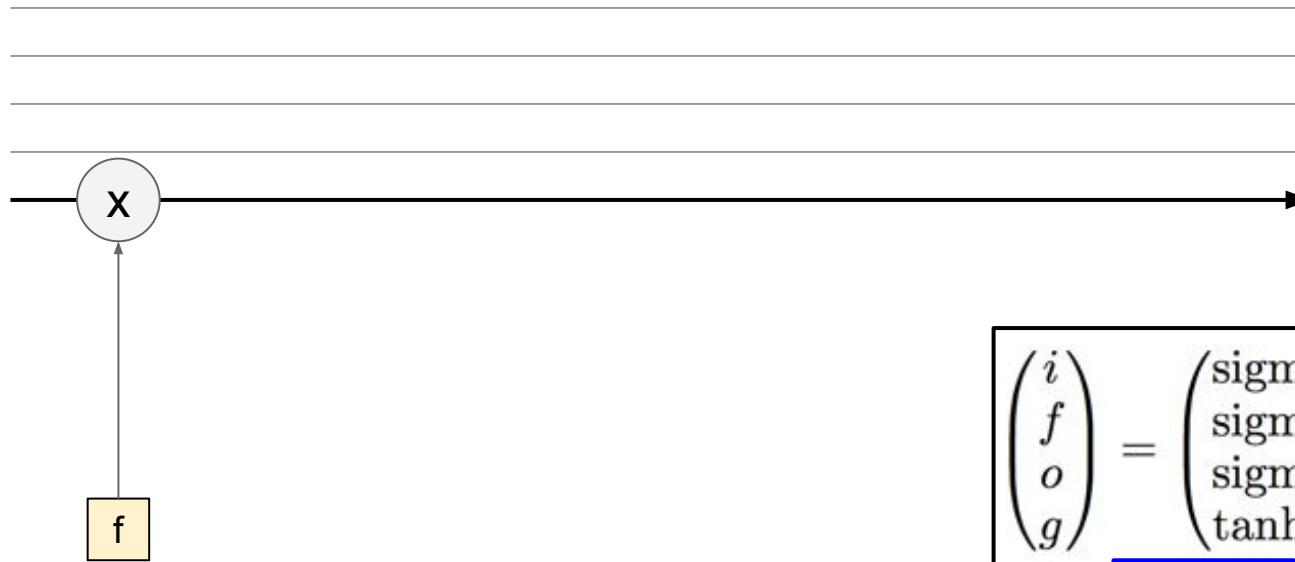


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_t^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state **c**

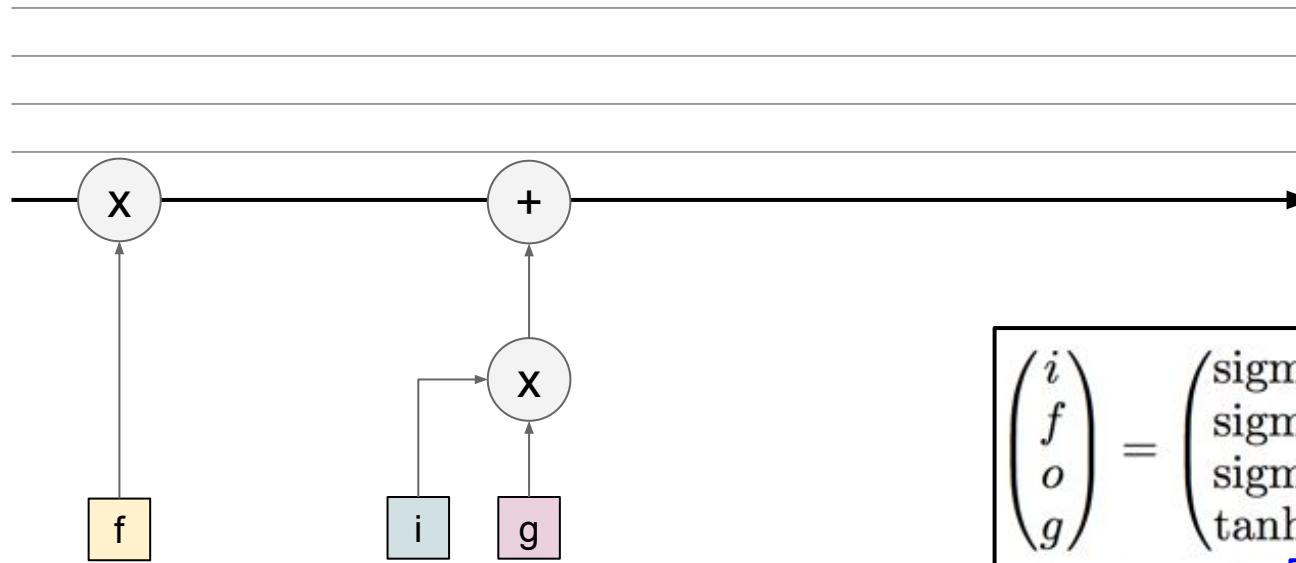


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c

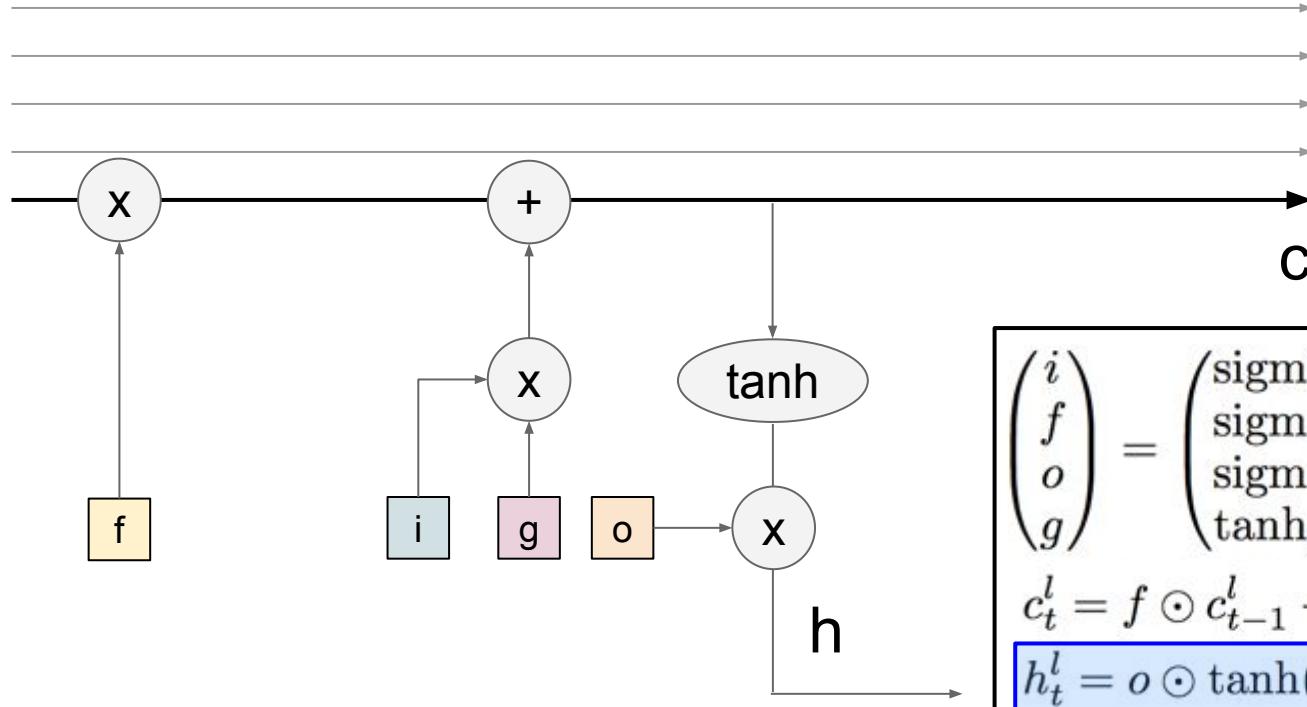


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c

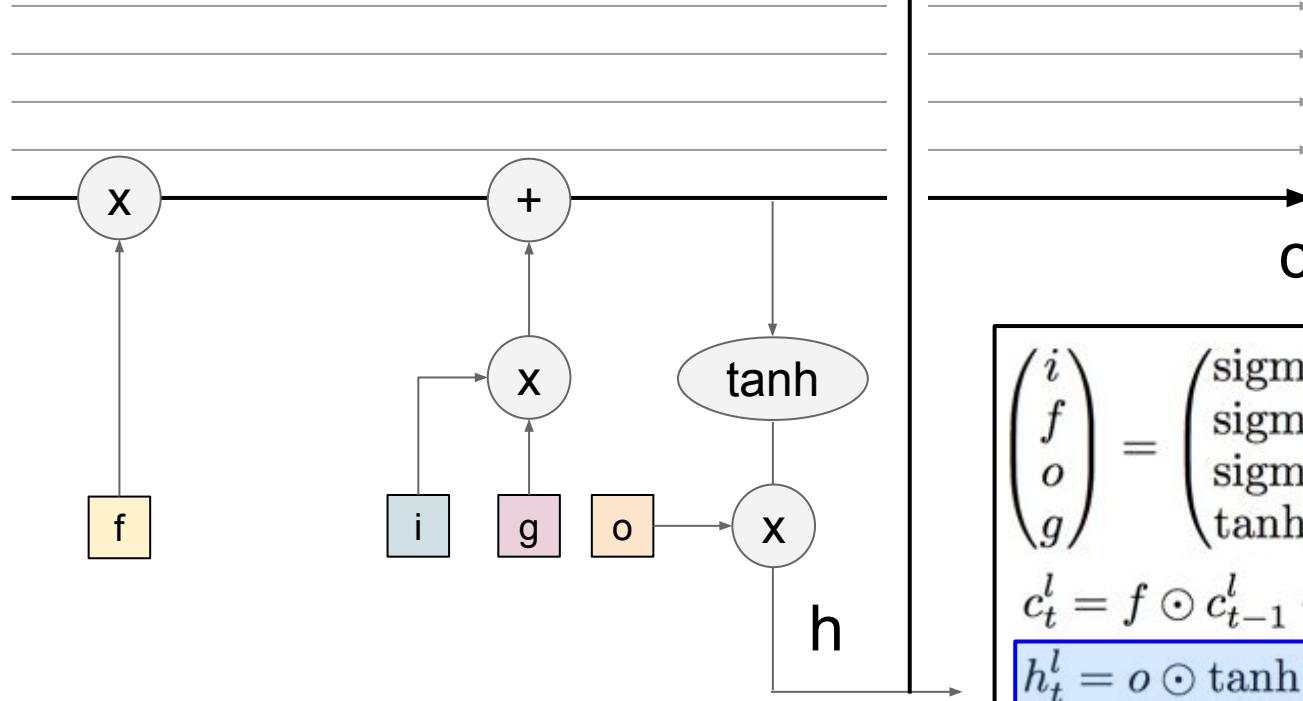


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c



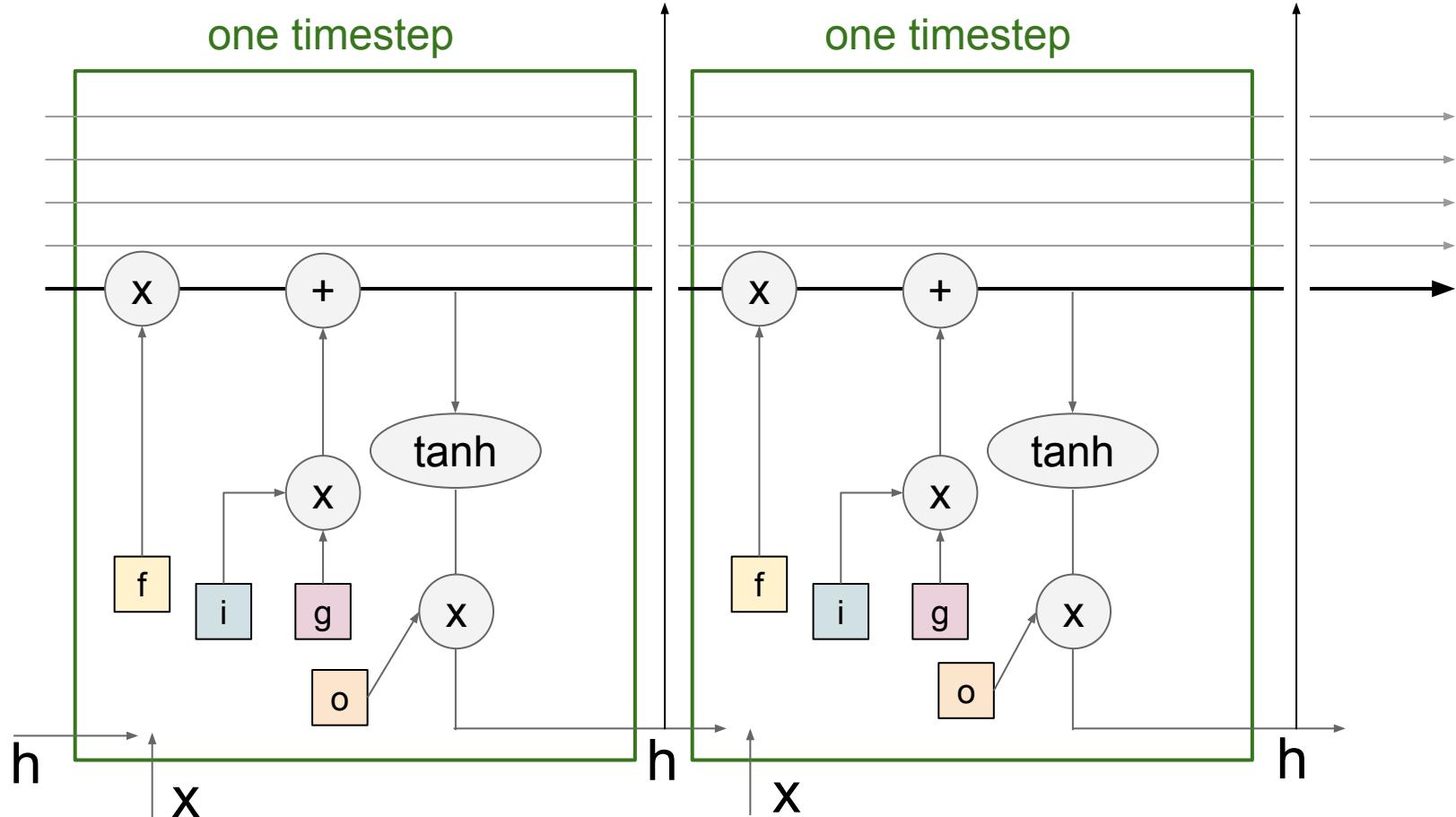
higher layer, or
prediction

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

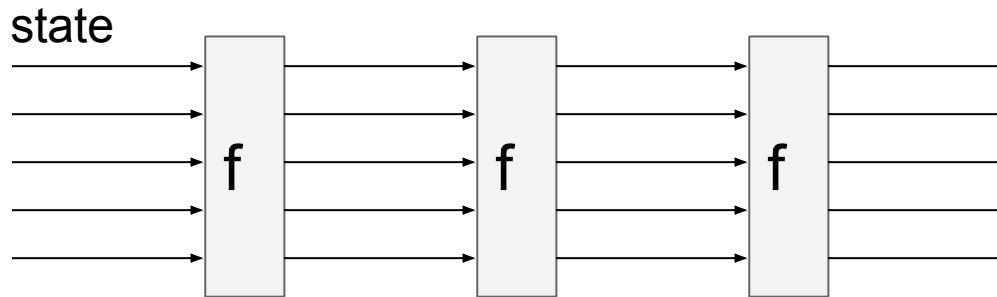
LSTM

one timestep

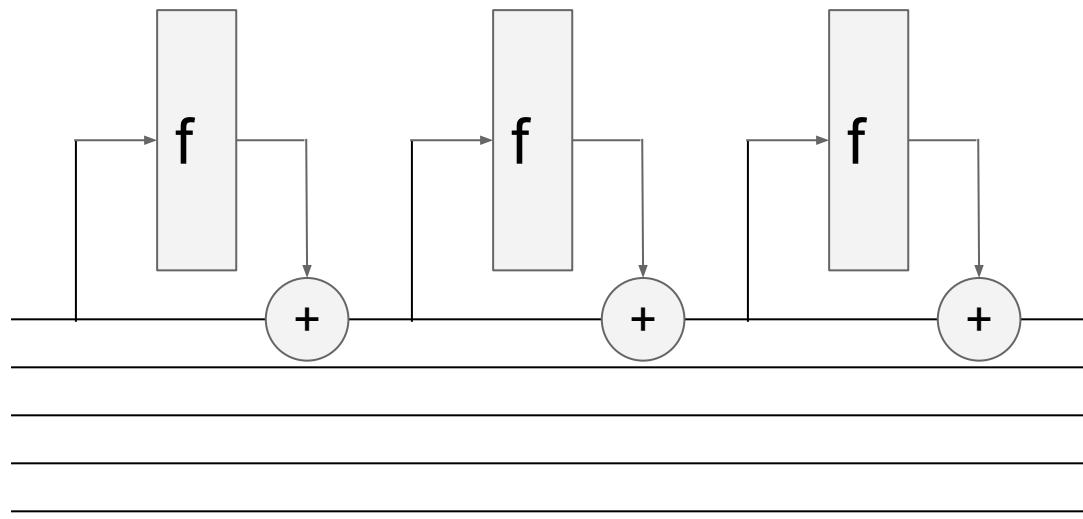
cell
state c



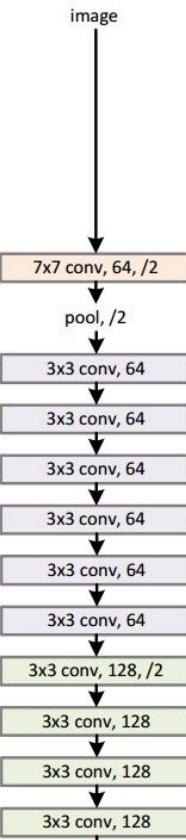
RNN



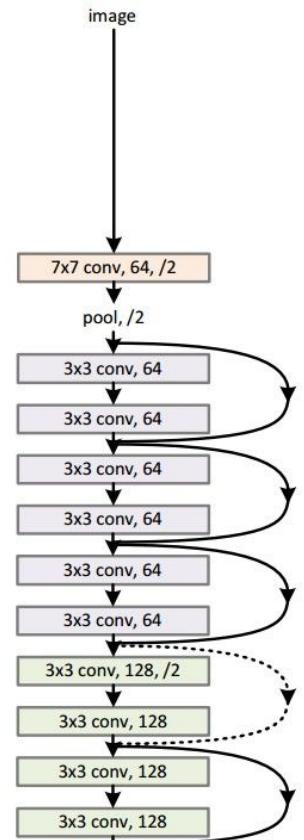
LSTM (ignoring forget gates)



34-layer plain

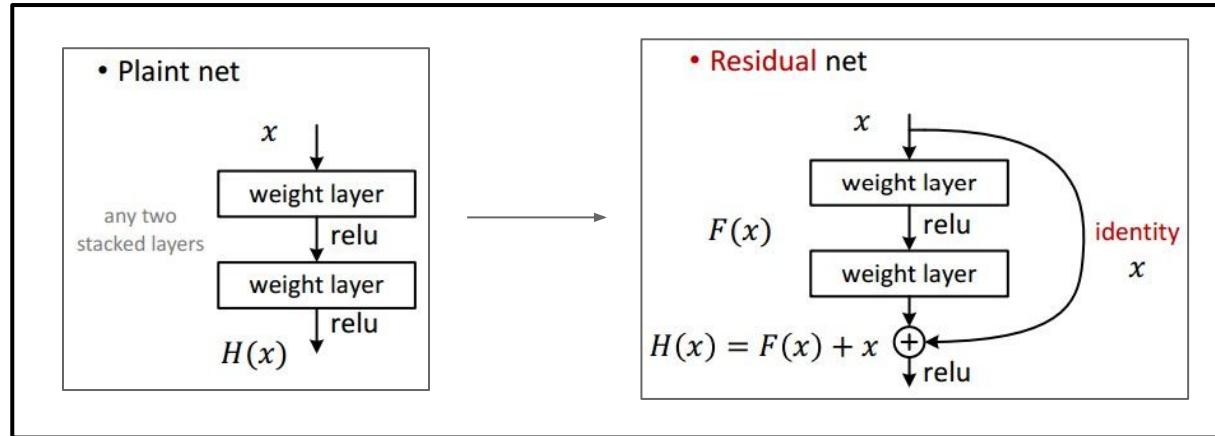


34-layer residual



Recall: “PlainNets” vs. ResNets

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



Understanding gradient flow dynamics

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

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

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

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

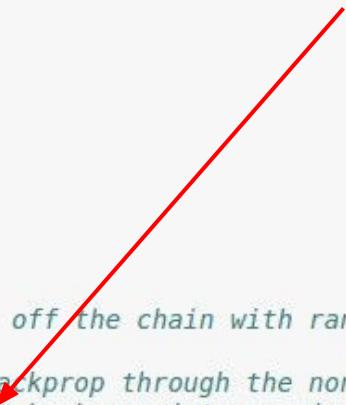
Understanding gradient flow dynamics

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

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

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

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



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

Understanding gradient flow dynamics

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

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

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

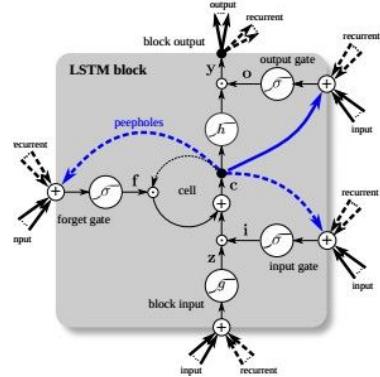
if the largest eigenvalue is > 1 , gradient will explode
if the largest eigenvalue is < 1 , gradient will vanish

can control exploding with gradient clipping
can control vanishing with LSTM

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

LSTM variants and friends

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



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

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

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

MUT1:

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

MUT2:

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

MUT3:

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

Summary

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