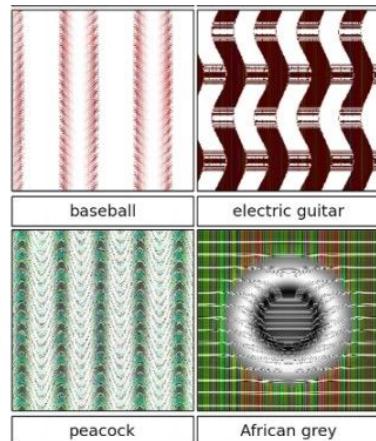
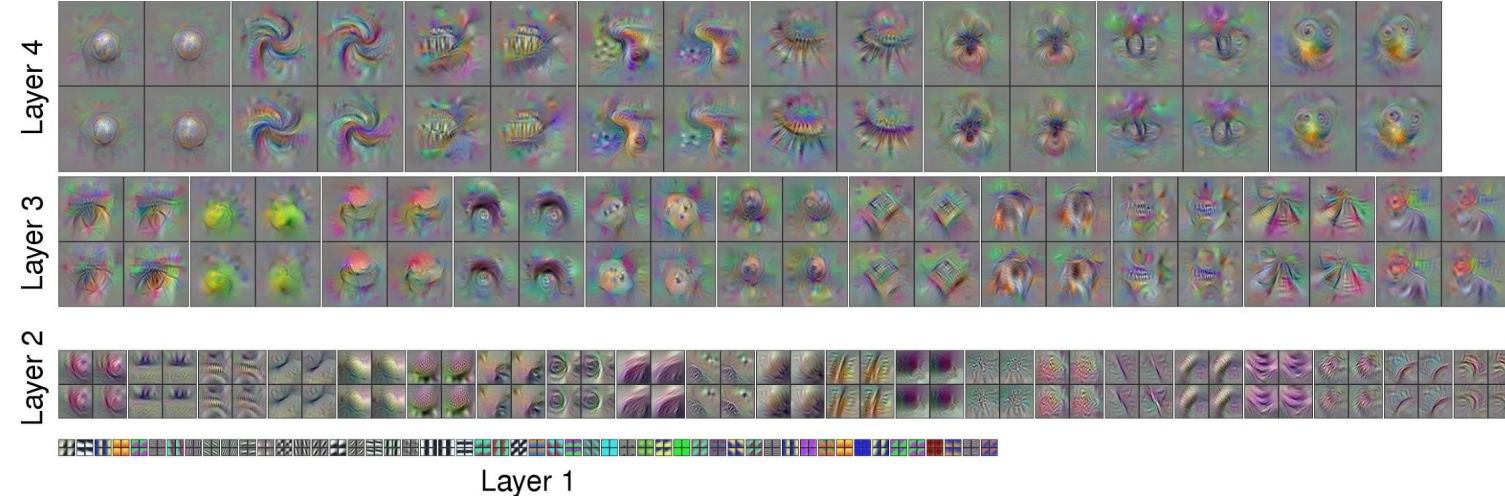
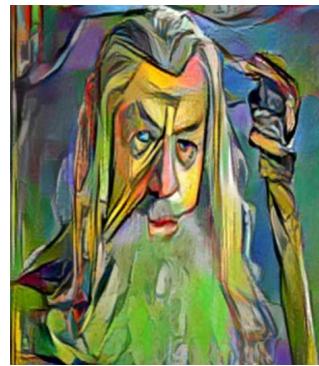
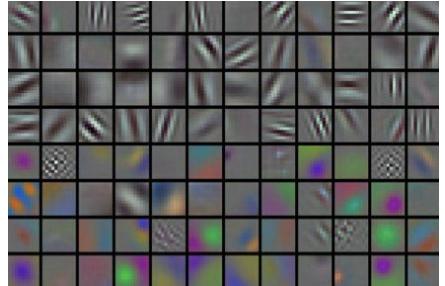


# Lecture 10:

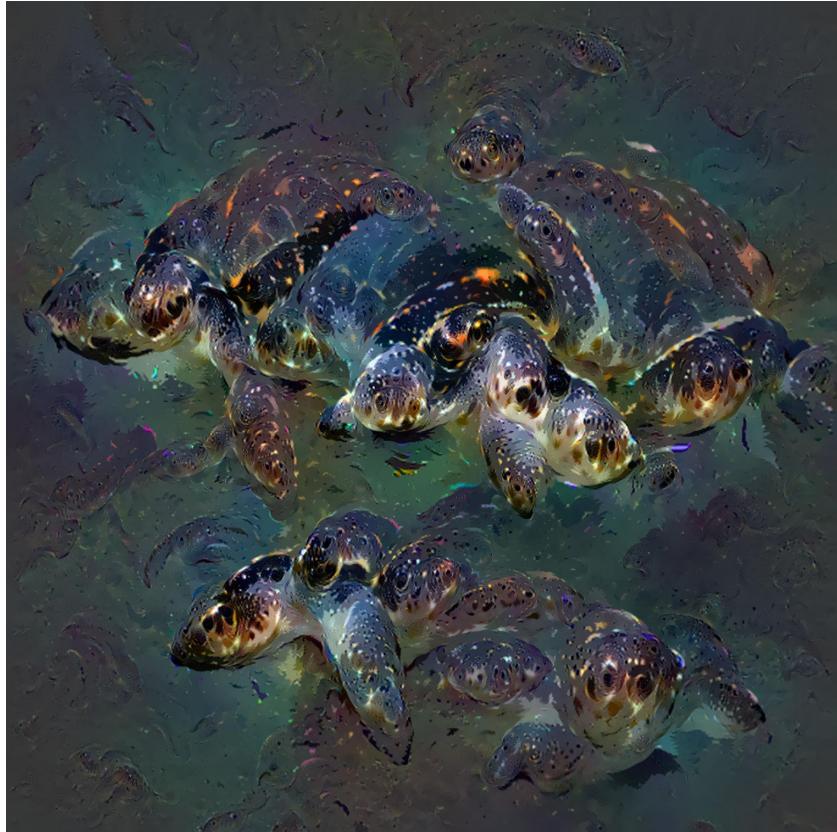
## Recurrent Neural Networks

# Administrative

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



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

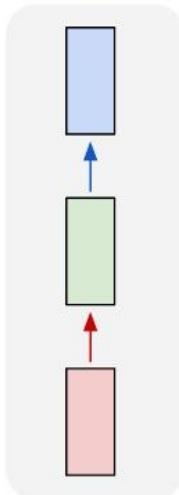


<http://mtyka.github.io/deepdream/2016/02/05/bilateral-class-vis.html>

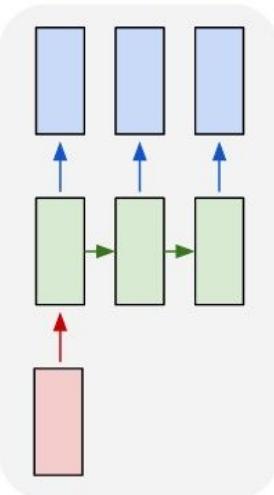


# Recurrent Networks offer a lot of flexibility:

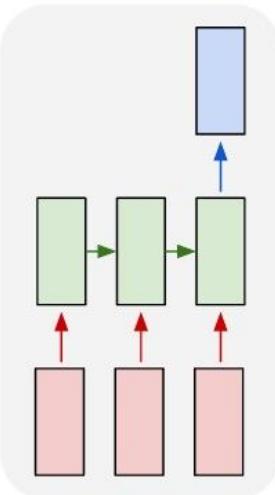
one to one



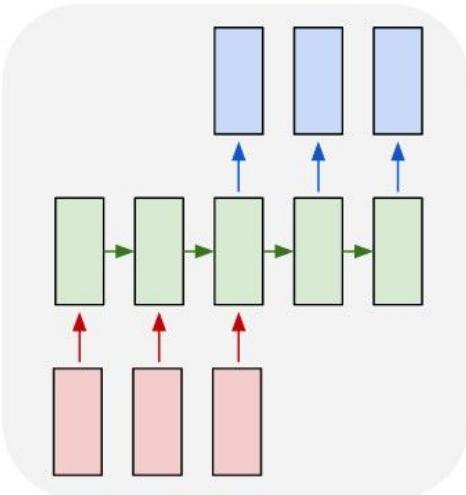
one to many



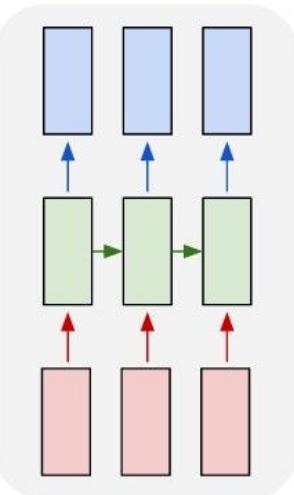
many to one



many to many



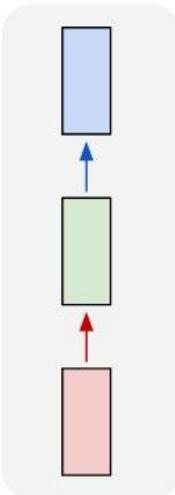
many to many



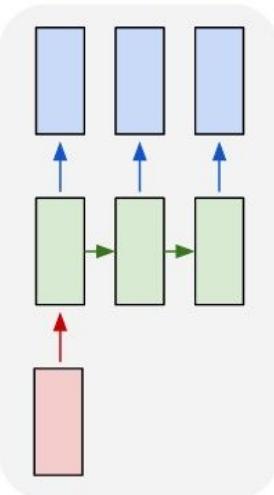
**Vanilla Neural Networks**

# Recurrent Networks offer a lot of flexibility:

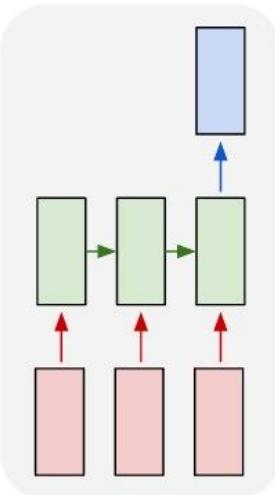
one to one



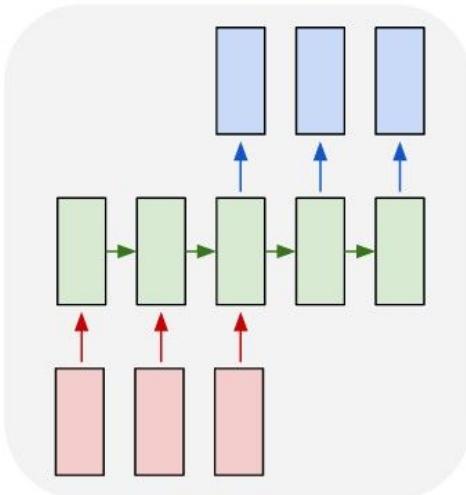
one to many



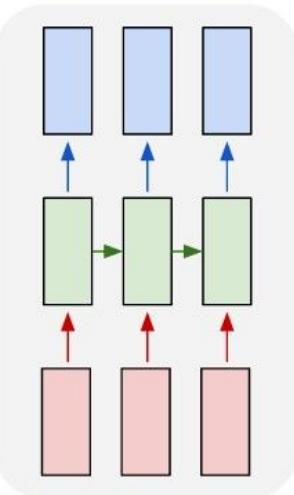
many to one



many to many



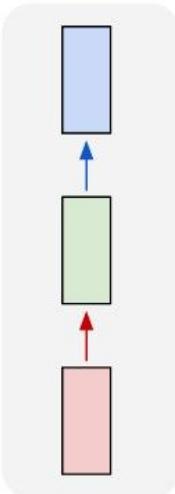
many to many



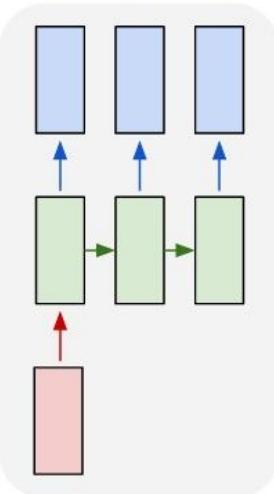
→ e.g. **Image Captioning**  
image -> sequence of words

# Recurrent Networks offer a lot of flexibility:

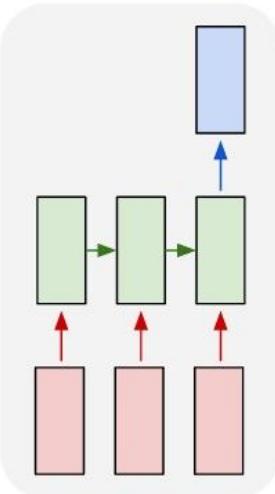
one to one



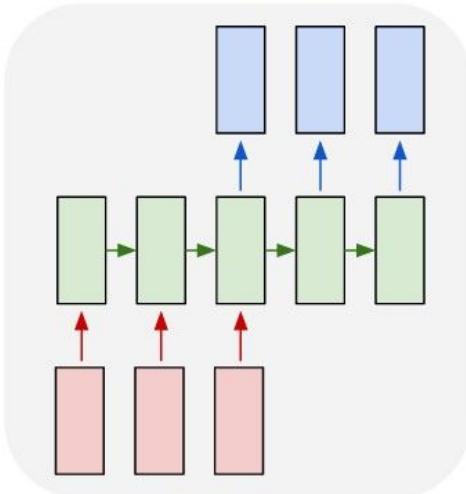
one to many



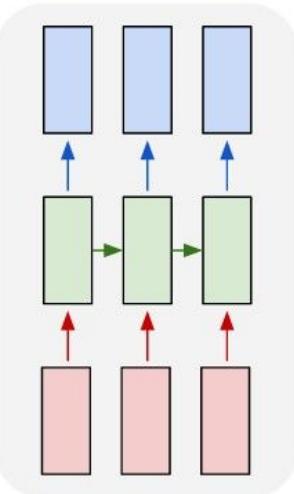
many to one



many to many



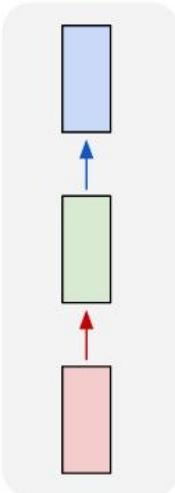
many to many



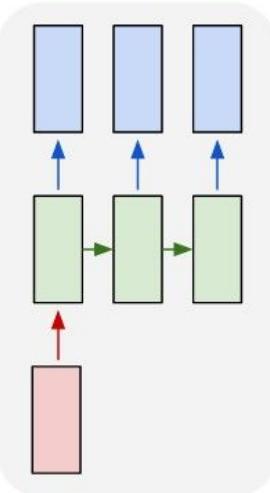
e.g. **Sentiment Classification**  
sequence of words -> sentiment

# Recurrent Networks offer a lot of flexibility:

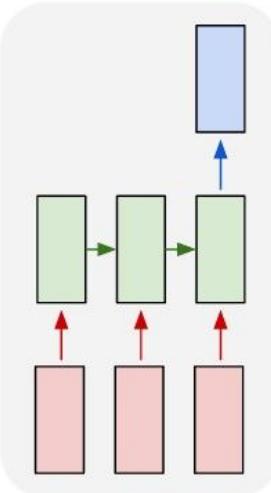
one to one



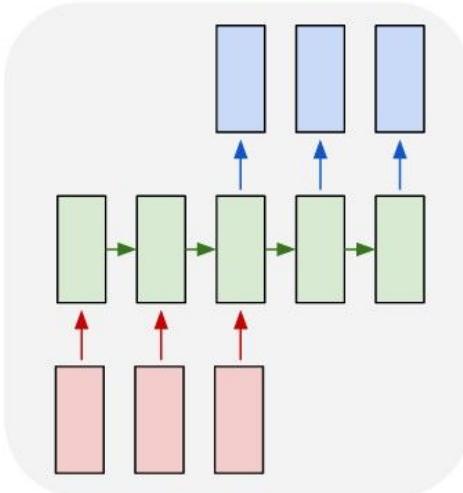
one to many



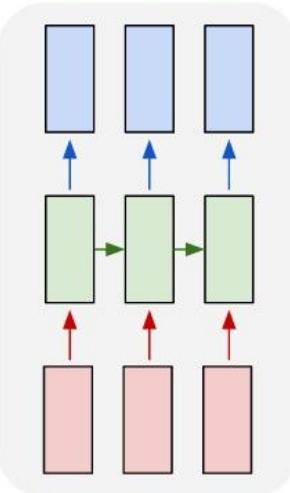
many to one



many to many



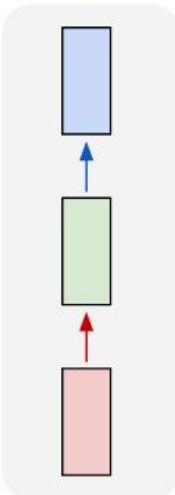
many to many



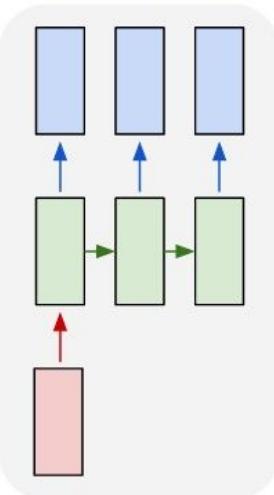
↑  
e.g. **Machine Translation**  
seq of words -> seq of words

# Recurrent Networks offer a lot of flexibility:

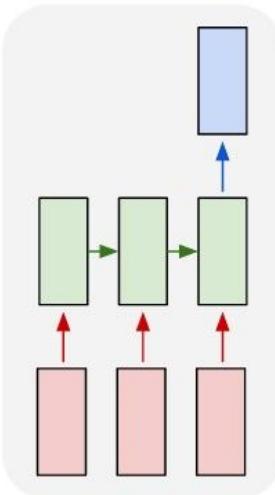
one to one



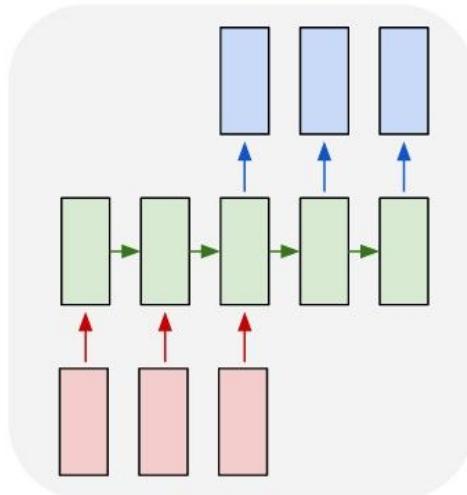
one to many



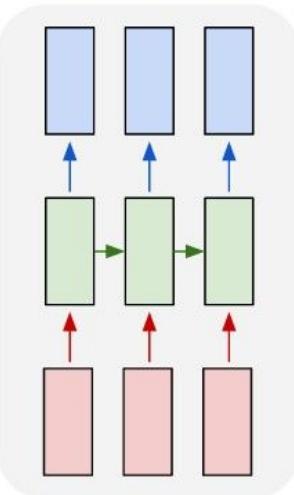
many to one



many to many



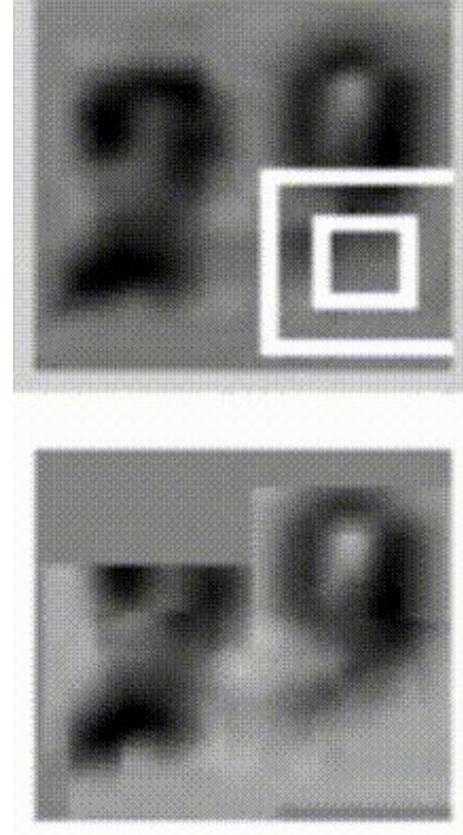
many to many



e.g. Video classification on frame level

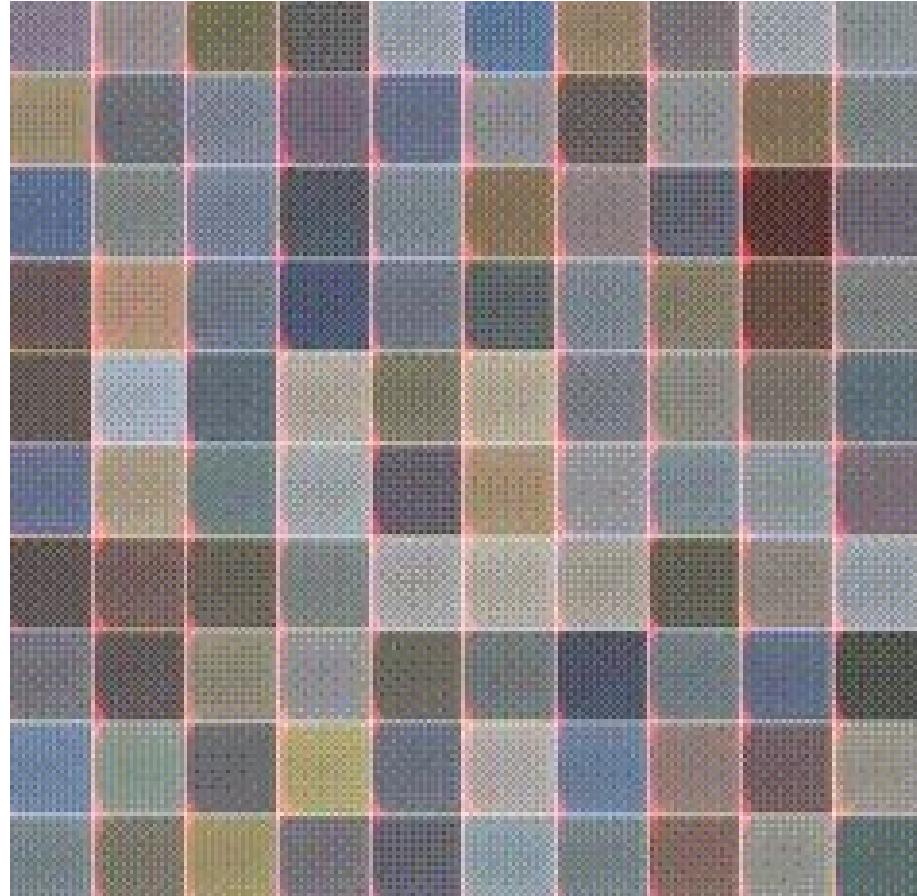
# Sequential Processing of fixed inputs

Multiple Object Recognition with  
Visual Attention, Ba et al.

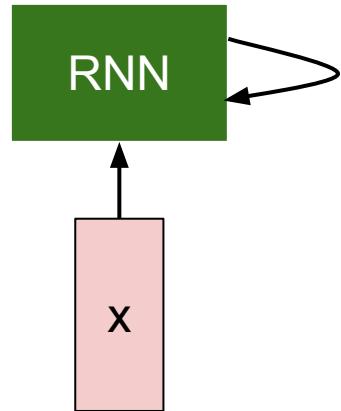


# Sequential Processing of fixed outputs

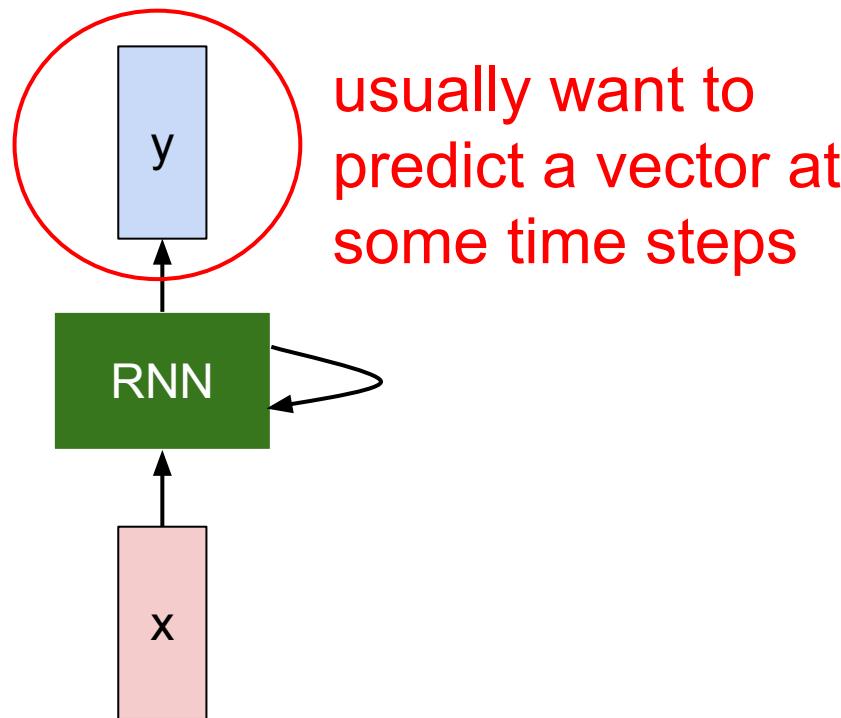
DRAW: A Recurrent  
Neural Network For  
Image Generation,  
Gregor et al.



# Recurrent Neural Network



# Recurrent Neural Network

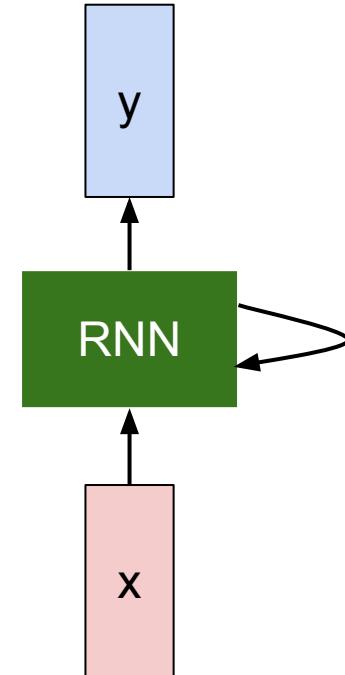


# Recurrent Neural Network

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

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

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

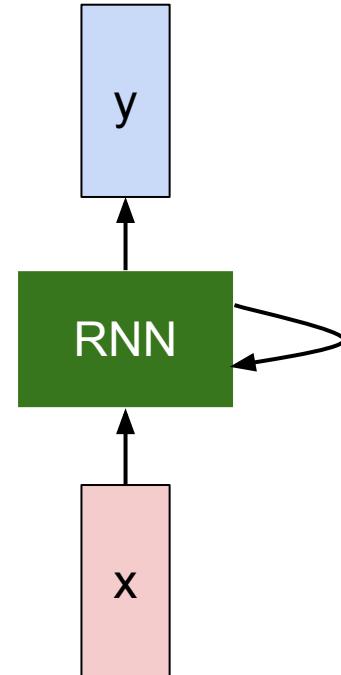


# Recurrent Neural Network

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

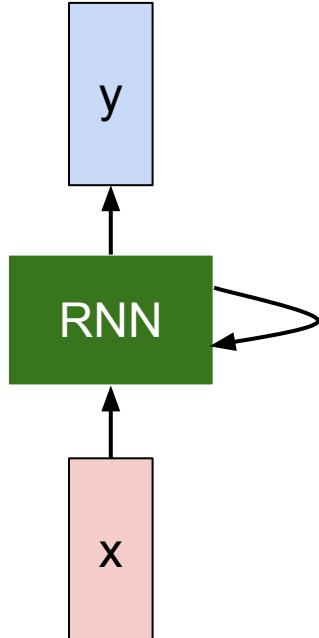
$$h_t = f_W(h_{t-1}, x_t)$$

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  $\mathbf{h}$ :



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



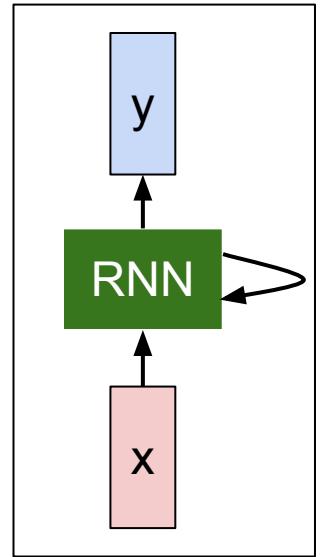
$$\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$$

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

# Character-level language model example

Vocabulary:  
[h,e,l,o]

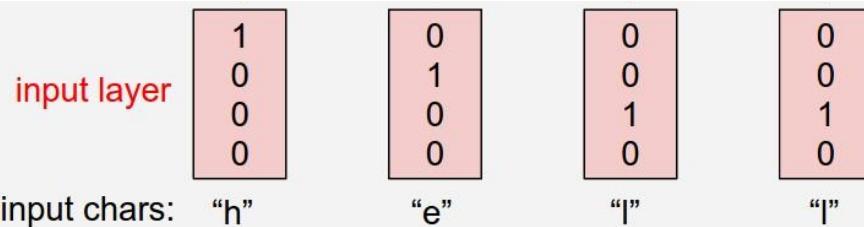
Example training  
sequence:  
“hello”



# Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”

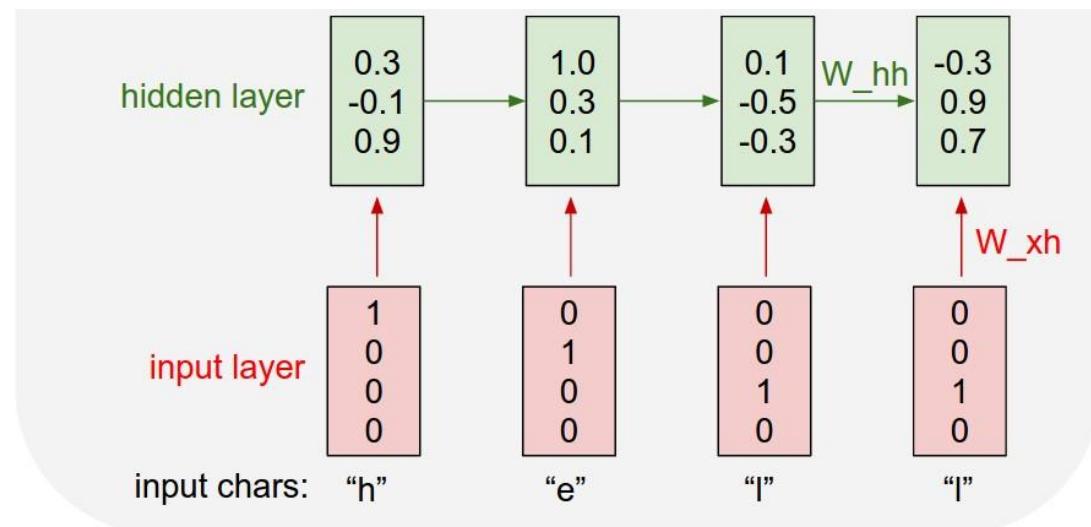


# Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”

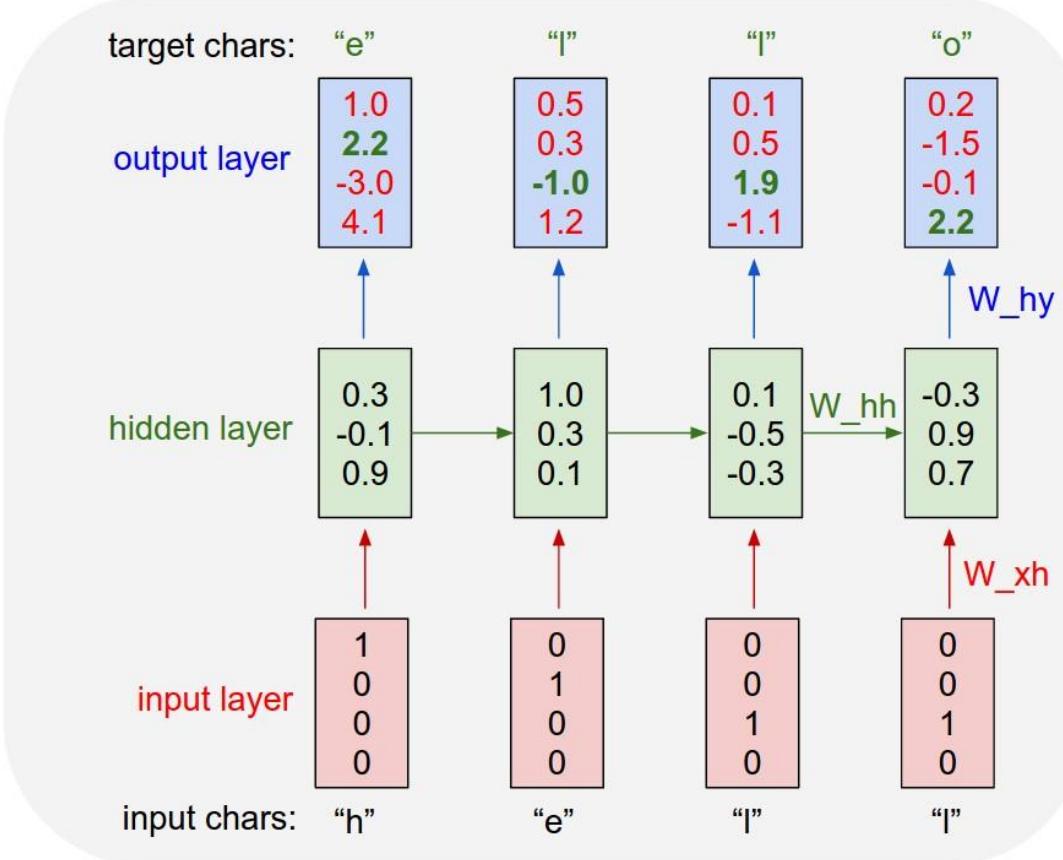
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



# 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))
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 = 20 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # Model parameters
21 #w = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 #wh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 #wy = 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 lossf(inputs, targets, hprev):
28     """ Inputs,targets are both lists of integers.
29
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     hprev = np.copy(hprev)
36
37     # forward pass
38     for t in xrange(len(inputs)):
39         x = np.zeros((vocab_size, 1)) # encode in 1-of-K representation
40         x[inputs[t]:, 1] = 1
41
42         hprev = np.tanh(np.dot(wh, x[1:]) + np.dot(wb, hprev[1:]) + bh) # hidden state
43         y = np.dot(wy, hprev) + by # compute unnormalized log probabilities for next chars
44         ps[t] = np.exp(y[1:]) / np.sum(np.exp(y[1:])) # softmax (cross-entropy loss)
45
46         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
47
48         # backward pass: compute gradients over time steps
49         dprev_dh = np.zeros_like(hprev)
50         dprev_dy = np.zeros_like(dy)
51         dprev_db = np.zeros_like(db)
52
53         dy = np.copy(ps[t])
54         dy[targets[t], 1] -= 1 # backprop into y
55         dh = np.dot(dy, wh.T) # backprop into h
56         dwh = np.dot(dy[1:], x[1:-1].T) # dh backprop through tanh nonlinearity
57         dbh = np.sum(dy[1:], axis=0, keepdims=True) # dh backprop through tanh nonlinearity
58         dprev_dy += np.dot(dy[1:], wh) # backprop into dy
59         dprev_db += db
60
61         for opname in [dprev_dh, dprev_dy, dprev_db]:
62             opname *= np.clip(dy[1:], -5, 5, out=opname) # clip to mitigate exploding gradients
63             opname /= np.sqrt(np.sum(opname**2)) # normalize
64             opname *= dbh # scale by dbh
65             opname *= dwh # scale by dwh
66             opname *= dy # scale by dy
67
68         dprev_dh += dprev_dy
69         dprev_dy += dprev_db
70
71         dh += np.dot(dy[1:], wh) + dprev_dh # backprop into dh
72         dprev_dh = np.dot(dy[1:], x[1:-1].T) # dh backprop through tanh nonlinearity
73         dprev_dy += np.dot(dy[1:], wh) # backprop into dy
74         dprev_db += db
75
76         dprev_dh *= np.tanh(dh)**2 # derivative of tanh
77         dprev_dy *= np.exp(dy[1:]) # derivative of softmax
78         dprev_db *= 1 # derivative of zero-like
79
80         for t in xrange(1, len(inputs)):
81             h = np.tanh(np.dot(w, x) + np.dot(wb, h) + bh)
82             x = np.zeros((vocab_size, 1))
83             x[inputs[t], 1] = 1
84
85             x[seed_ix] = 1
86
87             for t in xrange(1, n):
88                 h = np.tanh(np.dot(w, x) + np.dot(wb, h) + bh)
89                 x = np.zeros((vocab_size, 1))
90                 x[seed_ix] = 1
91
92                 if p < 0.5:
93                     if np.random.rand() < p:
94                         x = np.random.choice(range(vocab_size), p=px, replace=True)
95
96                 x[1:] = 0
97
98             return loss
99
100
101 # We clip gradients from the model, now and then
102 # to prevent exploding gradients
103 n = 100
104 sample_ix = sample(hprev, inputs[0], 200)
105 sample_ix = join(ix_to_char[sample_ix] for ix in sample_ix)
106 print ''.join(sample_ix)
107 print "done"
108
109 # Forward pass: compute raw scores, then softmax loss
110 loss, dxh, dwh, dbh, dyh, dby, hprev = lossf(inputs, targets, hprev)
111 smooth_loss = smooth_loss * 0.999 + loss * 0.001
112 if n % 100 == 0: print "iter %d, loss: %e, smooth loss: %e" % (n, loss, smooth_loss) # print progress
113
114 # Compute gradients, backprop through tanh and softmax
115 for parname, varname in zip([wh, wb, bh, by],
116                            [dwh, dbh, dby, dyh]):
117     varname *= np.clip(dxh, -5, 5, out=varname)
118     varname /= np.sqrt(np.sum(varname**2))
119
120     varname *= dbh # scale by dbh
121     varname *= dby # scale by dby
122
123     varname *= np.tanh(dxh)**2 # derivative of tanh
124     varname *= np.exp(dyh) # derivative of softmax
125
126     varname *= np.zeros_like(dxh) # zero out gradient for dxh
127
128     parname -= learning_rate * varname / np.sqrt(dxh + 1e-8) # update gradient
129
130 p += seq_length # move data pointer
131
132 n += 1 # iteration counter
```

# Data I/O



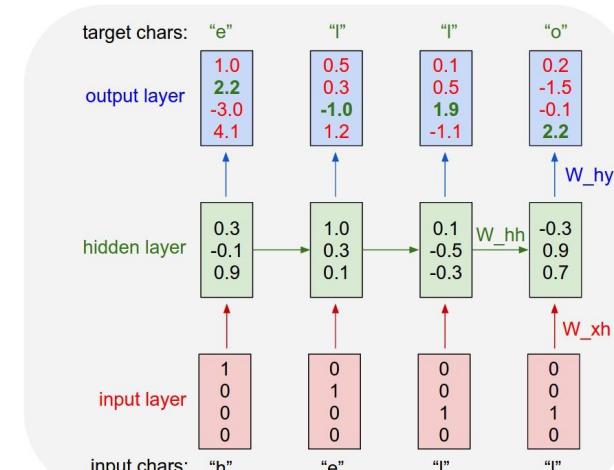
```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
823
824
825
825
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
```

# min-char-rnn.py gist

Initializations

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

## recall



# min-char-rnn.py gist

## 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) # a = b * dot(a, b)
50                 dparam = np.dot(dy, dparam)
51             dy = np.copy(dy[t])
52             dy[targets[t]] -= 1 # backprop into y
53             dh = np.dot(dy, dbh) # dh = dot(dy, dbh)
54             dh += np.dot(dy.T, dWhh) # dh = backprop through tanh nonlinearity
55             dh += dWhh
56             dWhh += np.dot(dh, hs[-1].T)
57             dbh += np.sum(dh, axis=0, keepdims=True)
58             dWhh *= np.tanh(dh)
59             for param in [dWxh, dWhh, dWhy, dbh, dby]:
60                 param = np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
61             if np.isnan(dbh) or np.isnan(dby):
62                 raise ValueError("Nan in dbh or dby, dbh, dby, dy, hprev[inputs-1]")
63             dWxh, dWhh, dWhy, dbh, dby, dy, hprev[inputs-1]
64             dbh, dby, dy = None, None, None
65
66     # Sample a sequence of integers from the model
67     h = np.zeros((hidden_size,)) # h is memory state, seed_ix is used later for first time step
68     if seed_ix is not None:
69         x = np.zeros(vocab_size).astype(np.float)
70         x[seed_ix] = 1
71     for t in xrange(seq_length):
72         h = np.tanh(np.dot(wxh, x) + np.dot(whh, h) + bh)
73         p = np.exp(h)
74         ix = np.argmax(p)
75         x = np.random.choice(range(vocab_size), p=p.ravel())
76         x[seed_ix] = 1
77         x[1:] = 0
78     return ixes
79
80 n, p = 0, 0
81 mem, mnh, mnw, mny = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)
82 mem, dnh, dmhw, dmwh, dmby = np.zeros_like(bh), np.zeros_like(whh), np.zeros_like(by)
83 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
84
85 while True:
86     n, p = 0, 0
87     mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
88     mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
89     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
90
91     while True:
92         # prepare inputs (we're sweeping from left to right in steps seq_length long)
93         if p+seq_length+1 >= len(data) or n == 0:
94             hprev = np.zeros((hidden_size,1)) # reset RNN memory
95             p = 0 # go from start of data
96         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
97         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
98
99         # sample from the model now and then
100        if n % 100 == 0:
101            sample_ix = sample(hprev, inputs[0], 200)
102            txt = ''.join(ix_to_char[ix] for ix in sample_ix)
103            print '----\n%s\n----' % (txt, )
104
105        # forward seq_length characters through the net and fetch gradient
106        loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
107        smooth_loss = smooth_loss * 0.999 + loss * 0.001
108
109        if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
110
111        # perform parameter update with Adagrad
112        for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
113                                      [dWxh, dWhh, dWhy, dbh, dby],
114                                      [mWxh, mWhh, mWhy, mbh, mby]):
115            mem += dparam * dparam
116            param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
117
118            p += seq_length # move data pointer
119            n += 1 # iteration counter
120
```

# Main loop



# min-char-rnn.py gist

```
1  ***
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  ***
5  Import numpy as np
6
7  # Data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(data[1:(1)])
10 vocab_size = len(chars), len(data), len(chars)
11 print("data has %d characters, %d unique." % (data_size, vocab_size))
12 char_to_ix = {c: i for i in range(len(chars))}
13 ix_to_char = {i: c for c in range(len(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
19 # Model parameters
20 dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
21 dbh, mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
22 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
23
24 while True:
25     # prepare inputs (we're sweeping from left to right in steps seq_length long)
26     if p+seq_length+1 >= len(data) or n == 0:
27         hprev = np.zeros((hidden_size,1)) # reset RNN memory
28         p = 0 # go from start of data
29         inputs, targets = [ord(c) for c in data[p:p+seq_length]], [ord(c) for c in data[p+1:p+seq_length+1]]
30
31     # Forward pass
32     for t in range(seq_length):
33         x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
34         x[inputs[t]] = 1
35         hprev = np.tanh(np.dot(wh, x) + np.dot(dbh, hprev) + bh) # hidden state
36         y = np.exp(hprev) # probabilities for next chars
37         py = y / sum(y) # softmax
38         loss += -np.log(py[targets[t]]) # softmax (cross-entropy loss)
39         dx = np.exp(y) # derivatives for next chars
40         dy = np.copy(dy * (1 - py)) # backprop into y
41         dh = np.dot(why.T, dy) + dmem # backprop into h
42         dWxh += np.dot(inputs[t].reshape(1, vocab_size), hprev.reshape(1, hidden_size)) # dh * backprop through tanh nonlinearity
43         dWhh += np.dot(hprev.reshape(1, hidden_size), hprev.reshape(1, hidden_size)) # dh * backprop through tanh nonlinearity
44         dWhy += np.dot(py.reshape(1, vocab_size), np.eye(hidden_size)) # dy * backprop through tanh nonlinearity
45         dmem += np.dot(dbh, dy) # dy * backprop through tanh nonlinearity
46         dbh += np.sum(dy * np.exp(y)) # gradients for hidden bias
47         dmem += np.sum(dy * np.exp(y)) # gradients for memory bias
48         for param in [dWxh, dWhh, dWhy, dbh, dmem]:
49             np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
50             param -= learning_rate * param # update parameter
51             dbh, dmem = np.zeros_like(dbh), np.zeros_like(dmem)
52
53     if n % 100 == 0:
54         sample_ix = sample(hprev, inputs[0], 200)
55         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
56         print('----\n %s \n----' % (txt, ))
57
58     # forward seq_length characters through the net and fetch gradient
59     loss, dWxh, dWhh, dWhy, dbh, dmem, hprev = lossFun(inputs, targets, hprev)
60     smooth_loss = smooth_loss * 0.999 + loss * 0.001
61     if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
62
63     # perform parameter update with Adagrad
64     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
65                                 [dWxh, dWhh, dWhy, dbh, dmem],
66                                 [mWxh, mWhh, mWhy, mbh, mby]):
67         mem += dparam * dparam
68         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
69
70         p += seq_length # move data pointer
71         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
86     while True:
87         # prepare inputs (we're sweeping from left to right in steps seq_length long)
88         if p+seq_length+1 >= len(data) or n == 0:
89             hprev = np.zeros((hidden_size,1)) # reset RNN memory
90             p = 0 # go from start of data
91             inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92             targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94         # sample from the model now and then
95         if n % 100 == 0:
96             sample_ix = sample(hprev, inputs[0], 200)
97             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98             print('----\n %s \n----' % (txt, ))
99
100        # forward seq_length characters through the net and fetch gradient
101        loss, dWxh, dWhh, dWhy, dbh, dmem, hprev = lossFun(inputs, targets, hprev)
102        smooth_loss = smooth_loss * 0.999 + loss * 0.001
103        if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
104
105        # perform parameter update with Adagrad
106        for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
107                                      [dWxh, dWhh, dWhy, dbh, dmem],
108                                      [mWxh, mWhh, mWhy, mbh, mby]):
109            mem += dparam * dparam
110            param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
111
112            p += seq_length # move data pointer
113            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 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 = [char_to_ix[c] for c in data[p+1:p+seq_length+1]] # fetch sequence of targets
33
34     x, hs, ys, ps = o, O, O, O
35     hprev = np.copy(hprev)
36     loss = 0
37
38     for t in range(seq_length):
39         # Unpack current input vector from 1-of-k representation
40         x_t = np.zeros((vocab_size,1)) # encode in 1-of-k representation
41         x_t[char_to_ix[inputs[t]]] = 1
42         h_t1 = np.tanh(np.dot(wh, x_t) + np.dot(dWhh, hs[-1]) + bh) # hidden state
43         y_t = np.dot(Wxh, h_t1) + b # compute output
44         ps_t = np.exp(y_t) / np.sum(np.exp(y_t)) # probabilities for next chars
45         loss += -np.log(ps_t[targets[t]]) # softmax (cross-entropy loss)
46
47         # Backprop through tanh nonlinearity
48         dWxh += np.outer(x_t, ps_t * (1 - ps_t))
49         dWhh += np.outer(hs[-1], ps_t * (1 - ps_t))
50         dbh += np.sum(ps_t * (1 - ps_t))
51
52         dy_t = np.zeros_like(y_t)
53         dy_t[targets[t]] = -1 # backward into y
54         dh_t1 = np.dot(Why, dy_t) + dWhy # backprop into h
55         dh_t1 *= np.tanh(h_t1) # backprop through tanh nonlinearity
56         dWxh += np.outer(x_t, dh_t1)
57         dWhh += np.outer(hs[-1], dh_t1)
58         dbh += np.sum(dh_t1)
59         dWhy += np.outer(h_t1, dy_t)
60
61         for dparam in [dWxh, dWhh, dbh, dWhy]:
62             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
63             dparam /= seq_length # average gradient
64             dbh /= seq_length # average gradient
65
66         if n % 100 == 0:
67             sample_ix = sample(hprev, inputs[0], 200)
68             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
69             print '----\n%s\n----' % (txt, )
70
71     # forward seq_length characters through the net and fetch gradient
72     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
73     smooth_loss = smooth_loss * 0.999 + loss * 0.001
74
75     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
76
77     # perform parameter update with Adagrad
78     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
79                                 [dWxh, dWhh, dWhy, dbh, dby],
80                                 [mWxh, mWhh, mWhy, mbh, mby]):
81         mem += dparam * dparam
82         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
83
84         p += seq_length # move data pointer
85
86         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
86     while True:
87
88         # prepare inputs (we're sweeping from left to right in steps seq_length long)
89         if p+seq_length+1 >= len(data) or n == 0:
90             hprev = np.zeros((hidden_size,1)) # reset RNN memory
91             p = 0 # go from start of data
92             inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
93             targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
94
95         # sample from the model now and then
96         if n % 100 == 0:
97             sample_ix = sample(hprev, inputs[0], 200)
98             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
99             print '----\n%s\n----' % (txt, )
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
117         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 = len(data)
12 print("data has %d characters, %d unique." % (data_size, vocab_size))
13 char_to_ix = {ch:i for i in range(len(chars))}
14 ix_to_char = {i:ch for ch in range(len(chars))} # 14
15
16 # Hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 20 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 model_params = {}
22
23 def softmax(x):
24     e = np.exp(x - np.max(x))
25     return e / e.sum()
26
27 inputs, targets = two lists of integers.
28 hprev is RNN array of initial hidden state
29 returns the loss, gradients on model parameters, and last hidden state
30
31
32 xs, hs, ys, ps = O, O, O, O
33 h0 = np.copy(hprev)
34 loss = 0
35 for t in range(seq_length):
36     # encode in 1-of-k representation
37     x = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
38     x[0][char_to_ix[inputs[t]]] = 1
39
40     h0_t = np.tanh(np.dot(wih, x) + np.dot(bh, h0[-1]) + bh) # hidden state
41     y = np.dot(why, h0_t) + by # output
42     yprob = softmax(y) # probabilities for next chars
43     ps[t] = np.exp(y) / np.sum(np.exp(y)) # softmax (cross-entropy loss)
44     loss += -np.log(ps[t][target[0]]) # softmax (cross-entropy loss)
45
46     # backprop through tanh nonlinearity
47     dprev_dh = np.dot(dh0, np.zeros_like(wih)) # dh0 = gradient from loss
48     dprev_db = np.zeros_like(bh), np.zeros_like(by), np.zeros_like(why)
49     dprev_dy = np.zeros_like(y)
50     for i in range(len(inputs)):
51         dy[i] = np.copy(ps[t])
52         dy[targets[i]] -= 1 # backprop into y
53         dprev_dy += dy[i]
54         dprev_db += np.dot(dy[i], bh) # dh = backprop through tanh nonlinearity
55         dprev_dy *= np.dot(dy[i], wih.T) # dh = backprop through tanh nonlinearity
56         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
57         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
58         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
59         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
60         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
61         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
62         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
63         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
64         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
65         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
66         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
67         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
68         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
69         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
70         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
71         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
72         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
73         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
74         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
75         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
76         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
77         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
78         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
79         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
80         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
81         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
82         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
83         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
84         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
85         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
86         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
87         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
88         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
89         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
90         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
91         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
92         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
93         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
94         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
95         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
96         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
97         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
98         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
99         dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
100        dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
101        dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
102        dprev_dy *= np.dot(dy[i], x) # dh = backprop through tanh nonlinearity
103
104    # forward seq_length characters through the net and fetch gradient
105    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
106    smooth_loss = smooth_loss * 0.999 + loss * 0.001
107
108    if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
109
110
111    # perform parameter update with Adagrad
112    for param, dparam, mem in zip([Wxh, Whh, why, bh, by],
113                                 [dWxh, dWhh, dWhy, dbh, dby],
114                                 [mWxh, mWhh, mWhy, mbh, mby]):
115
116        mem += dparam * dparam
117        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
118
119        p += seq_length # move data pointer
120
121        n += 1 # iteration counter
122
```

# Main loop

```
81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85
86 while True:
87
88     # prepare inputs (we're sweeping from left to right in steps seq_length long)
89     if p+seq_length+1 >= len(data) or n == 0:
90
91         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
92         p = 0 # go from start of data
93
94     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
95     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
96
97
98     # sample from the model now and then
99     if n % 100 == 0:
100
101         sample_ix = sample(hprev, inputs[0], 200)
102         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
103         print '----\n%s\n----' % (txt, )
104
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
113    # perform parameter update with Adagrad
114    for param, dparam, mem in zip([Wxh, Whh, why, bh, by],
115                                 [dWxh, dWhh, dWhy, dbh, dby],
116                                 [mWxh, mWhh, mWhy, mbh, mby]):
117
118        mem += dparam * dparam
119        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
120
121        p += seq_length # move data pointer
122
123        n += 1 # iteration counter
124
```

# min-char-rnn.py gist

## Loss function

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

```
***  
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 lists of integers.  
    hprev is Hx1 array of initial hidden state  
    returns the loss, gradients on model parameters, and last hidden state  
    """  
  
    xs, hs, ys, ps = {}, {}, {}, {}  
    hs[-1] = np.copy(hprev)  
    loss = 0  
    for t in xrange(len(inputs)):  
        xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation  
        xs[t][inputs[t]] = 1  
        wht = np.tanh(np.dot(wh, xs[t]) + np.dot(bh, hs[t-1]) + bh) # hidden state  
        whyt = np.dot(why, wht) + by # unnormalized log probabilities for next chars  
        ps[t] = np.exp(whyt) / np.sum(np.exp(whyt)) = softmax (cross-entropy loss)  
        loss += -np.log(ps[t][targets[t],0])  
        dy = np.zeros_like(whyt)  
        dy[targets[t]] = -1 # backprop into y  
        dy = np.dot(dy.T, wh) # backprop through tanh nonlinearity  
        dwh = np.zeros_like(wh) # gradients for wh  
        dwh += np.dot(xs[t].T, dy) # backprop through dot product  
        dbh = np.zeros_like(bh) # gradients for bh  
        dbh += np.sum(dy, axis=0) # backprop through sum  
        dby = np.zeros_like(by) # gradients for by  
        dby += dy # backprop through dot product  
        dh = np.dot(why.T, dy) + dwh + dbh # backprop into h  
        dwh += np.dot(dh, xs[t]) # backprop through tanh nonlinearity  
        ddbh = np.zeros_like(dbh) # gradients for dbh  
        ddbh += np.sum(dh, axis=0) # backprop through sum  
        dby += dh # backprop through dot product  
        for dparam in [dwh, dbh, dby]:  
            np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
        for dparam in [dwh, dbh, dby]:  
            dparam -= learning_rate * dparam # update parameter  
            dparam = np.clip(dparam + 1e-8, 0, 1) # adagrad update  
  
    return loss, ps[t][targets[t],0]  
  
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  
    """  
  
    x = np.zeros((vocab_size, 1))  
    x[seed_ix] = 1  
  
    for t in xrange(n):  
        h = np.tanh(np.dot(wh, x) + np.dot(bh, h) + bh)  
        p = np.exp(why * np.dot(h, wh))  
        ix = np.argmax(np.random.multinomial(1, p.ravel()))  
        x[seed_ix] = 0  
        x[i] = 1  
  
    return ix  
  
n, p, t, b  
meh, meth, why = np.zeros_like(bh), np.zeros_like(wh), np.zeros_like(why)  
mem, memh, memy = np.zeros_like(bh), np.zeros_like(wh), np.zeros_like(why)  
smooth_loss = 0  
for i in range(seq_length):  
    if i == 0:  
        h = np.zeros(hidden_size)  
    else:  
        h = mem  
    if i < seq_length - 1:  
        targets.append(ix_to_char[ix])  
    inputs.append(ix_to_char[ix])  
  
    # Forward pass: compute scores through the net and fetch gradient  
    loss, dxh, dmbh, dmh, dyh, dby, dprev = lossFun(inputs, targets, hprev)  
    smooth_loss = smooth_loss * 0.999 + loss * 0.001  
    if i % 100 == 0:  
        print 'iter %d, loss: %f' % (i, smooth_loss) # print progress  
  
    # Backward pass: compute gradients going backwards  
    dwhh, dwh, dwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)  
    dbh, dyb = 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  
        dwh += np.dot(dy, hs[t].T)  
        dbh += dy  
        dh = np.dot(why.T, dy) + dhnext # backprop into h  
        ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity  
        dbh += ddraw  
        dwhh += np.dot(ddraw, xs[t].T)  
        dwhh += np.dot(dh, hs[t-1].T)  
        dhnext = np.dot(why.T, ddraw)  
  
    for dparam in [dwhh, dwh, dwhy, dbh, dyb, dprev]:  
        np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients  
        dparam -= learning_rate * dparam # update parameter  
        dparam = np.clip(dparam + 1e-8, 0, 1) # adagrad update  
  
    return loss, dxh, dmbh, dmh, dyh, 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.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros(hidden_size, 1) # hidden bias
25 by = np.zeros(vocab_size, 1) # output bias
26
27 by = np.zeros(vocab_size, 1) # output bias

```

```

28 def lossFun(inputs, targets, hprev):
29     """"
30     inputs,targets are both list of integers.
31     hprev is Hx1 array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34
35     xs, hs, ys, ps = {}, {}, {}, {}
36     hs[-1] = np.copy(hprev)
37     loss = 0
38
39     # forward pass
40     for t in xrange(len(inputs)):
41         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
42         xs[t][inputs[t]] = 1
43
44         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
45
46         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
47         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
48
49         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
50
51     # backward pass: compute gradients going backwards
52     dchh, dbh, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
53     dch, dby = np.zeros_like(bh), np.zeros_like(by)
54     dhnext = np.zeros_like(hs[0])
55
56     for t in reversed(xrange(len(inputs)-1)):
57         dy = np.copy(ps[t])
58         dy[targets[t]] -= 1 # backprop into y
59         dhy = np.dot(Why.T, dy) # backprop through Why
60         dbh += np.sum(dhy) # backprop through bh
61         dch = np.dot(Wxh.T, dy) + dhnext # backprop through tanh nonlinearity
62         dch += dby # backprop through by
63         dchh += np.dot(Whh.T, dch) # backprop through Whh
64         dch += np.sum(dch) # backprop through hidden state
65         dbh += np.sum(dch) # backprop through bh
66         dhnext = dch # store for next time step
67
68     for opname in [dchh, dbh, dhy, dby]:
69         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
70         opname *= learning_rate # scale down gradients
71
72     return loss, hs[-1]
73
74 def sample(hprev, seed_ix, n):
75     """"
76     sample a sequence of integers from the model
77     h is memory state, seed_ix is seed letter for first time step
78     """
79     x = np.zeros((vocab_size, 1))
80     x[seed_ix] = 1
81
82     for t in xrange(n):
83         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
84         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
85         ix = np.random.choice(range(vocab_size), p=p.ravel())
86         x[0] = 0 # reset previous hidden state
87         x[1] = 1 # start with first character
88
89     return ix
90
91 n, p = 0, 0
92
93 mean, std = np.zeros(vocab_size), np.zeros(vocab_size)
94 mean_dot, std_dot = np.zeros(vocab_size), np.zeros(vocab_size)
95 mean_dot_dot, std_dot_dot = np.zeros(vocab_size), np.zeros(vocab_size)
96
97 smooth_mean, smooth_std = np.zeros(vocab_size), np.zeros(vocab_size)
98 smooth_mean_dot, smooth_std_dot = np.zeros(vocab_size), np.zeros(vocab_size)
99 smooth_mean_dot_dot, smooth_std_dot_dot = np.zeros(vocab_size), np.zeros(vocab_size)
100
101 # forward pass: compute gradients through the net and fetch gradient
102 loss, dchh, dbh, dhy, dby, hprev = lossFun(inputs, targets, hprev)
103 smooth_loss = smooth_loss * 0.999 + loss * 0.001
104 smooth_mean = smooth_mean * 0.999 + mean * 0.001
105 smooth_std = smooth_std * 0.999 + std * 0.001
106
107 if n % 100 == 0: print 'iter %d, loss: %f, smooth loss: %f' % (n, loss, smooth_loss)
108
109 # backward pass: compute gradients through the net and fetch gradient
110 for parname, arr in zip([wh, whh, why, bh, by],
111                         [dchh, dbh, dhy, dby, dhnext]):
112     arr *= learning_rate # scale down gradients
113     arr /= np.sqrt(arr + 1e-8) # adaptive learning rate
114
115     arr += np.dot(darr, darr.T) # update gradients
116
117     parname -= learning_rate * arr # update parameters
118
119 p = seq_length
120 n = 1 # iteration counter

```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

```

27 def lossFun(inputs, targets, hprev):
28     """"
29     inputs,targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33
34     xs, hs, ys, ps = {}, {}, {}, {}
35     hs[-1] = np.copy(hprev)
36     loss = 0
37
38     # forward pass
39     for t in xrange(len(inputs)):
40         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
41         xs[t][inputs[t]] = 1
42
43         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
44
45         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
46         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
47
48         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
49
50     # backward pass: compute gradients going backwards
51     dchh, dbh, dhy = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
52     dch, dby = np.zeros_like(bh), np.zeros_like(by)
53     dhnext = np.zeros_like(hs[0])
54
55     for t in reversed(xrange(len(inputs)-1)):
56         dy = np.copy(ps[t])
57         dy[targets[t]] -= 1 # backprop into y
58         dhy = np.dot(Why.T, dy) # backprop through Why
59         dbh += np.sum(dhy) # backprop through bh
60         dch = np.dot(Wxh.T, dy) + dhnext # backprop through tanh nonlinearity
61         dch += dby # backprop through by
62         dchh += np.dot(Whh.T, dch) # backprop through Whh
63         dch += np.sum(dch) # backprop through hidden state
64         dbh += np.sum(dch) # backprop through bh
65         dhnext = dch # store for next time step
66
67     for opname in [dchh, dbh, dhy, dby]:
68         np.clip(opname, -5, 5, out=opname) # clip to mitigate exploding gradients
69         opname *= learning_rate # scale down gradients
70
71     return loss, hs[-1]
72
73 def sample(hprev, seed_ix, n):
74     """"
75     sample a sequence of integers from the model
76     h is memory state, seed_ix is seed letter for first time step
77     """
78     x = np.zeros((vocab_size, 1))
79     x[seed_ix] = 1
80
81     for t in xrange(n):
82         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
83         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
84         ix = np.random.choice(range(vocab_size), p=p.ravel())
85         x[0] = 0 # reset previous hidden state
86         x[1] = 1 # start with first character
87
88     return ix
89
90 n, p = 0, 0
91
92 mean, std = np.zeros(vocab_size), np.zeros(vocab_size)
93 mean_dot, std_dot = np.zeros(vocab_size), np.zeros(vocab_size)
94 mean_dot_dot, std_dot_dot = np.zeros(vocab_size), np.zeros(vocab_size)
95
96 smooth_mean, smooth_std = np.zeros(vocab_size), np.zeros(vocab_size)
97 smooth_mean_dot, smooth_std_dot = np.zeros(vocab_size), np.zeros(vocab_size)
98 smooth_mean_dot_dot, smooth_std_dot_dot = np.zeros(vocab_size), np.zeros(vocab_size)
99
100 # forward pass: compute gradients through the net and fetch gradient
101 loss, dchh, dbh, dhy, dby, hprev = lossFun(inputs, targets, hprev)
102 smooth_loss = smooth_loss * 0.999 + loss * 0.001
103 smooth_mean = smooth_mean * 0.999 + mean * 0.001
104 smooth_std = smooth_std * 0.999 + std * 0.001
105
106 if n % 100 == 0: print 'iter %d, loss: %f, smooth loss: %f' % (n, loss, smooth_loss)
107
108 # backward pass: compute gradients through the net and fetch gradient
109 for parname, arr in zip([wh, whh, why, bh, by],
110                         [dchh, dbh, dhy, dby, dhnext]):
111     arr *= learning_rate # scale down gradients
112     arr /= np.sqrt(arr + 1e-8) # adaptive learning rate
113
114     arr += np.dot(darr, darr.T) # update gradients
115
116     parname -= learning_rate * arr # update parameters
117
118 p = seq_length
119 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
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 wh = np.random.rand(hidden_size, size=vocab_size)*0.01 # input to hidden
24 bh = np.random.rand(hidden_size, size=hidden_size)*0.01 # hidden to hidden
25 why = np.random.rand(vocab_size, size=hidden_size)*0.01 # hidden to output
26 bh = np.zeros(hidden_size, size=1) # hidden bias
27 by = np.zeros(vocab_size, size=1) # output bias
28
29 by += np.zeros(vocab_size, size=1) # without bias
30
31 def lossFun(inputs, targets, hprev):
32     """ inputs,targets are both lists of integers.
33     hprev is Hx1 array of initial hidden state
34     returns the loss, gradients on model parameters, and last hidden state
35     """
36
37     xs, hs, ys, ps, o, O, D, h0 = np.copy(hprev)
38
39     for t in xrange(1, len(inputs)):
40         x = np.zeros(vocab_size, size=1) # encode in 1-of-k representation
41         x[inputs[t]] = 1
42
43         h0_t1 = np.tanh(np.dot(wh, x[1:]) + np.dot(bh, hs[t-1]) + bh) # hidden state
44         y = np.dot(why.T, h0_t1) # calculate output
45         v = np.exp(y) / np.sum(np.exp(y)) # probabilities for next chars
46         ps[t] = -np.log(v[targets[t]]) / np.sum(np.exp(y)) # softmax (cross-entropy loss)
47
48     loss = -ps[0]
49
50     dnh, dwh, dbh, why = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(bh),
51     db, dy = np.zeros_like(bh), np.zeros_like(by),
52     dhnext = np.zeros_like(hs[0])
53
54     for t in reversed(xrange(len(inputs))):
55         dy = np.copy(ps[t])
56         dy[targets[t]] -= 1 # backprop into y
57         dwhy += np.dot(dy, hs[t].T)
58         dyb = dy * why
59         dbh += np.dot(dyb, h0_t1.T)
60         dwh += np.dot(dyb, xs[t].T)
61         dhnext = np.dot(why.T, dhnext) # backprop into h
62         dh = np.tanh(dhnext) * dhnext # backprop through tanh nonlinearity
63         dnh += np.dot(dy, dh) # backprop into h
64         dnh += np.dot(dy, np.dot(dbh, h0_t1.T))
65         dbh += np.dot(dy, np.dot(dnh, h0_t1.T))
66         dwh += np.dot(dy, np.dot(dbh, xs[t].T))
67
68     for dparam in [dnh, dwh, dbh, why]:
69         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
70
71     return loss, dnh, dwh, dbh, why, dh, dy, hs[len(inputs)-1]
72
73 def sample(seed_ix, n):
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
78     x = np.zeros(vocab_size, size=1)
79     x[seed_ix] = 1
80
81     for t in xrange(n):
82         h = np.tanh(np.dot(wh, x) + np.dot(bh, h) + bh)
83         p = np.exp(y) / np.sum(np.exp(y))
84         ix = np.random.choice(range(vocab_size), p=p.ravel())
85         x[1:] = 0
86         x[ix] = 1
87
88    return ix
89
90 n, p = 0
91
92 mem, mnh, mbh, mhy = np.zeros_like(wh), np.zeros_like(wh), np.zeros_like(bh),
93 mem0, mnh0, mbh0, mhy0 = np.zeros_like(wh), np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(why)
94 smooth_loss = 0
95
96 while True:
97     if p == seq_length or (n == len(data) - seq_length):
98         if p == seq_length:
99             print 'done'
100         p = sample(seq_length, 200)
101         mem = np.zeros_like(wh)
102         mnh = np.zeros_like(bh)
103         mbh = np.zeros_like(bh)
104         mhy = np.zeros_like(why)
105
106     inputs = [char_to_ix[ch] for ch in data[n:(n+seq_length)]]
107     targets = [char_to_ix[ch] for ch in data[(n+seq_length):(n+seq_length)+1]]
108
109     a = np.zeros((vocab_size, hidden_size), dtype=np.float)
110     b = np.zeros((hidden_size, vocab_size), dtype=np.float)
111     c = np.zeros((hidden_size, hidden_size), dtype=np.float)
112     d = np.zeros((vocab_size, 1), dtype=np.float)
113
114     sample_ix = sample(p, 200)
115     sample_ix = jacobian_ix_to_char[sample_ix]
116     print " ".join(sample_ix)
117
118     # Forward pass: compute activations through the net and fetch gradient
119     loss, dnh, dwh, dbh, dy, hprev = lossFun(inputs, targets, hprev)
120     smooth_loss = smooth_loss * 0.999 + loss * 0.001
121     if n % 100 == 0: print 'iter %d, loss: %f, smooth loss: %f' % (n, loss, smooth_loss)
122
123     # Backward pass: compute gradients going backwards
124     for param, dparam in zip([wh, bh, why, db], [dwh, dbh, dhy, dyb]):
125         np.clip(dparam, -5, 5, out=dparam)
126         param += -learning_rate * dparam / np.sqrt(dparam**2 + 1e-8) # adaptive update
127
128     n += 1 # iteration counter
129
130 p = seq_length # move data pointer
131
132 n = 1 # iteration counter

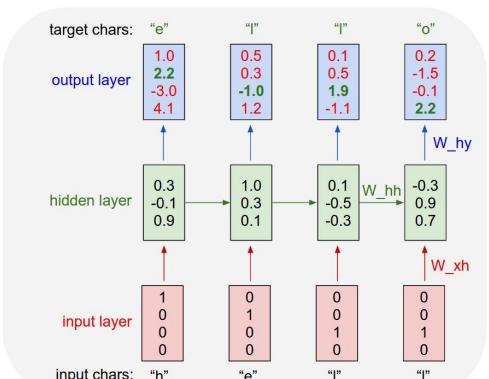
```

```

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, dyb = np.zeros_like(bh), np.zeros_like(by)
47 dhnext = np.zeros_like(hs[0])
48
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     dyb = dy * Why
54     dbh += np.dot(dyb, h0_t1.T)
55     dwhh += np.dot(dyb, xs[t].T)
56     dhnext = np.dot(Why.T, dhnext) # backprop into h
57     dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
58     dbh += dhraw
59     dwhh += np.dot(dhraw, Wxh.T)
60     dwhy += np.dot(dhraw, Whh.T)
61
62     for dparam in [dwxh, dwhh, dwhy, dbh, dyb]:
63         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
64
65     return loss, dwxh, dwhh, dwhy, dbh, dyb, hs[len(inputs)-1]

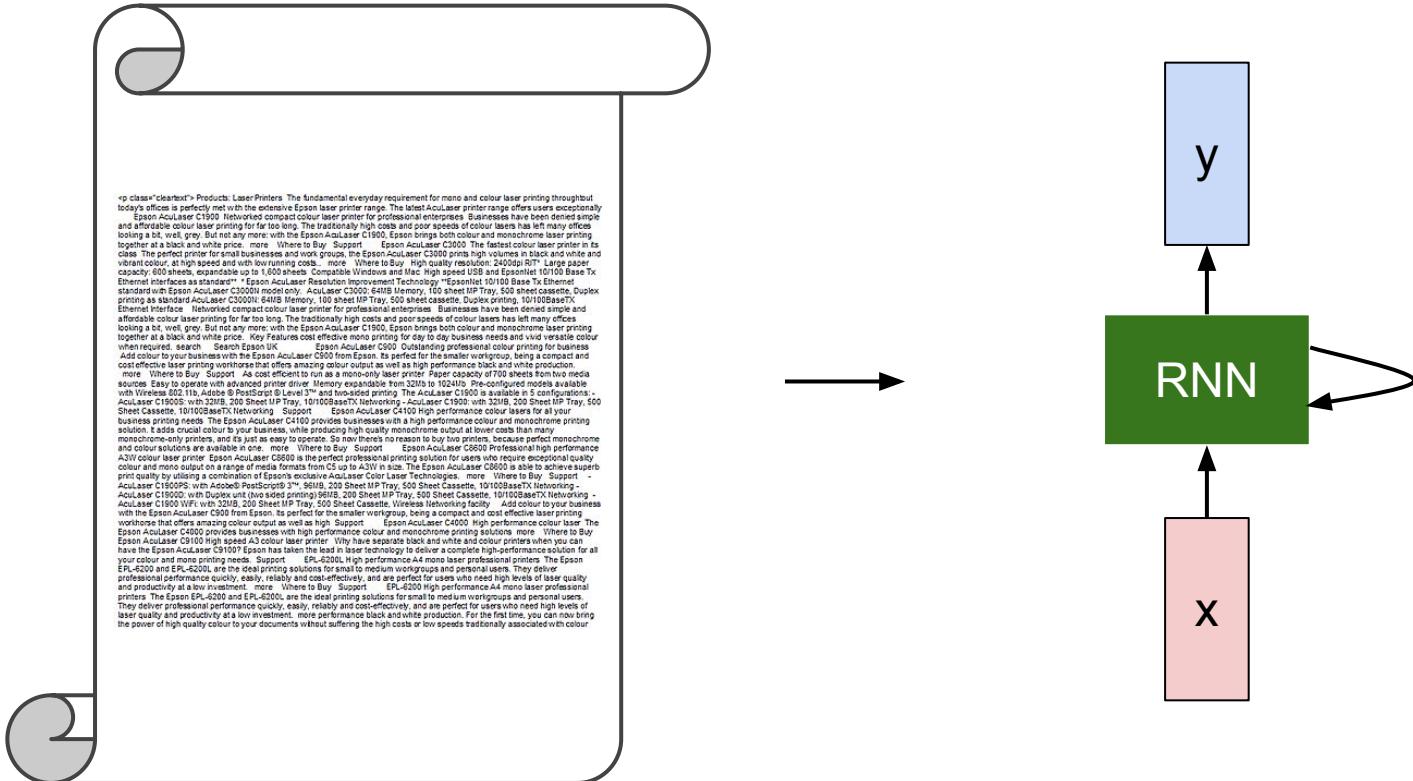
```

recall:



# min-char-rnn.py gist

```
def sample(h, seed_ix, n):
    """
    sample a sequence of integers from the model
    h is memory state, seed_ix is seed letter for first time step
    """
    x = np.zeros((vocab_size, 1))
    x[seed_ix] = 1
    ixes = []
    for t in xrange(n):
        h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
        y = np.dot(Why, h) + by
        p = np.exp(y) / np.sum(np.exp(y))
        ix = np.random.choice(range(vocab_size), p=p.ravel())
        x = np.zeros((vocab_size, 1))
        x[ix] = 1
        ixes.append(ix)
    return ixes
```



## Sonnet 116 – Let me not ...

*by William Shakespeare*

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

at first:

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

↓ train more

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

↓ train more

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

↓ train more

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

PANDARUS:

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

Second Senator:

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

DUKE VINCENTIO:

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

Second Lord:

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

Clown:

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

VIOLA:

I'll drink it.

VIOLA:

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

KING LEAR:

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

# open source textbook on algebraic geometry

The Stacks Project

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

Browse chapters

| Part          | Chapter                 | online                 | TeX source          | view pdf            |
|---------------|-------------------------|------------------------|---------------------|---------------------|
| Preliminaries | 1. Introduction         | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 2. Conventions          | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 3. Set Theory           | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 4. Categories           | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 5. Topology             | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 6. Sheaves on Spaces    | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 7. Sites and Sheaves    | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 8. Stacks               | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 9. Fields               | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |
|               | 10. Commutative Algebra | <a href="#">online</a> | <a href="#">tex</a> | <a href="#">pdf</a> |

Parts

1. [Preliminaries](#)
2. [Schemes](#)
3. [Topics in Scheme Theory](#)
4. [Algebraic Spaces](#)
5. [Topics in Geometry](#)
6. [Deformation Theory](#)
7. [Algebraic Stacks](#)
8. [Miscellany](#)

Statistics

The Stacks project now consists of

- o 455910 lines of code
- o 14221 tags (56 inactive tags)
- o 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  $\text{Sch}_{fppf}$  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  $S_{\text{Zar}}$ , 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  $\delta_{n+1}$  is a scheme over  $S$ .  $\square$

*Proof.* Omitted. □

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

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

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

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

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

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

io: remove unnecessary check for multi-block transfers when doing direct memory access

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;
}
```

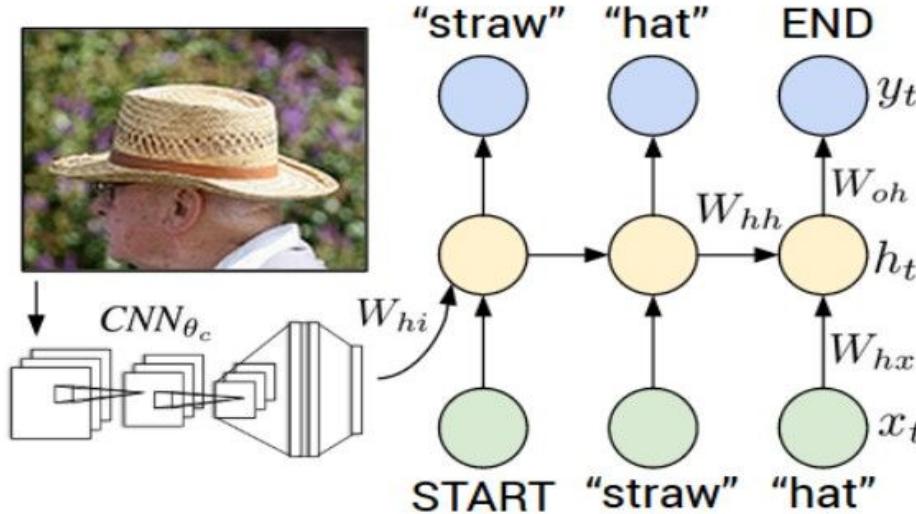
quote/comment cell

# 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.

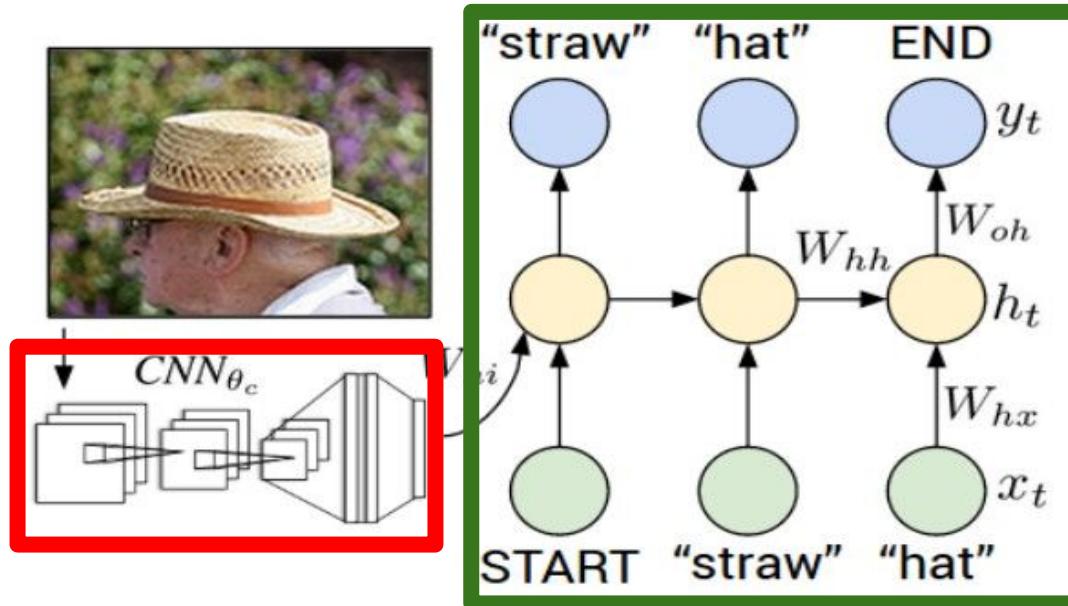
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

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



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

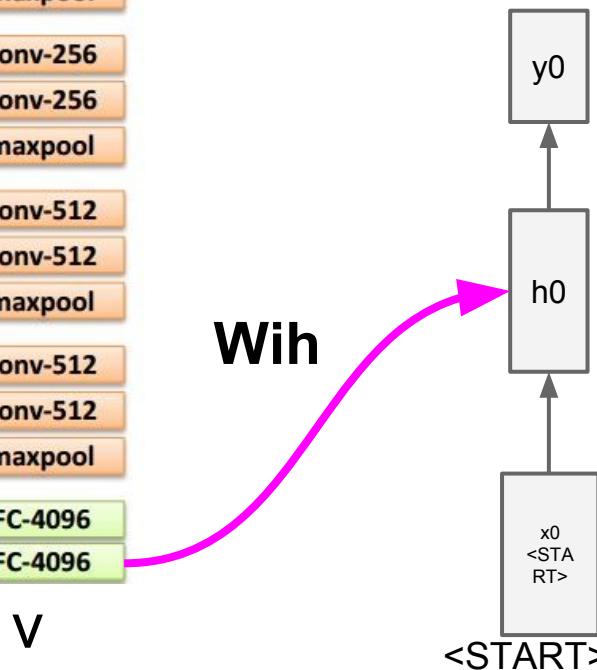
FC-4096



<START>



test image



**before:**

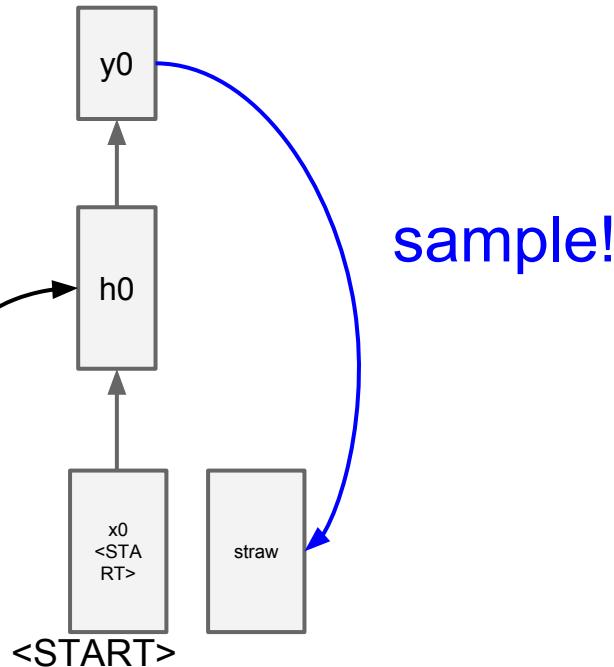
$$h = \tanh(Wxh * x + Whh * h)$$

**now:**

$$h = \tanh(Wxh * x + Whh * h + Wihs * v)$$



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

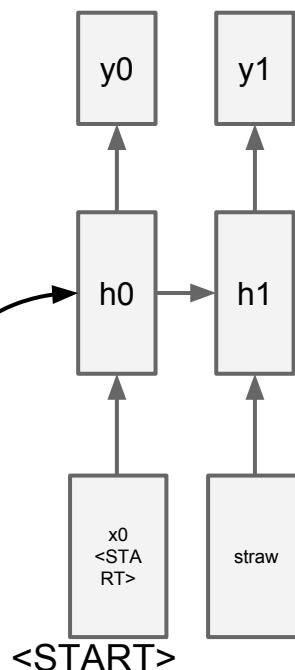
conv-512

conv-512

maxpool

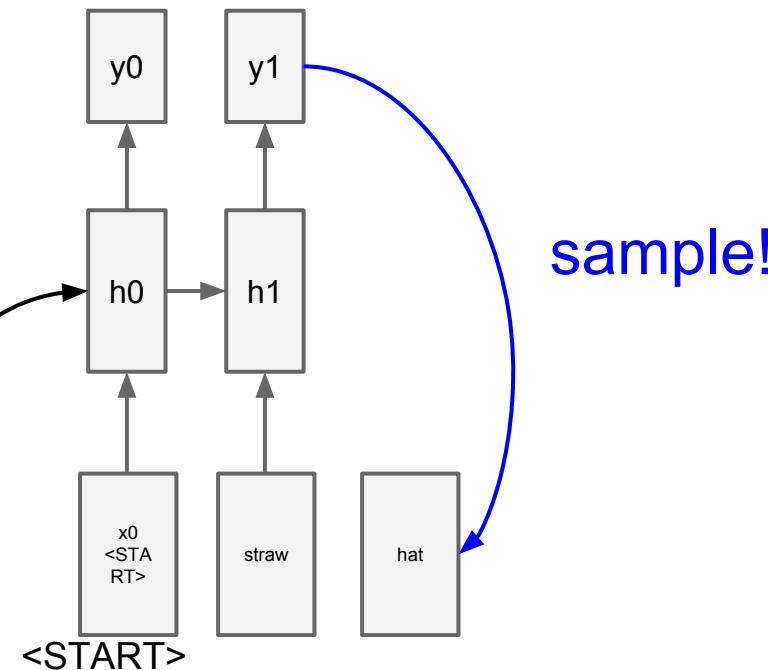
FC-4096

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

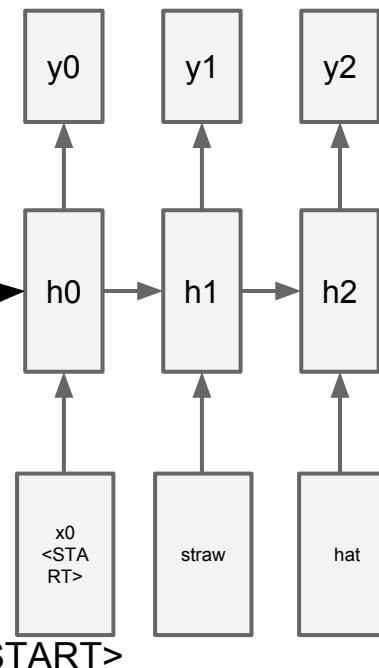
conv-512

conv-512

maxpool

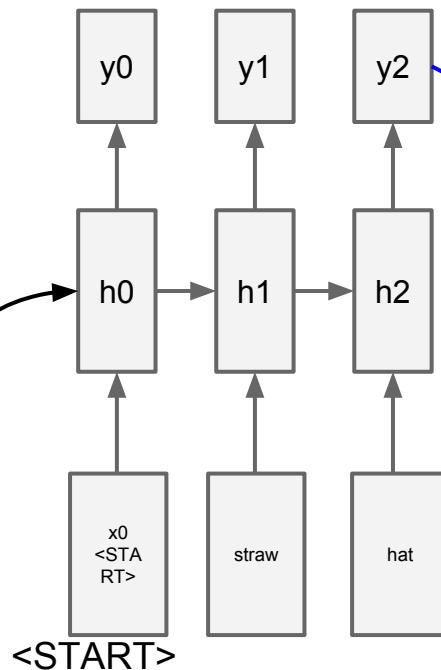
FC-4096

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](http://mscoco.org)

currently:  
~120K images  
~5 sentences each



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



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



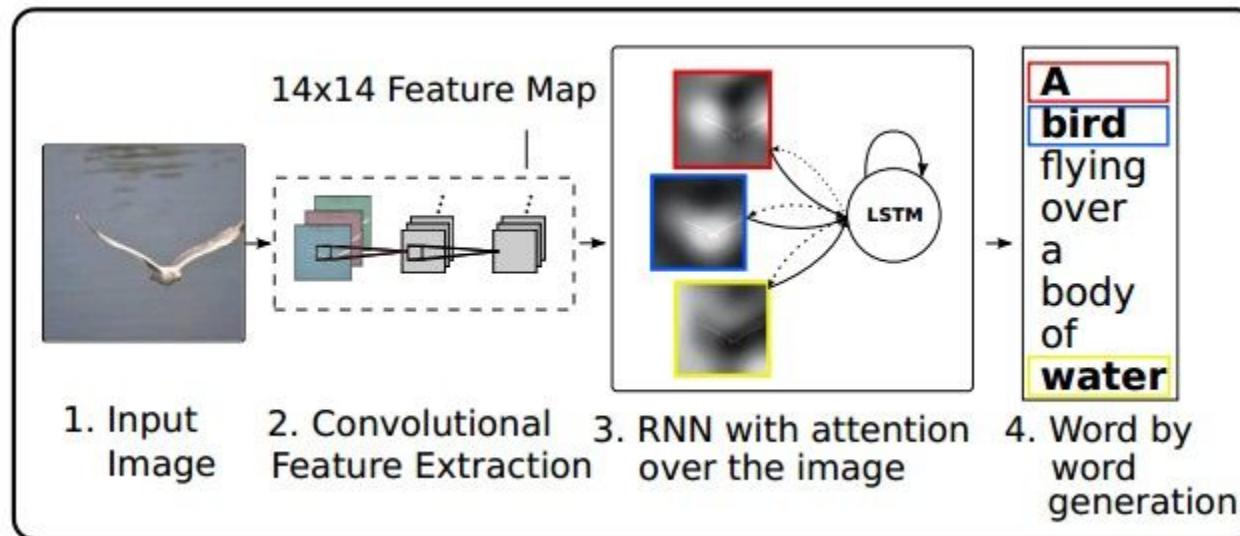
"a woman holding a teddy bear in front of a mirror."



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

# Preview of fancier architectures

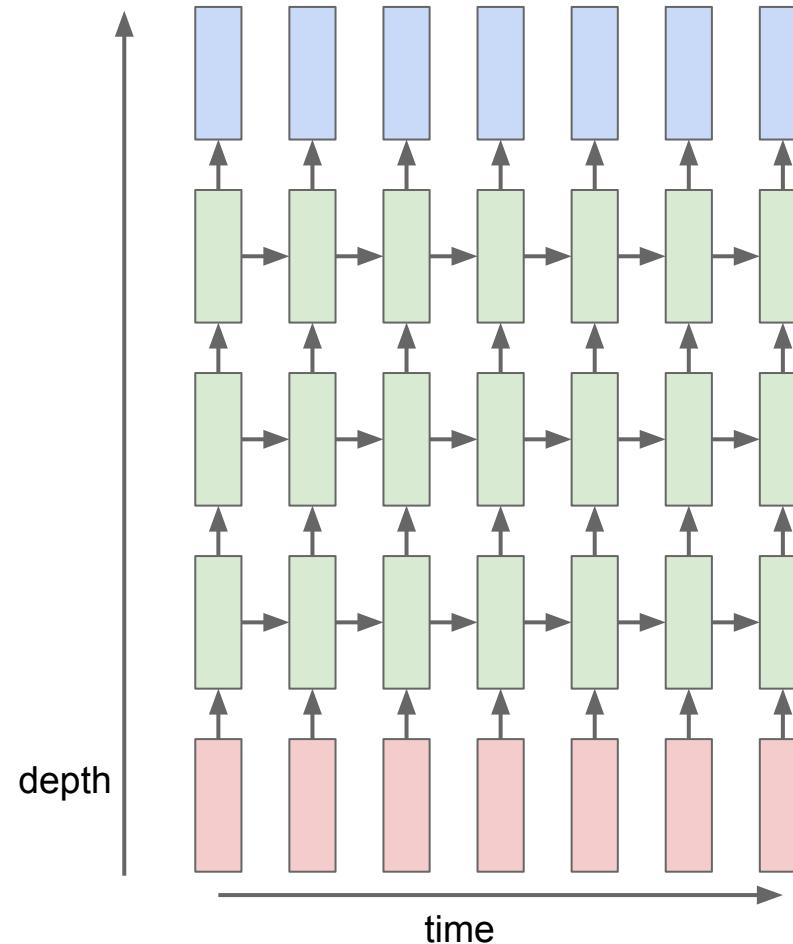
RNN attends spatially to different parts of images while generating each word of the sentence:



# 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:

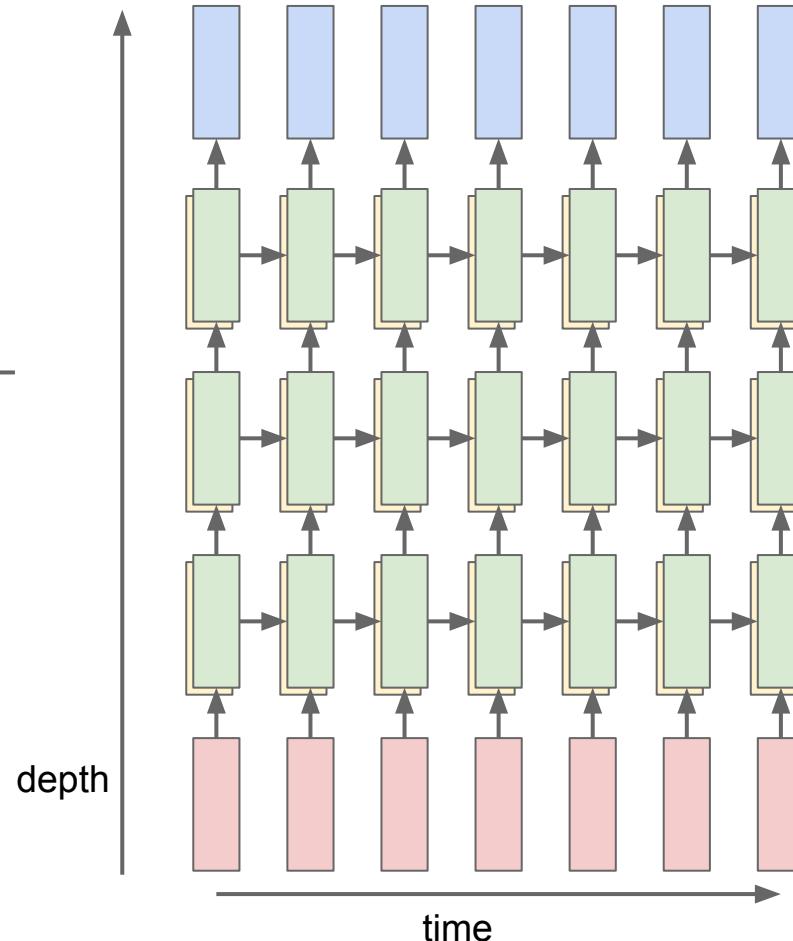
$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

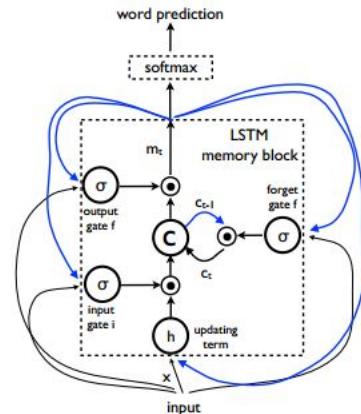
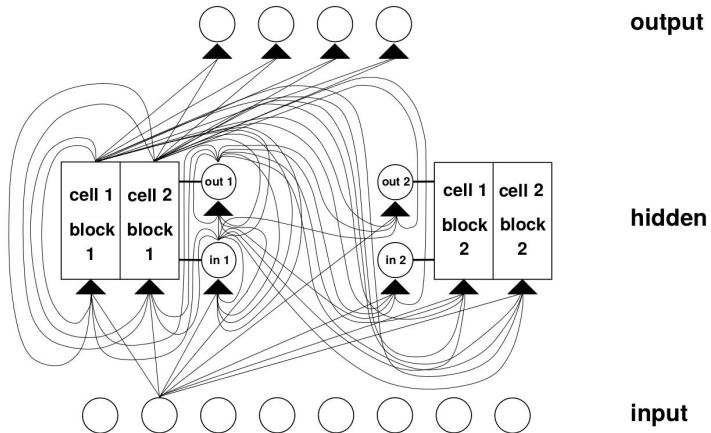
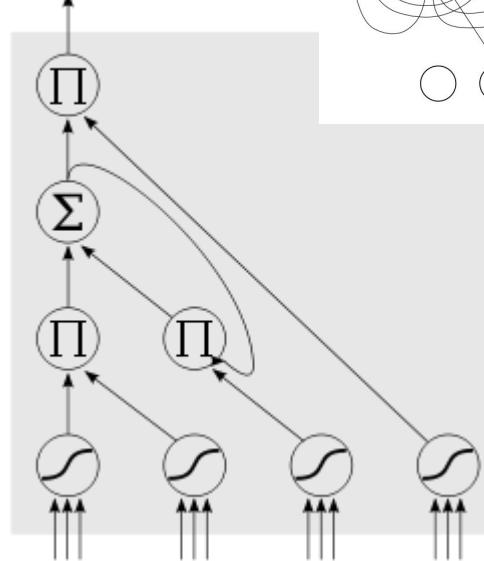
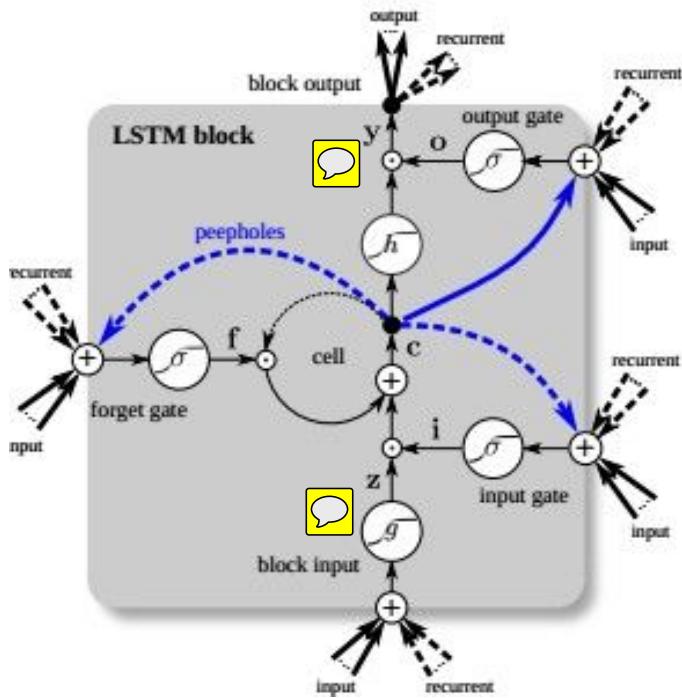
# LSTM:

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

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

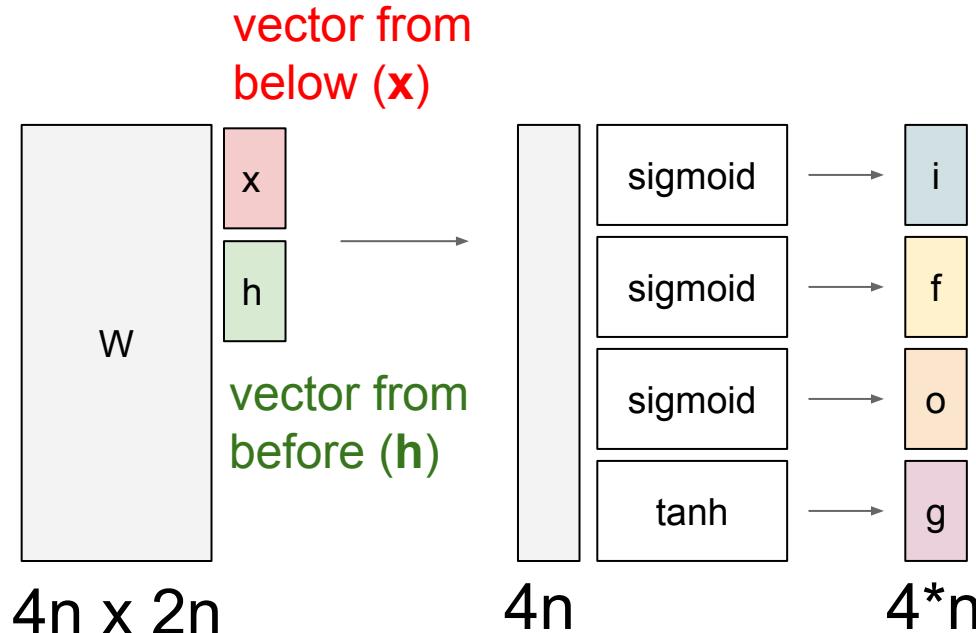


# LSTM



# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

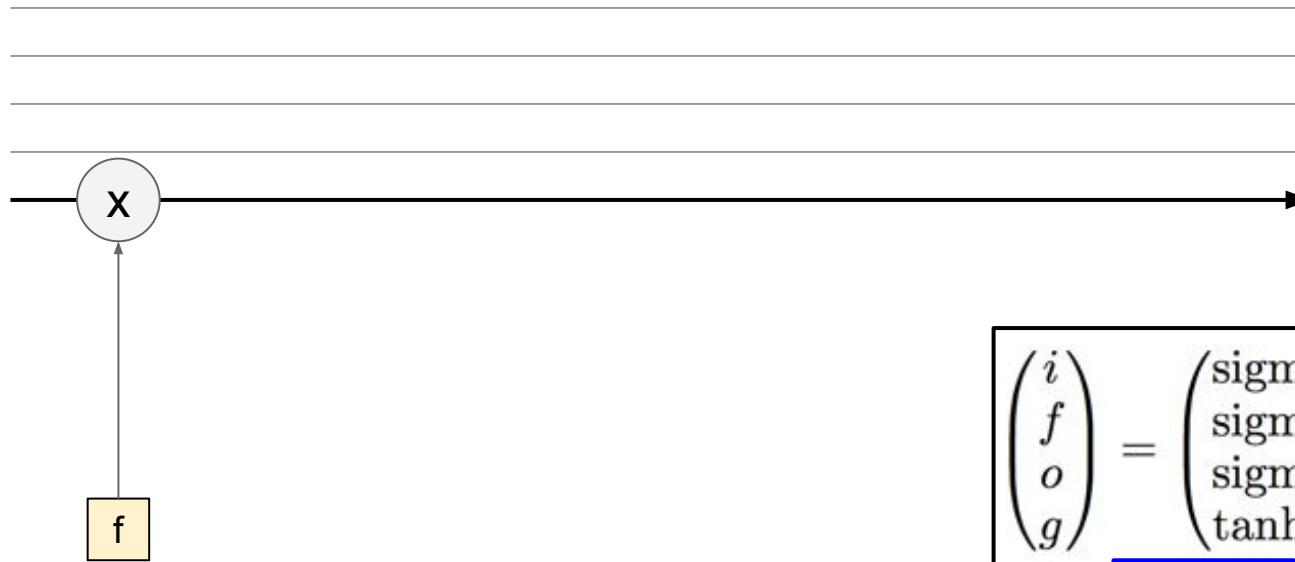


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_t^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell  
state **c**

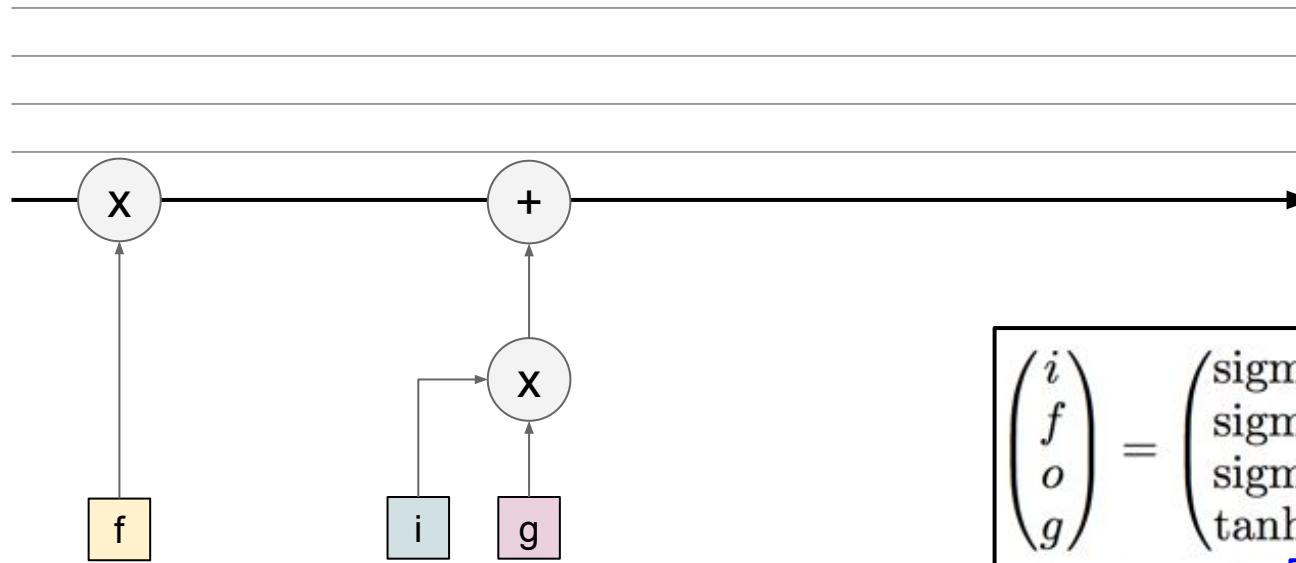


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell  
state  $c$



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

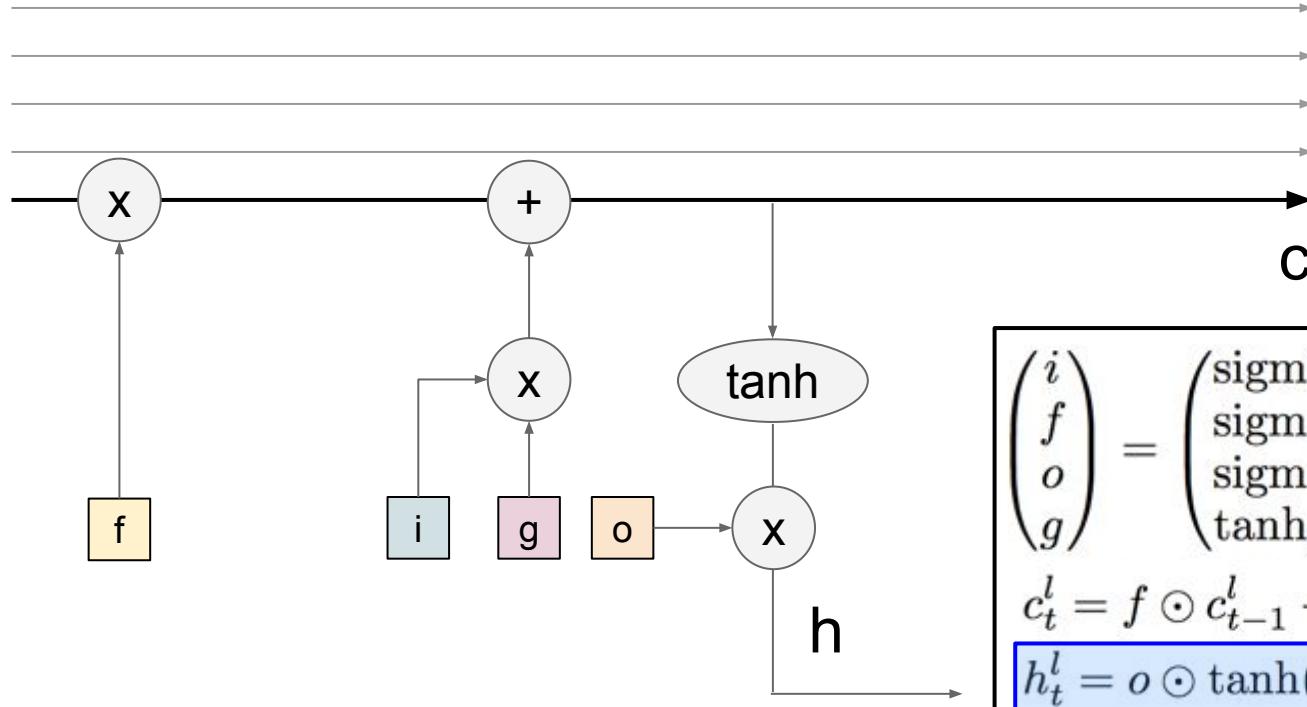
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

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

# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell  
state  $c$

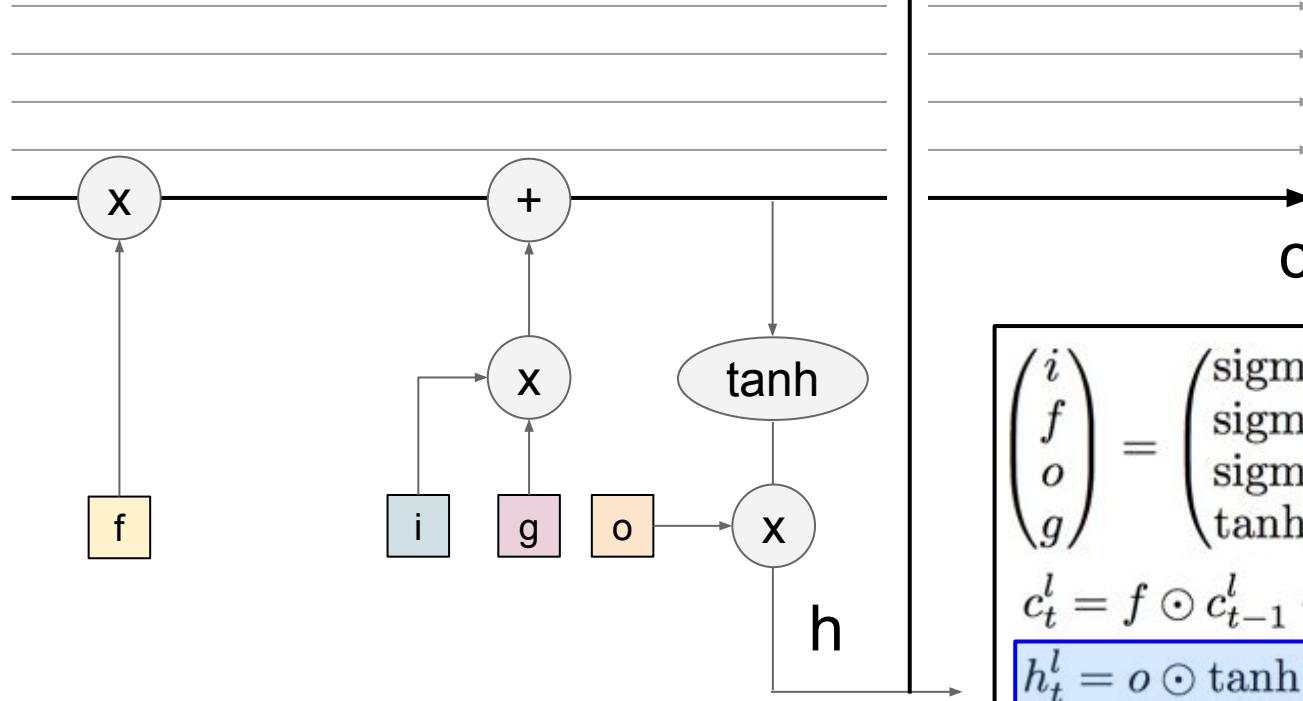


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \text{tanh}(c_t^l)$$

# Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell  
state  $c$



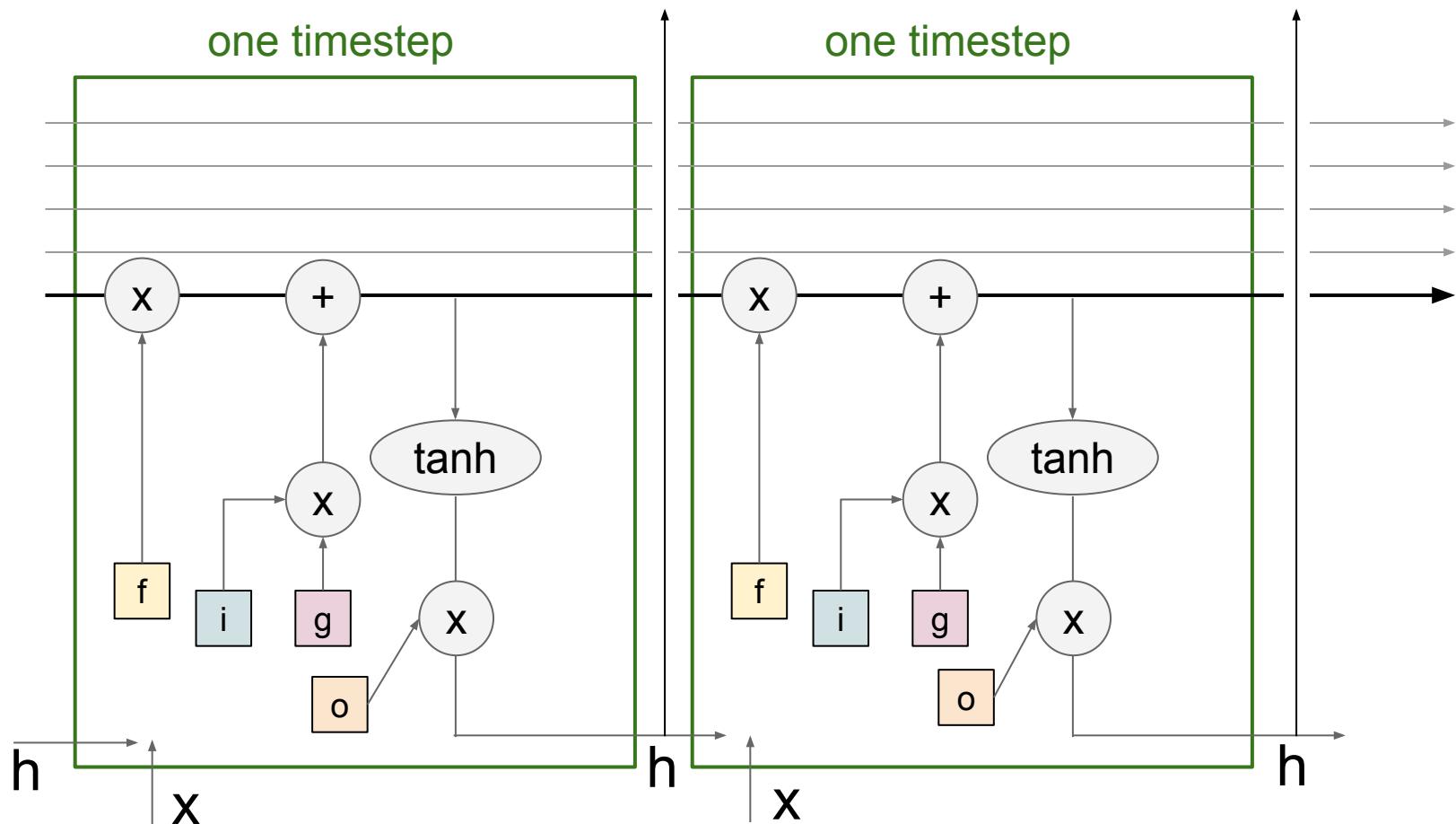
higher layer, or  
prediction

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

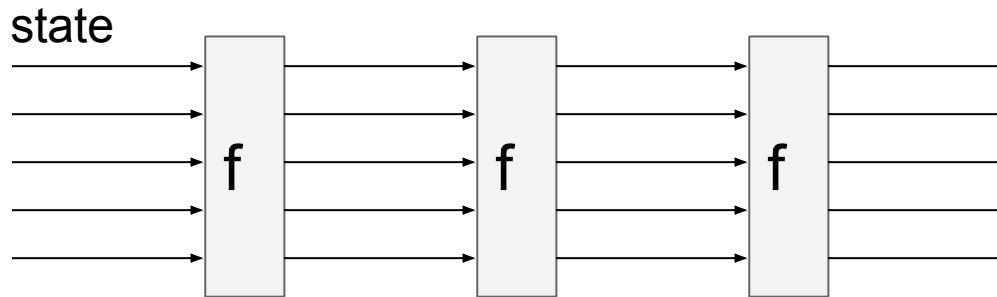
# LSTM

one timestep

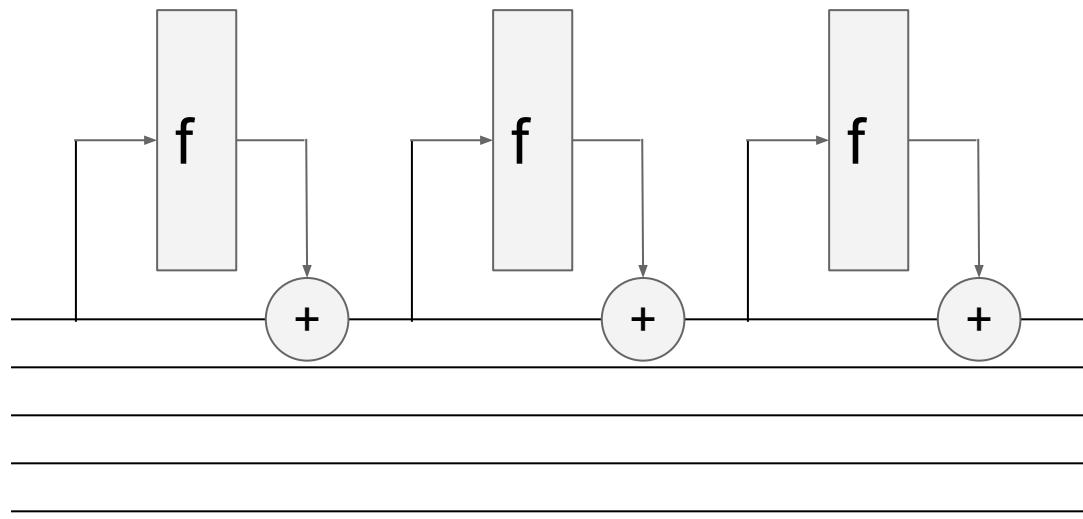
cell  
state  $c$



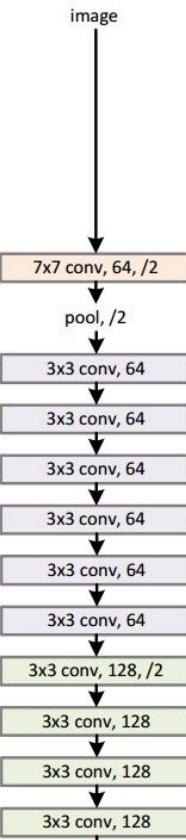
# RNN



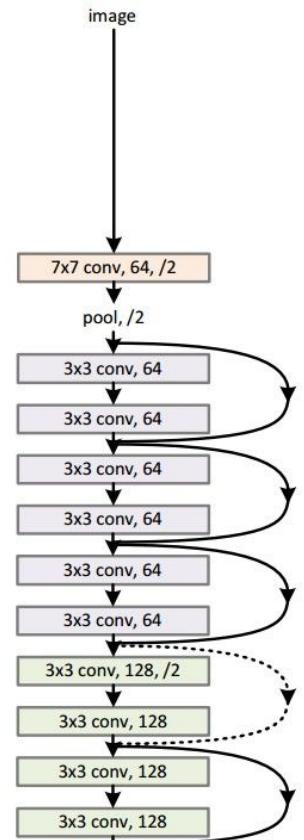
# LSTM (ignoring forget gates)



34-layer plain

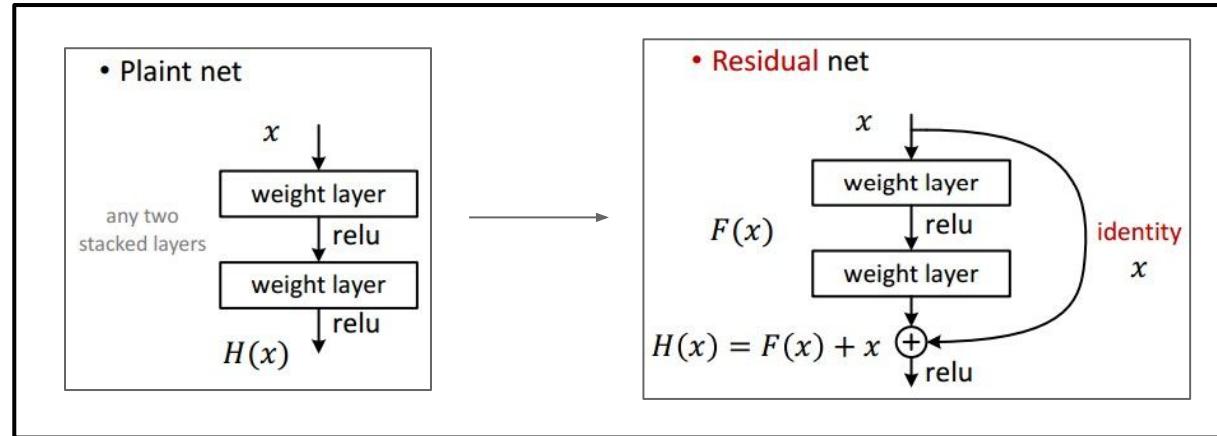


34-layer residual



## 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
```

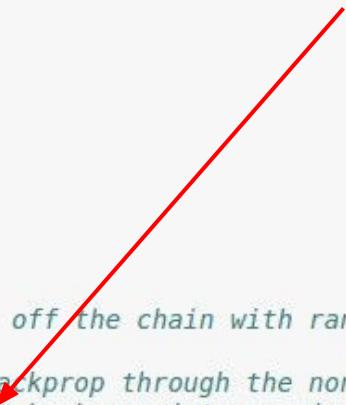
# Understanding gradient flow dynamics

```
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
```

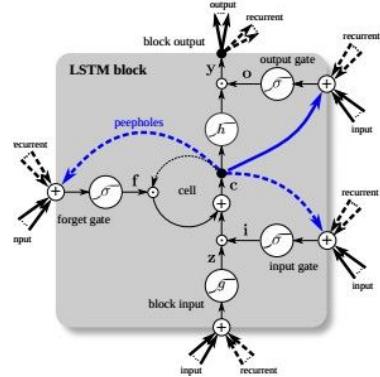
if the largest eigenvalue is  $> 1$ , gradient will explode  
if the largest eigenvalue is  $< 1$ , gradient will vanish

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.