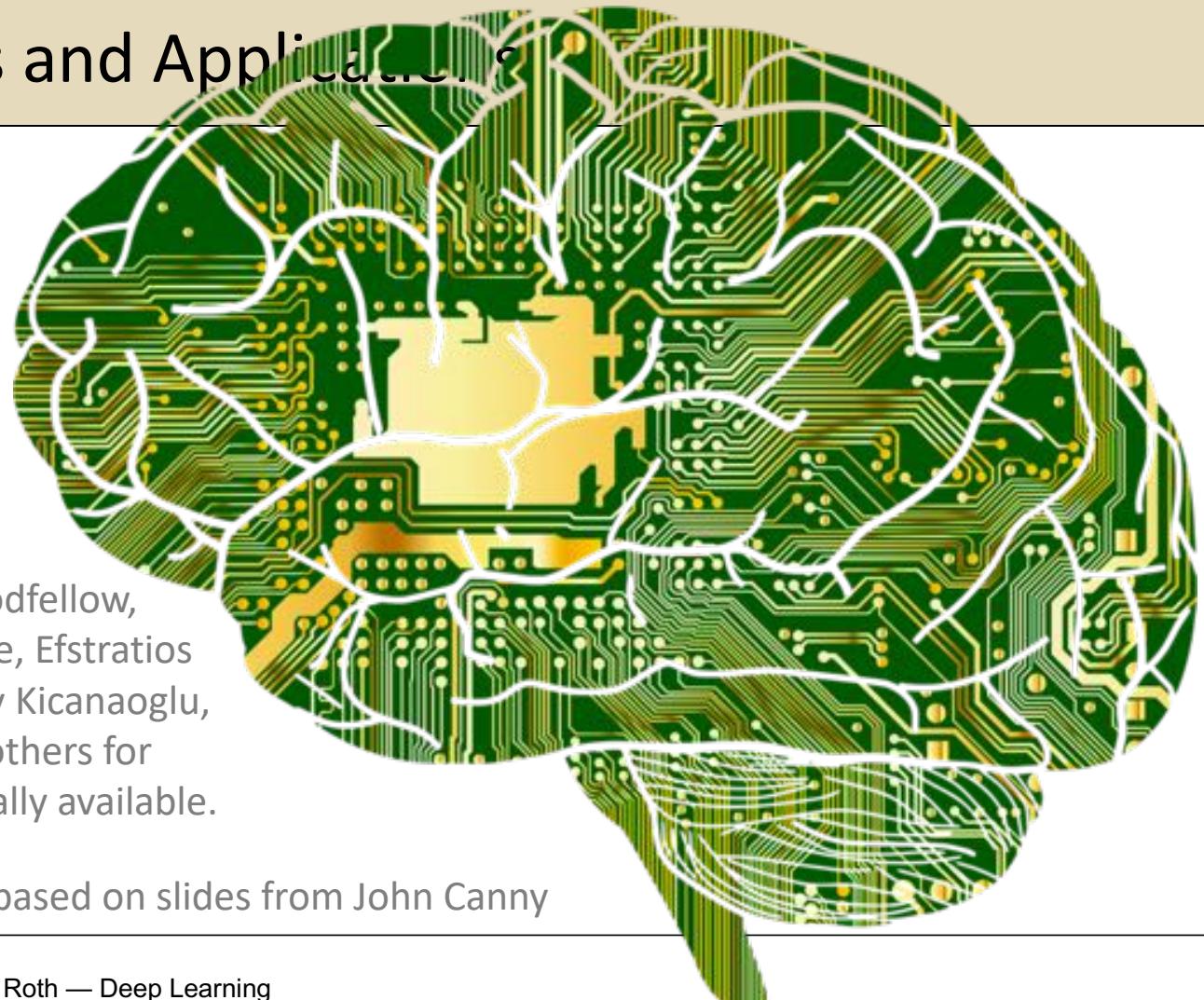


Deep Learning

Architectures and Methods: Recurrent Networks, LSTMs and Applications



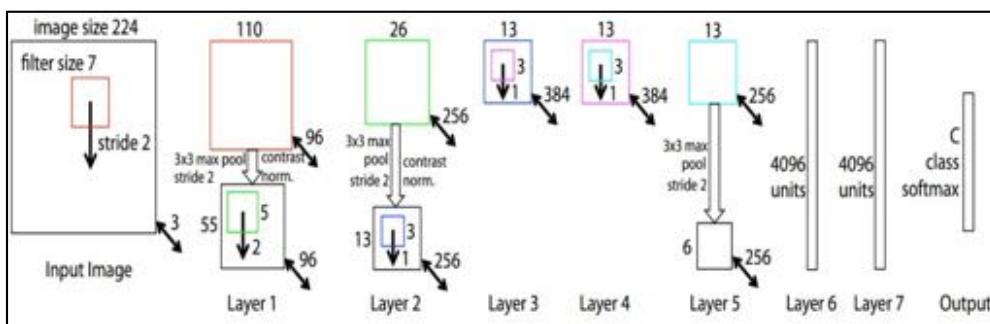
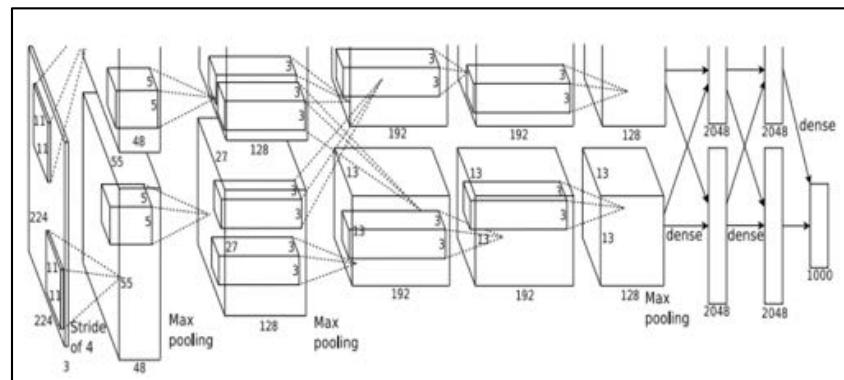
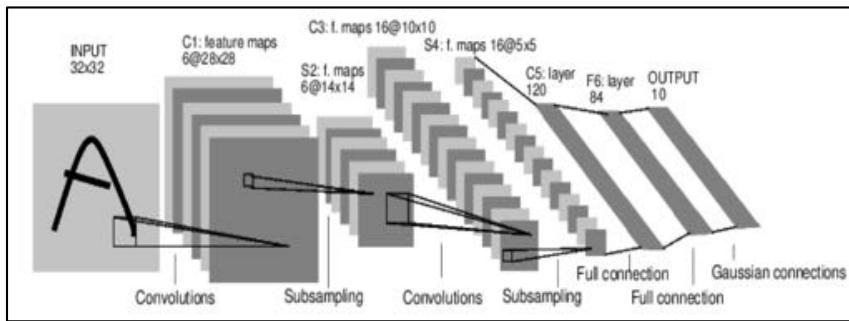
TECHNISCHE
UNIVERSITÄT
DARMSTADT



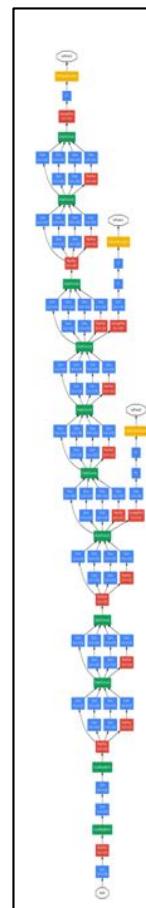
Thanks to John Canny, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Efstratios Gavves, Kirill Gavrilyuk, Berkay Kicanaoglu, and Patrick Putzky and many others for making their materials publically available.

The present slides are mainly based on slides from John Canny

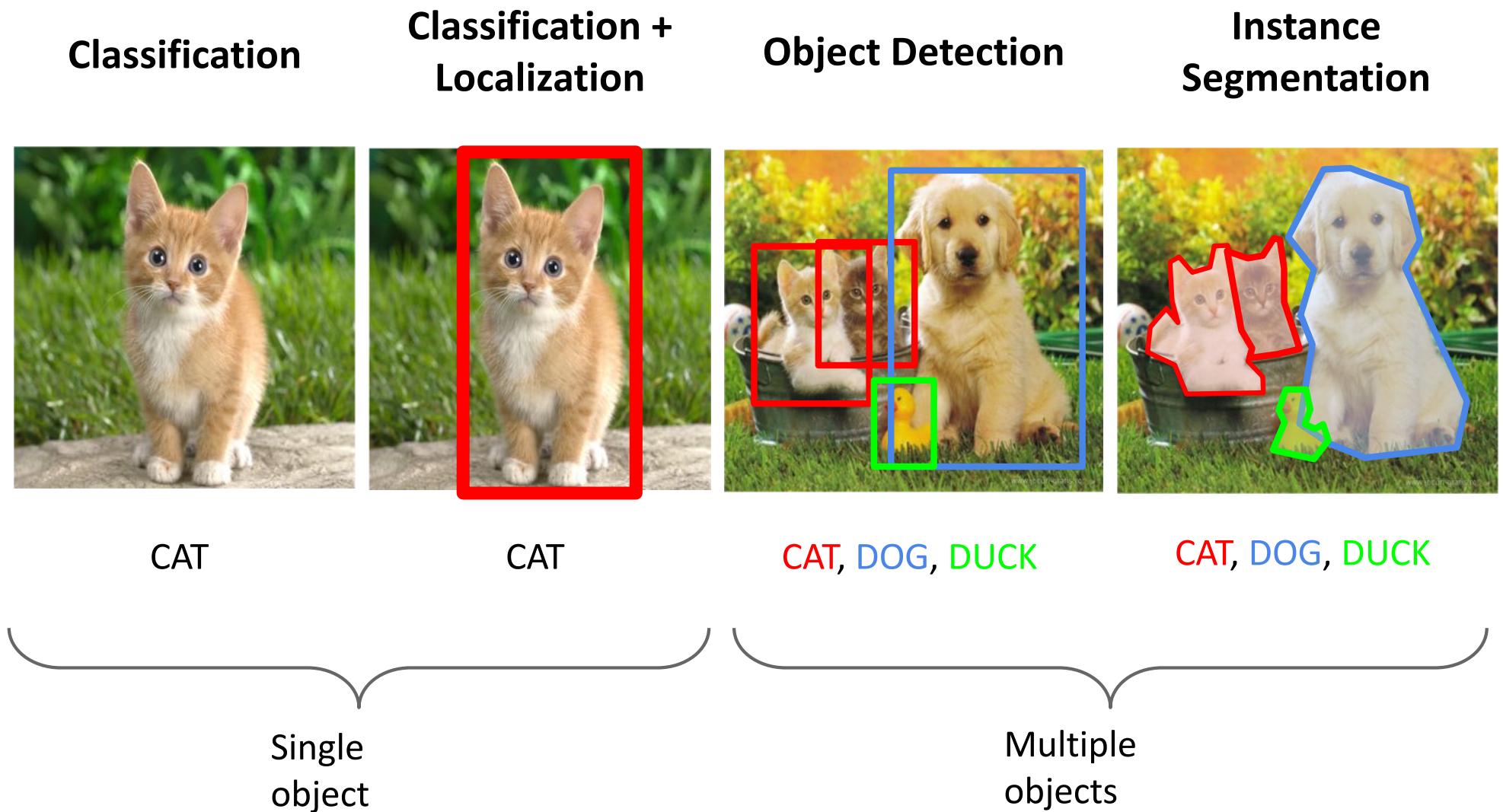
Last time: ConvNets



D	E
16 weight layers	19 weight layers
conv3-64	conv3-64
conv3-64	conv3-64
conv3-128	conv3-128
conv3-128	conv3-128
conv3-256	conv3-256
conv3-256	conv3-256
conv3-256	conv3-256
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
maxpool	
FC-4096	
FC-4096	
FC-1000	
soft-max	



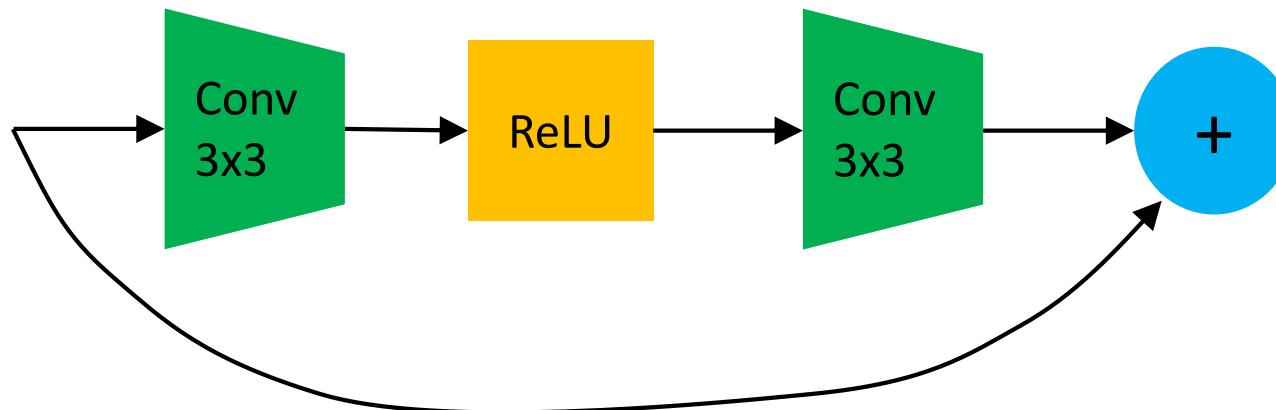
Last time: Localization and Detection



Neural Network structure

Standard Neural Networks are DAGs (Directed Acyclic Graphs). That means they have a topological ordering.

- The topological ordering is used for activation propagation, and for gradient back-propagation.

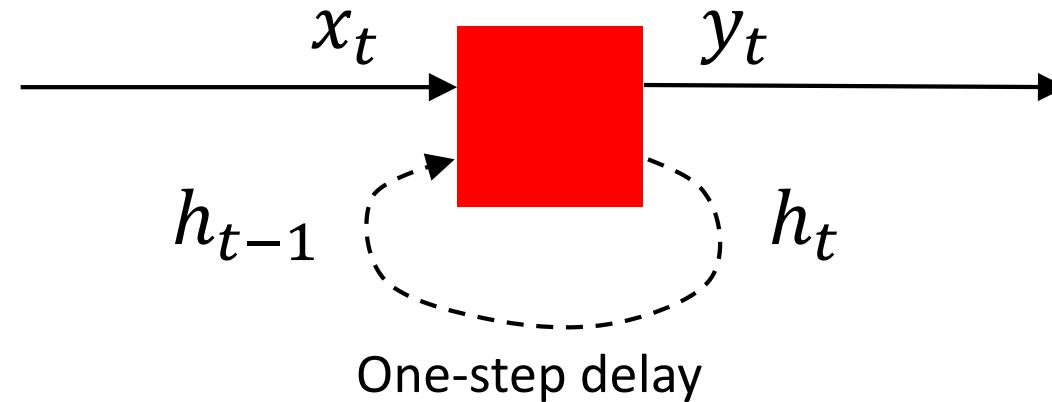


- They process one input instance at a time.



Recurrent Neural Networks (RNNs)

Recurrent networks introduce cycles and a notion of time.

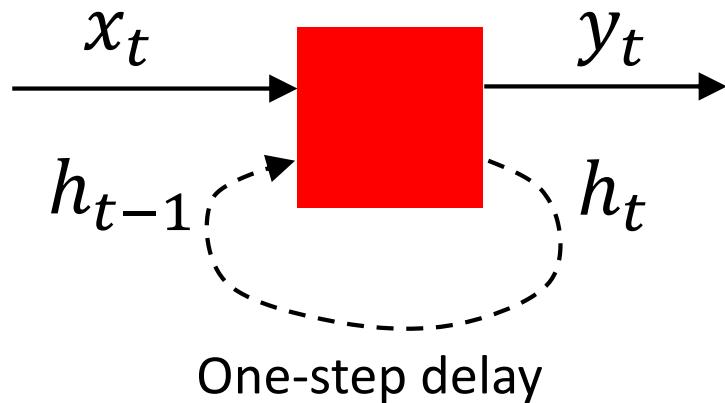


- They are designed to process sequences of data x_1, \dots, x_n and can produce sequences of outputs y_1, \dots, y_m .



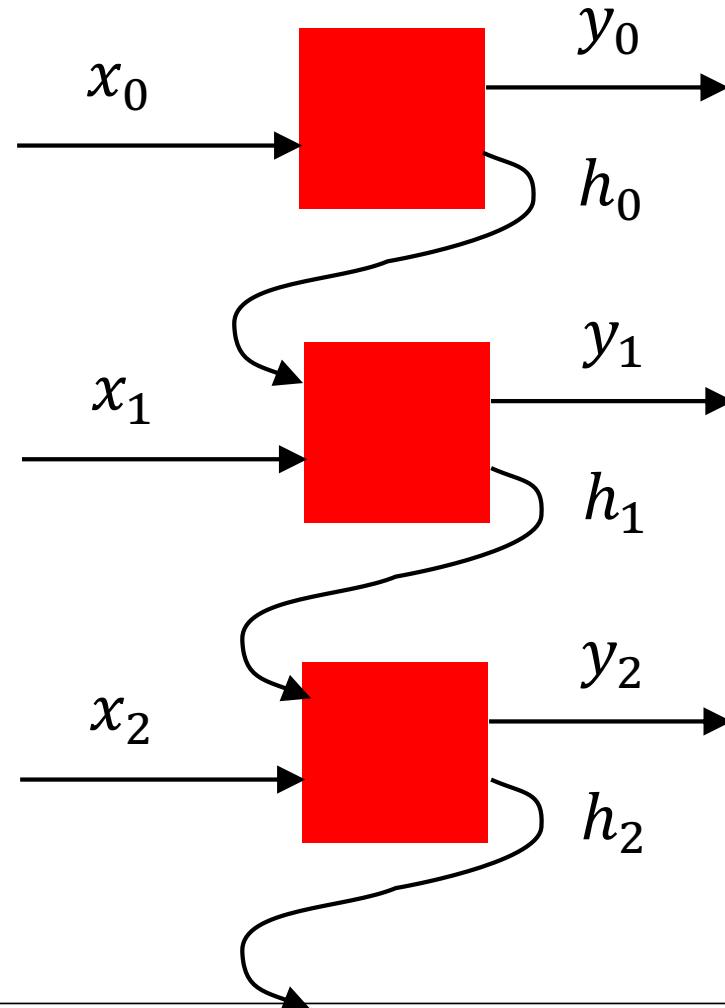
Unrolling RNNs

RNNs can be unrolled across multiple time steps.



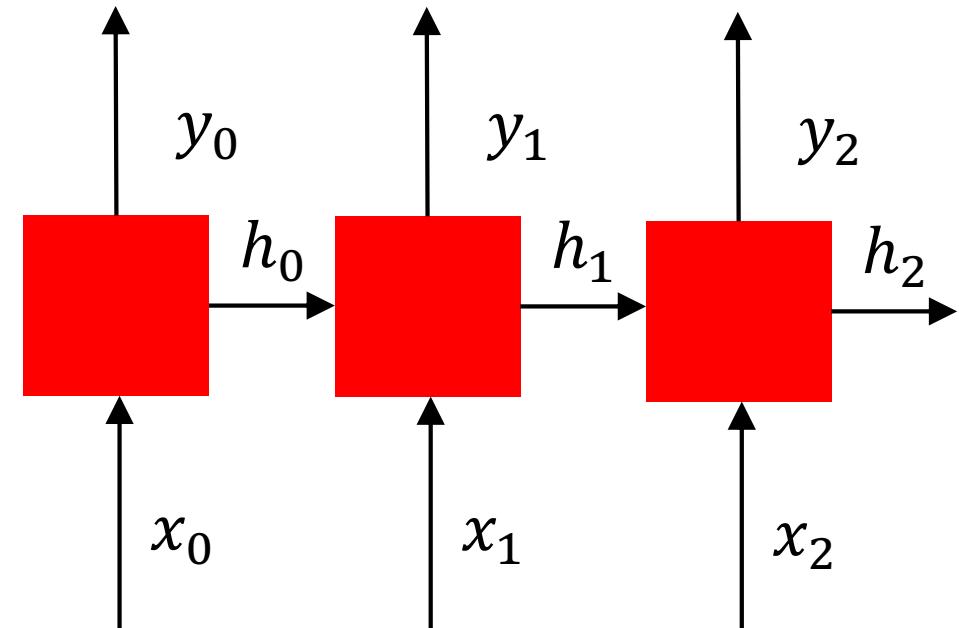
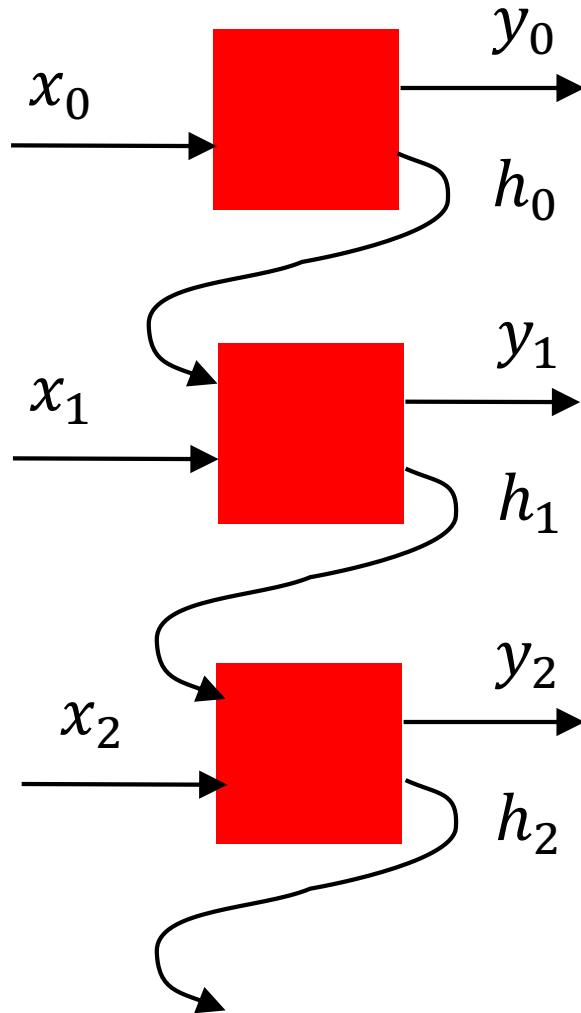
This produces a DAG which supports backpropagation.

But its size depends on the input sequence length.



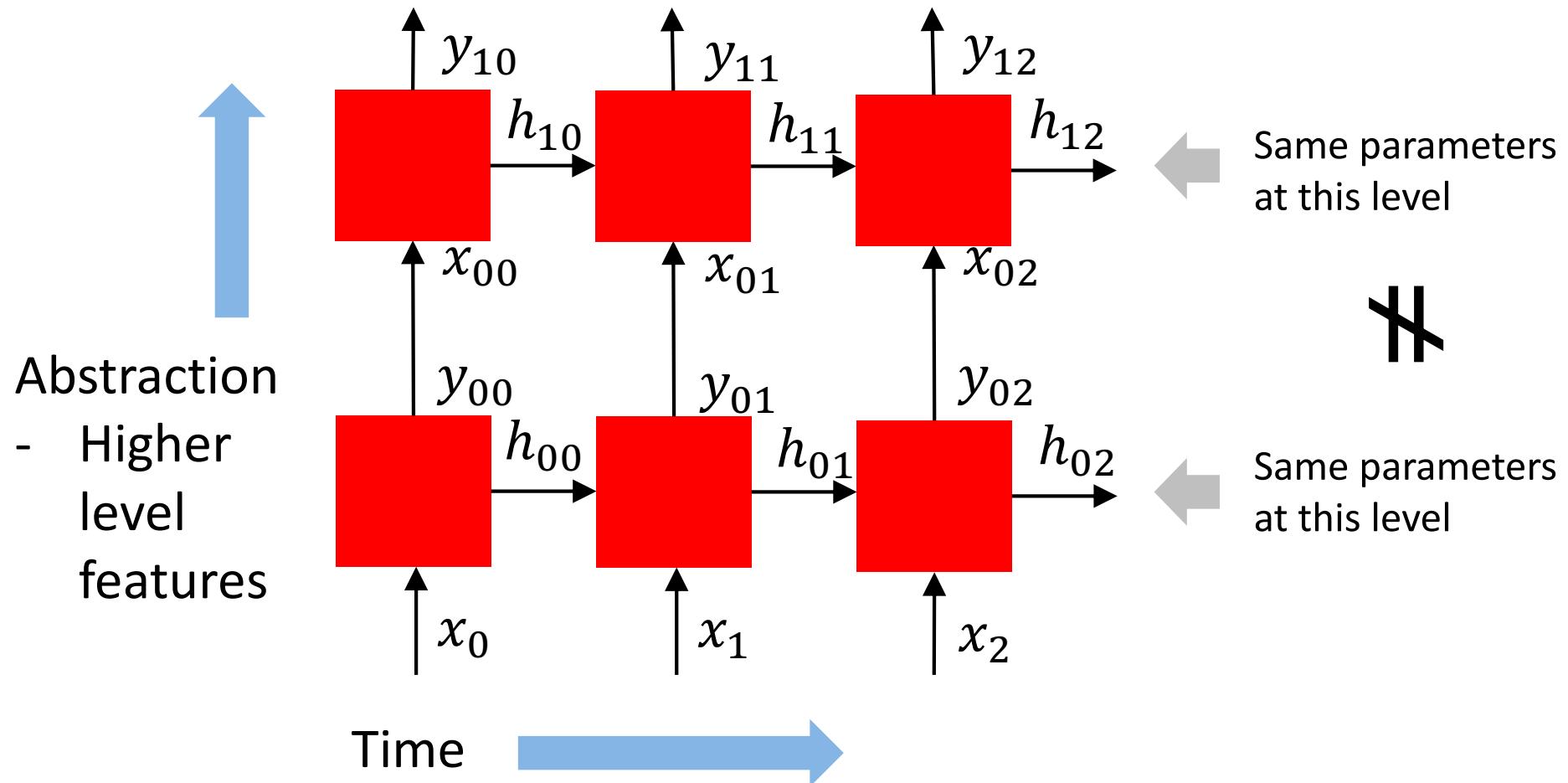
Unrolling RNNs

Usually drawn as:



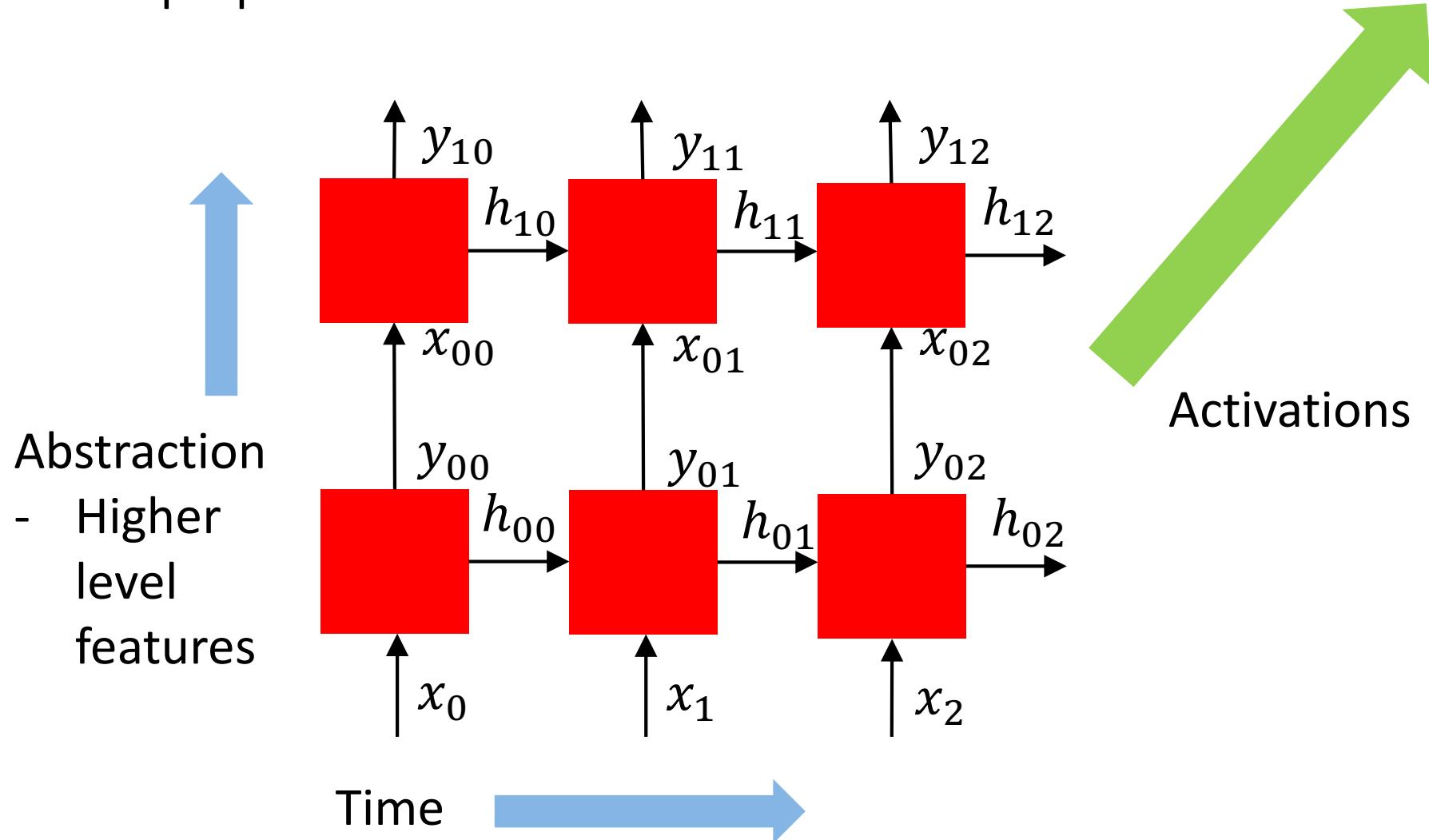
RNN structure

Often layers are stacked vertically (deep RNNs):



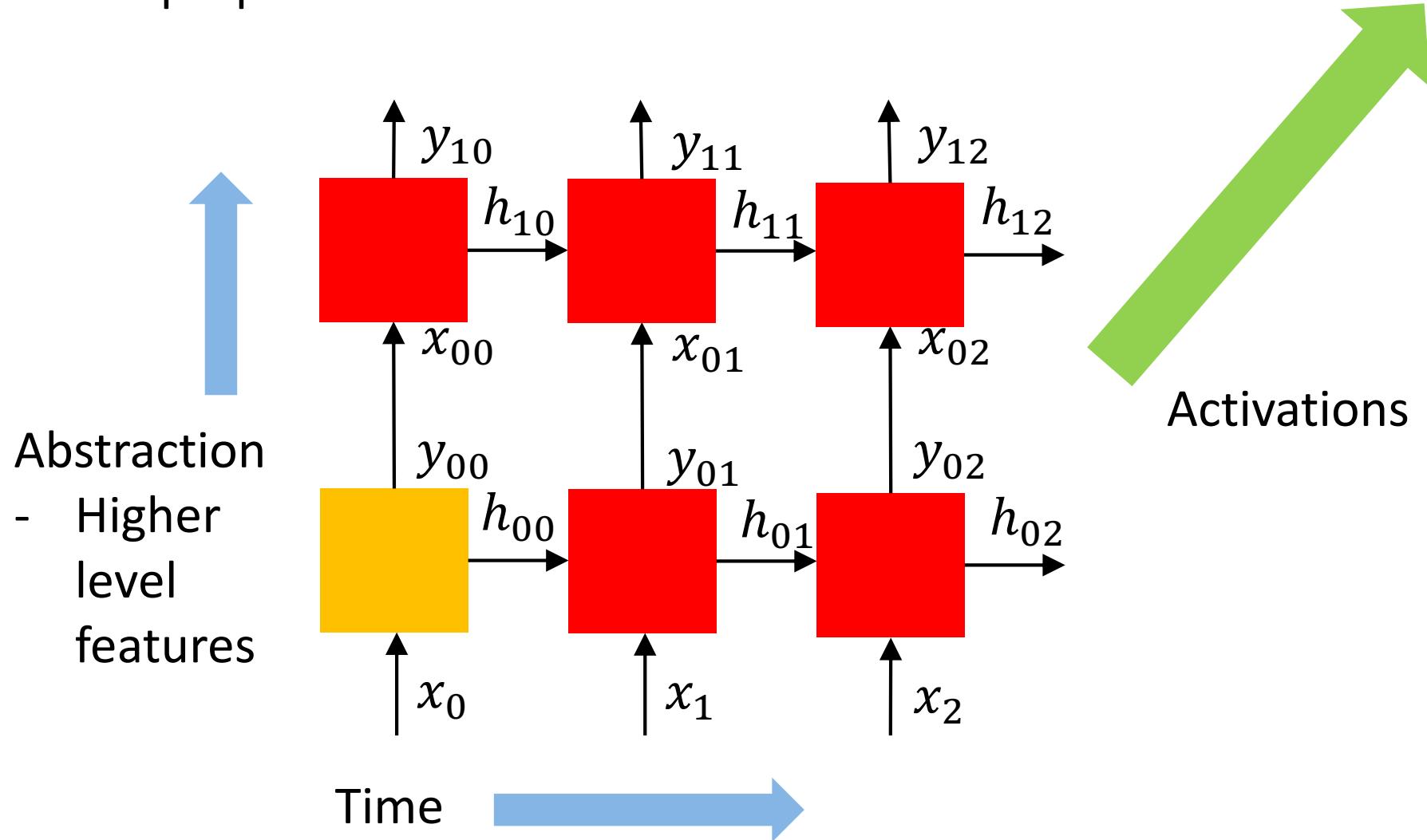
RNN structure

Backprop still works:



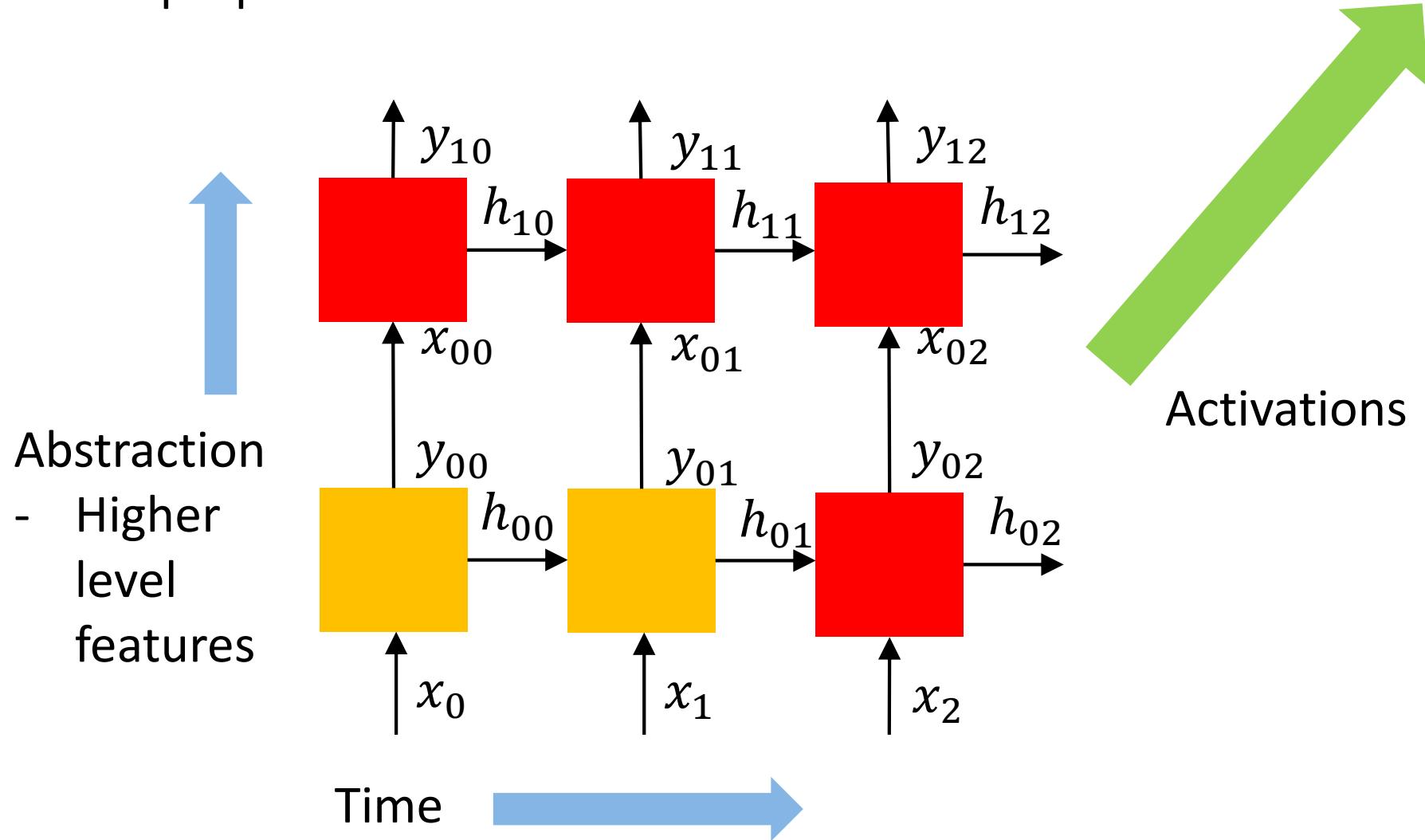
RNN structure

Backprop still works:



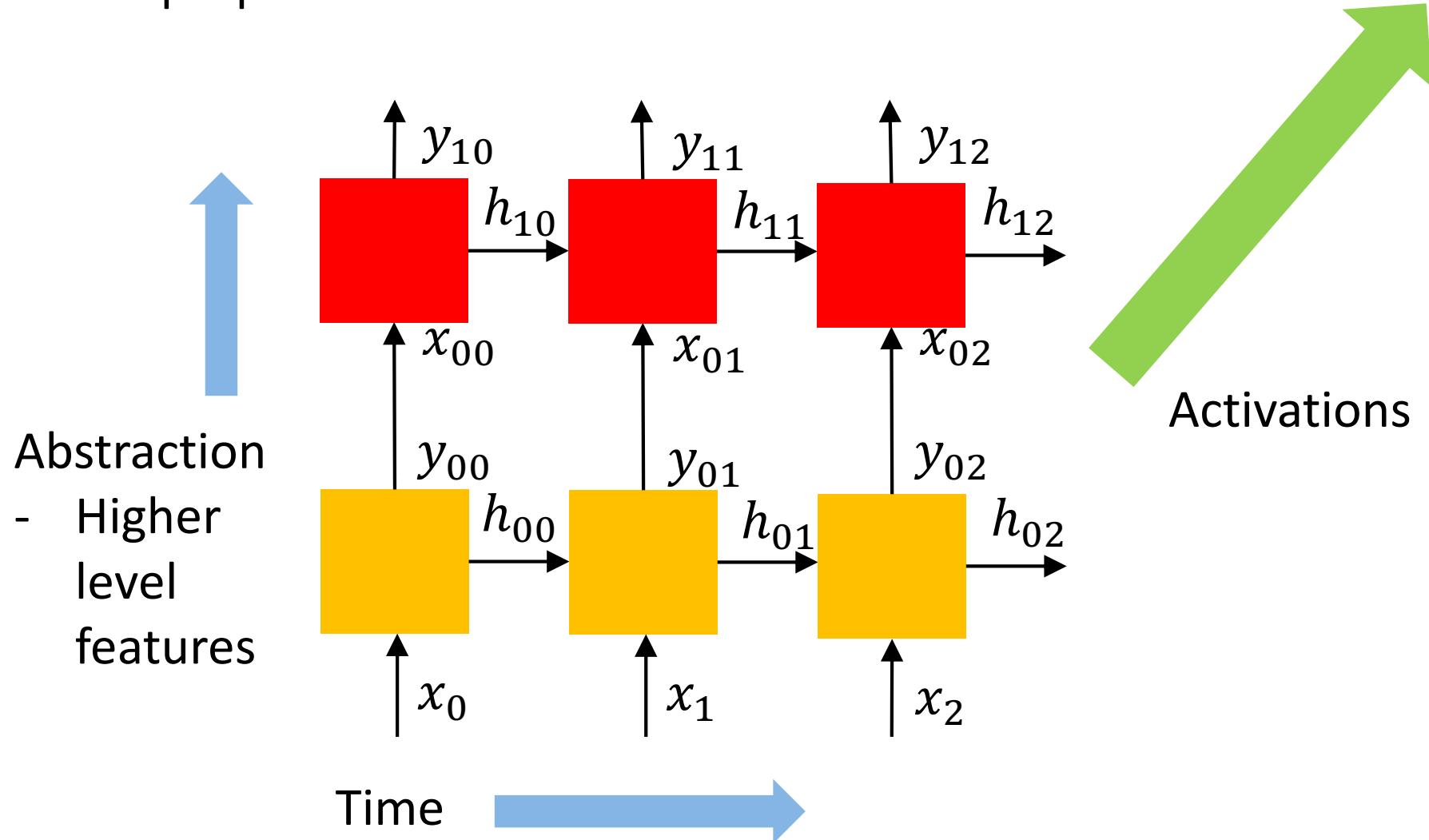
RNN structure

Backprop still works:



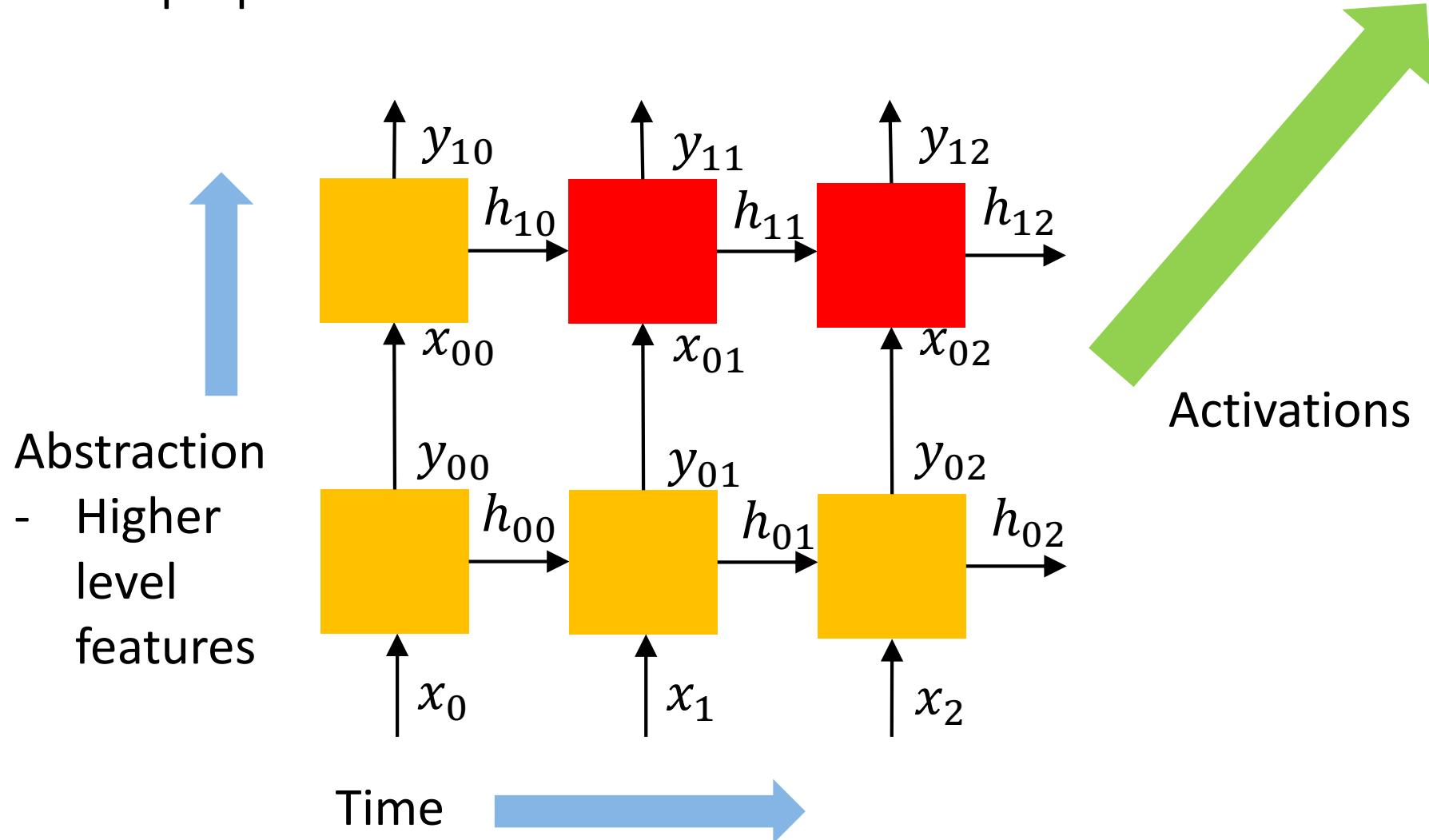
RNN structure

Backprop still works:



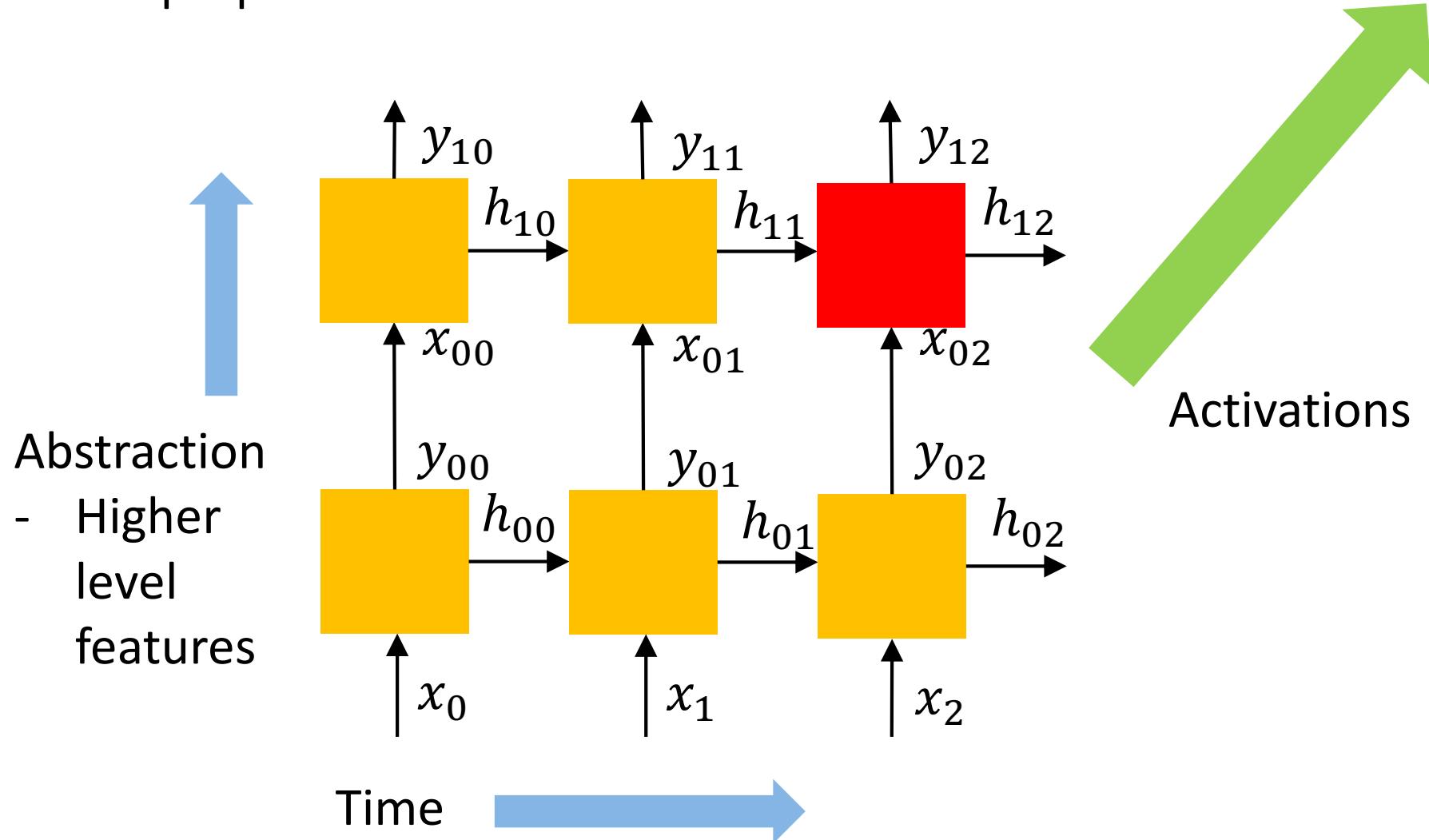
RNN structure

Backprop still works:



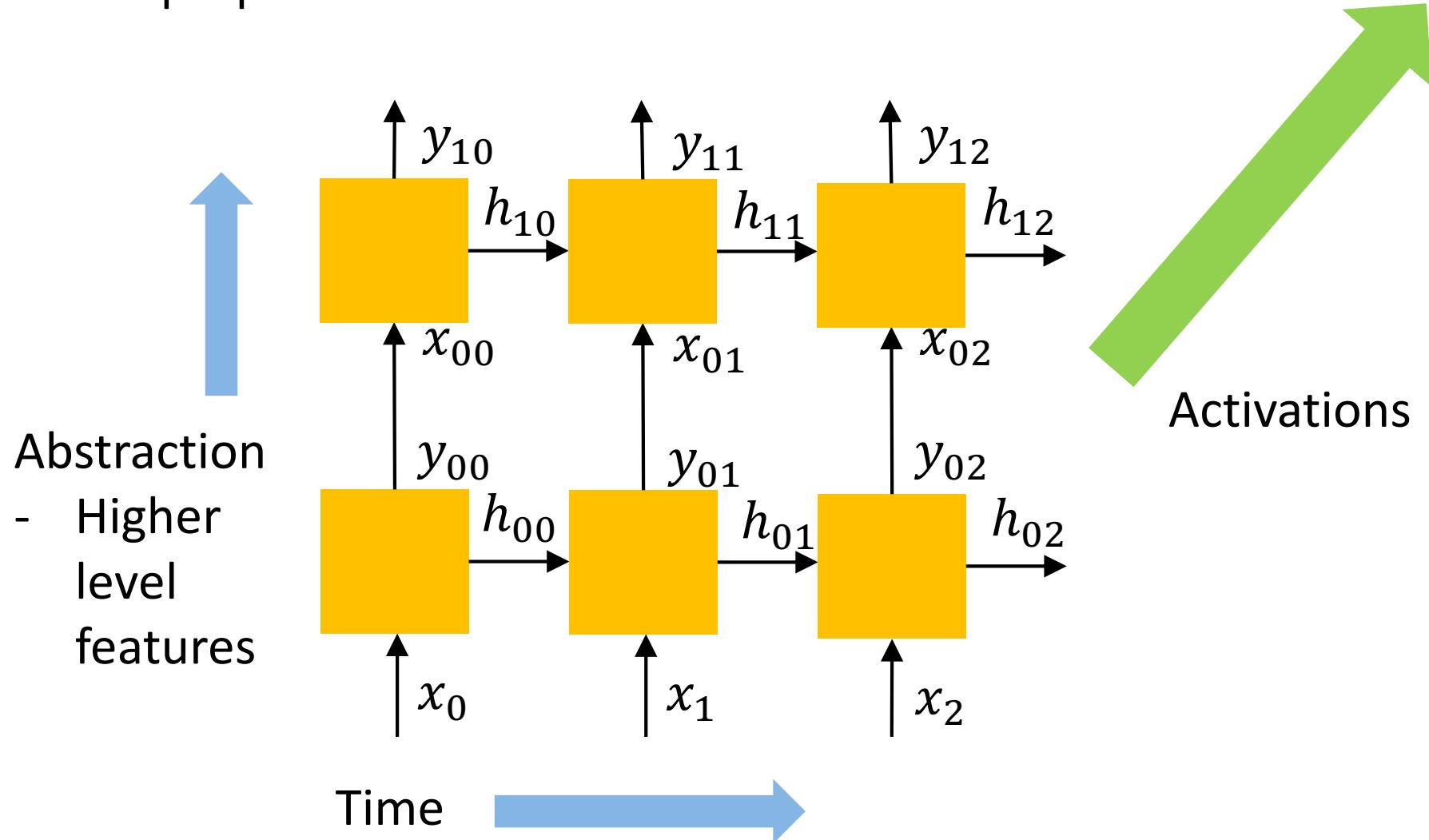
RNN structure

Backprop still works:



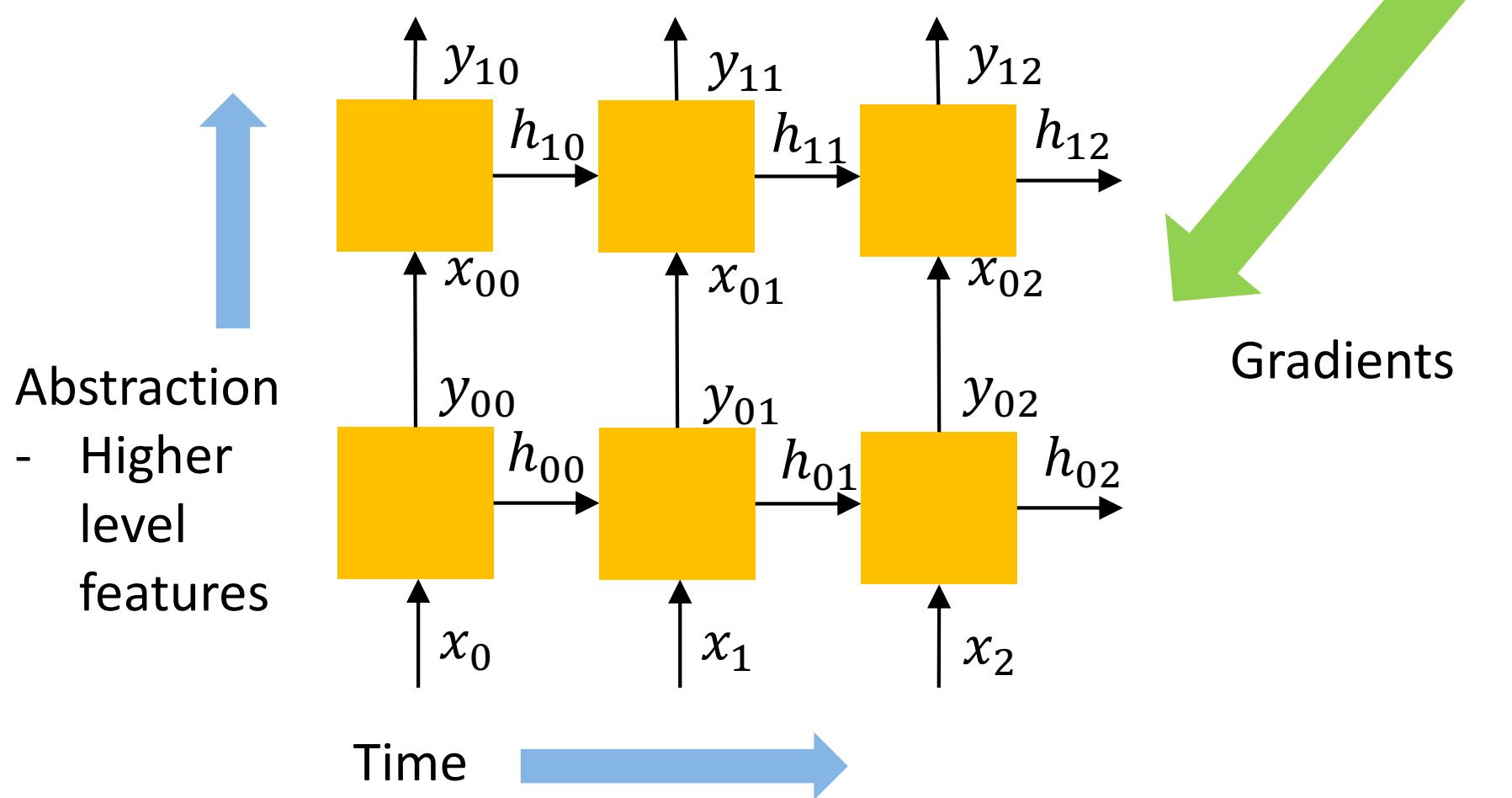
RNN structure

Backprop still works:



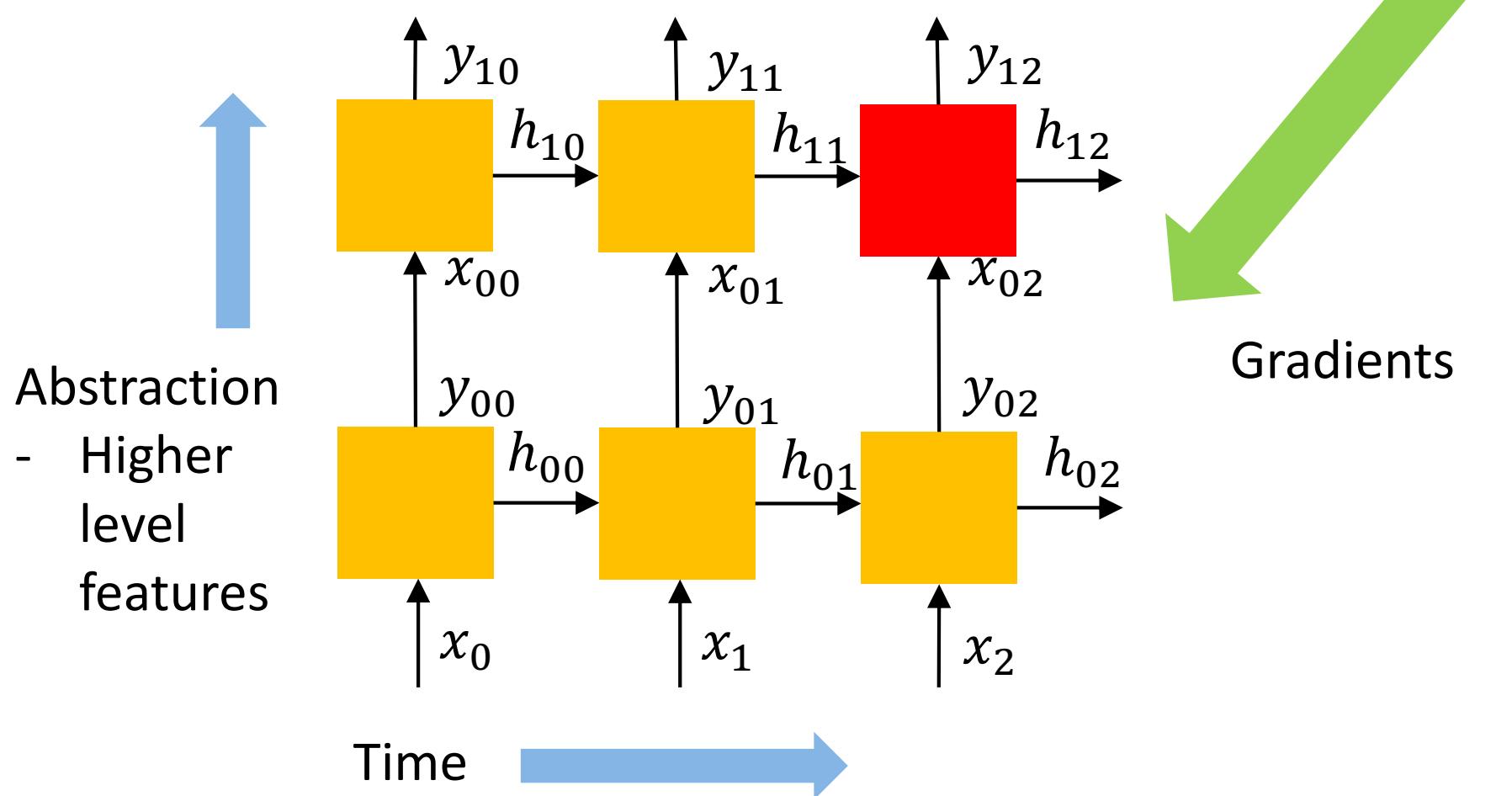
RNN structure

Backprop still works:



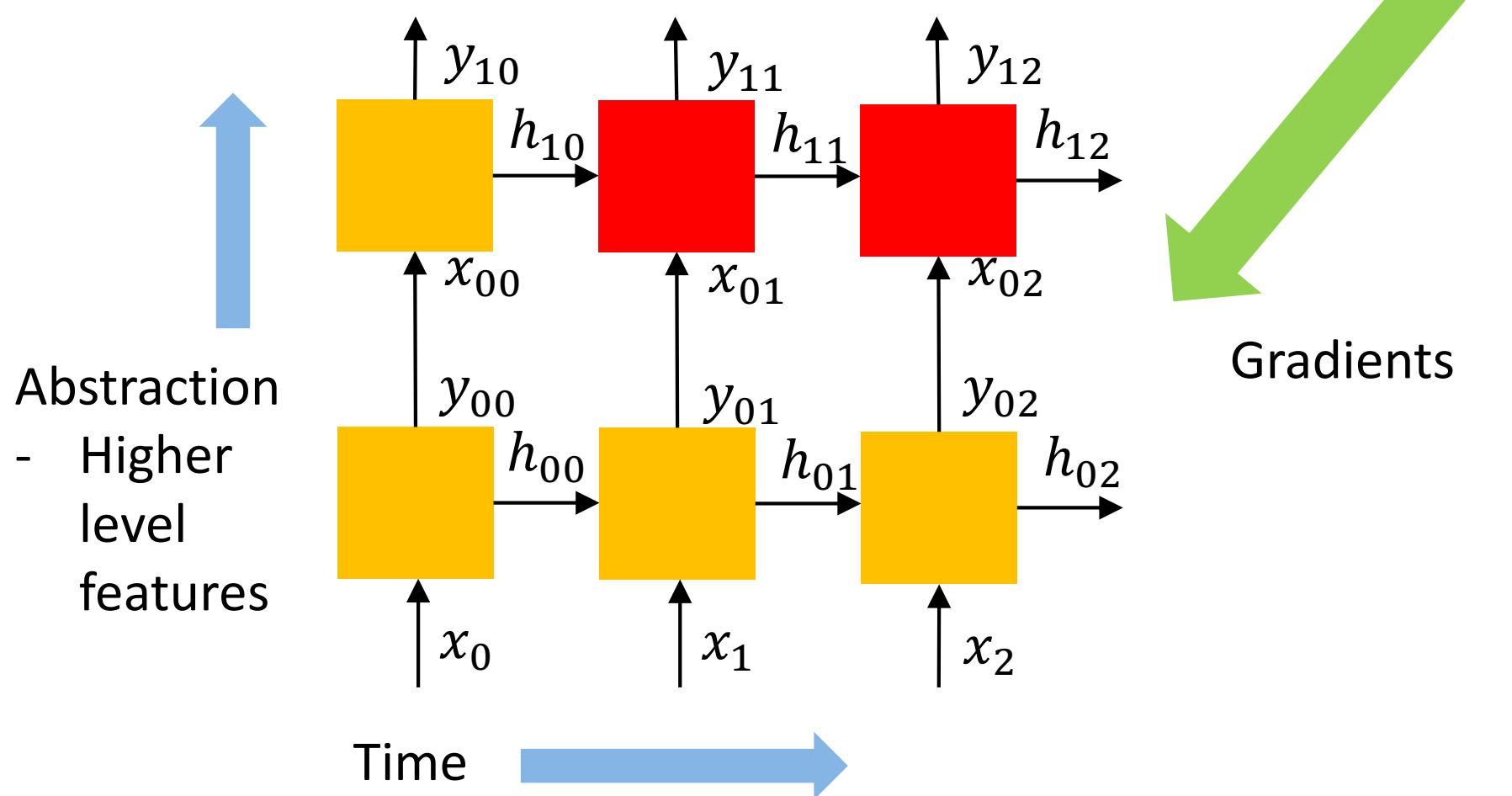
RNN structure

Backprop still works:



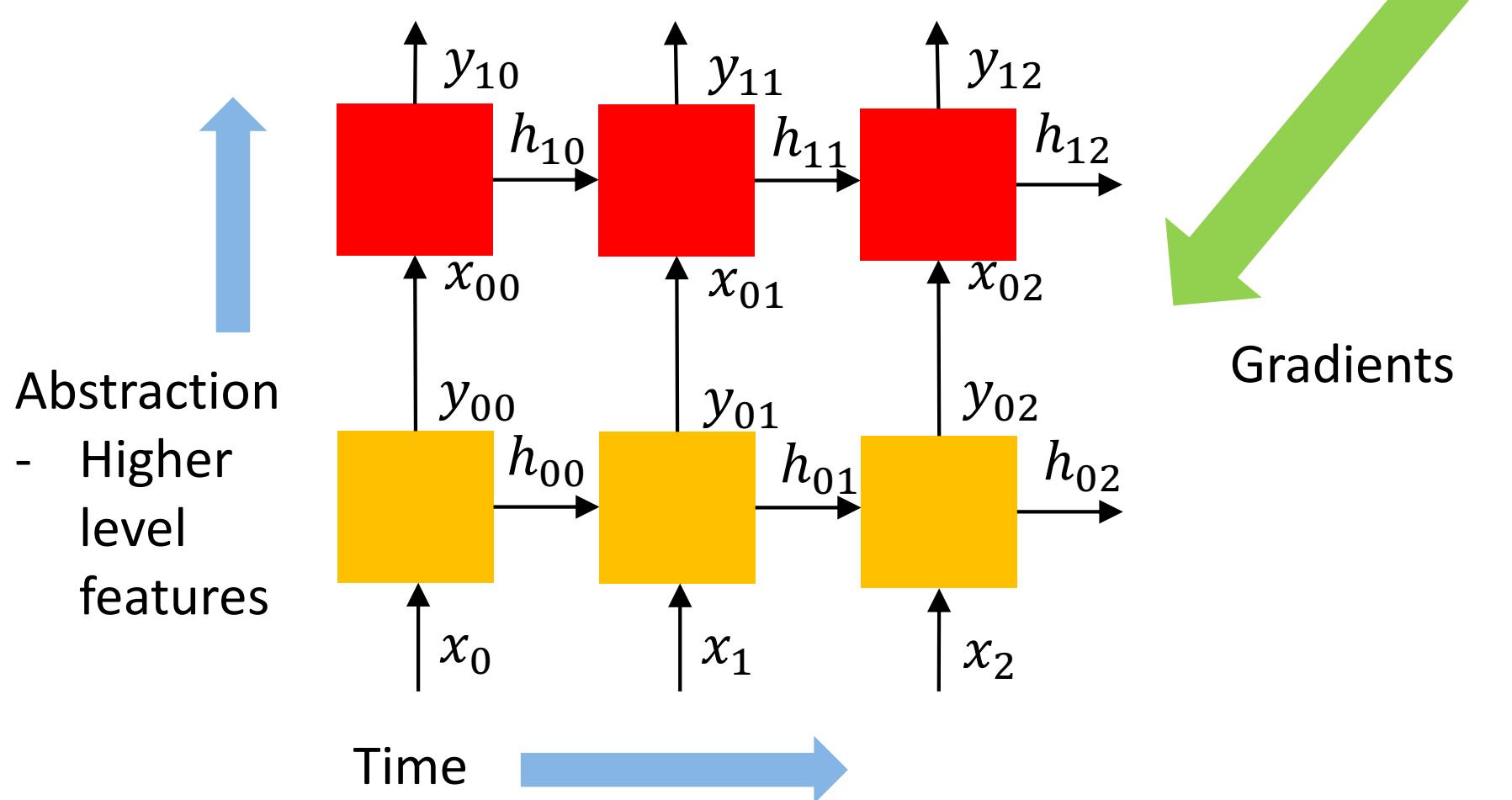
RNN structure

Backprop still works:



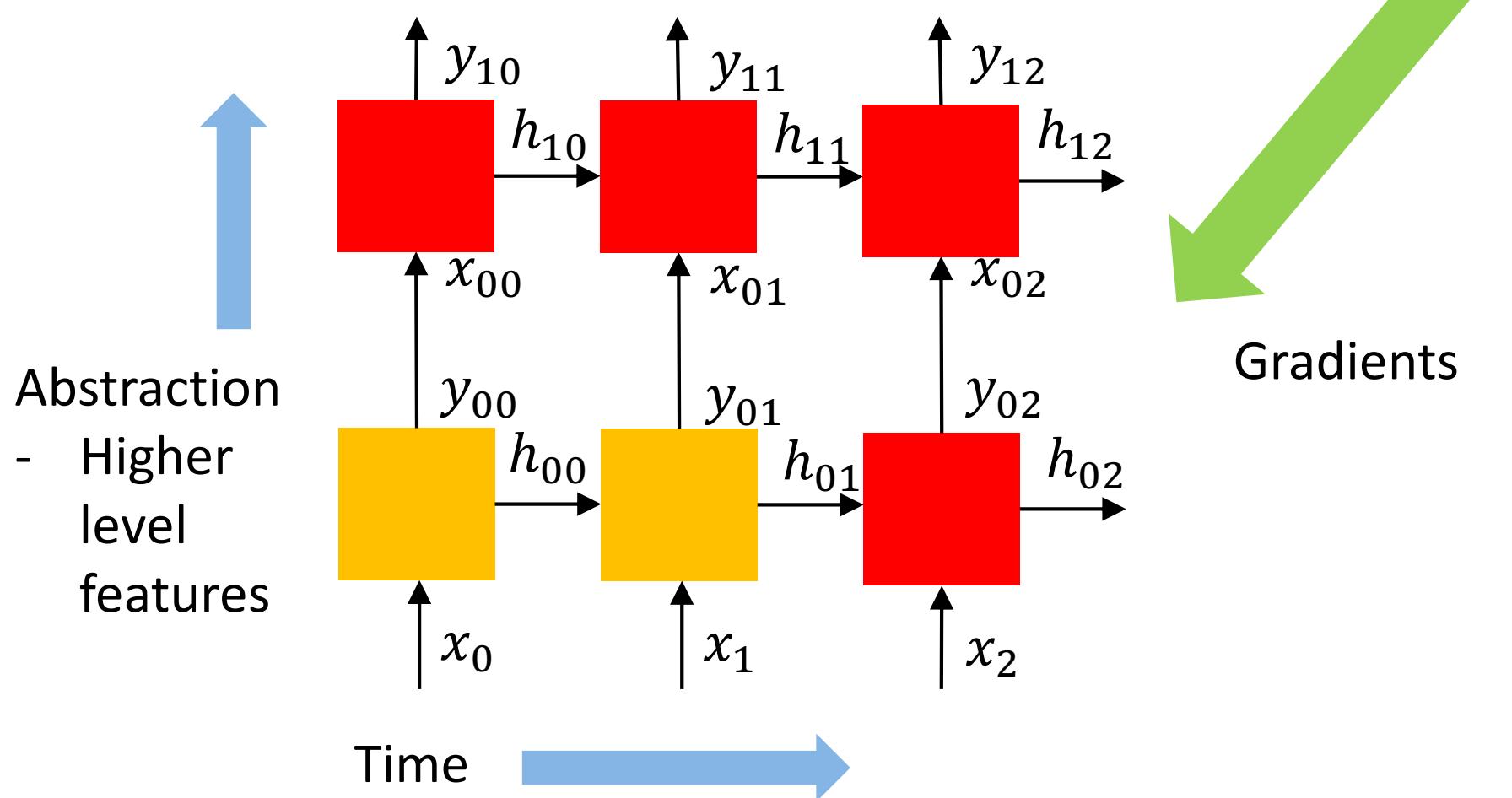
RNN structure

Backprop still works:



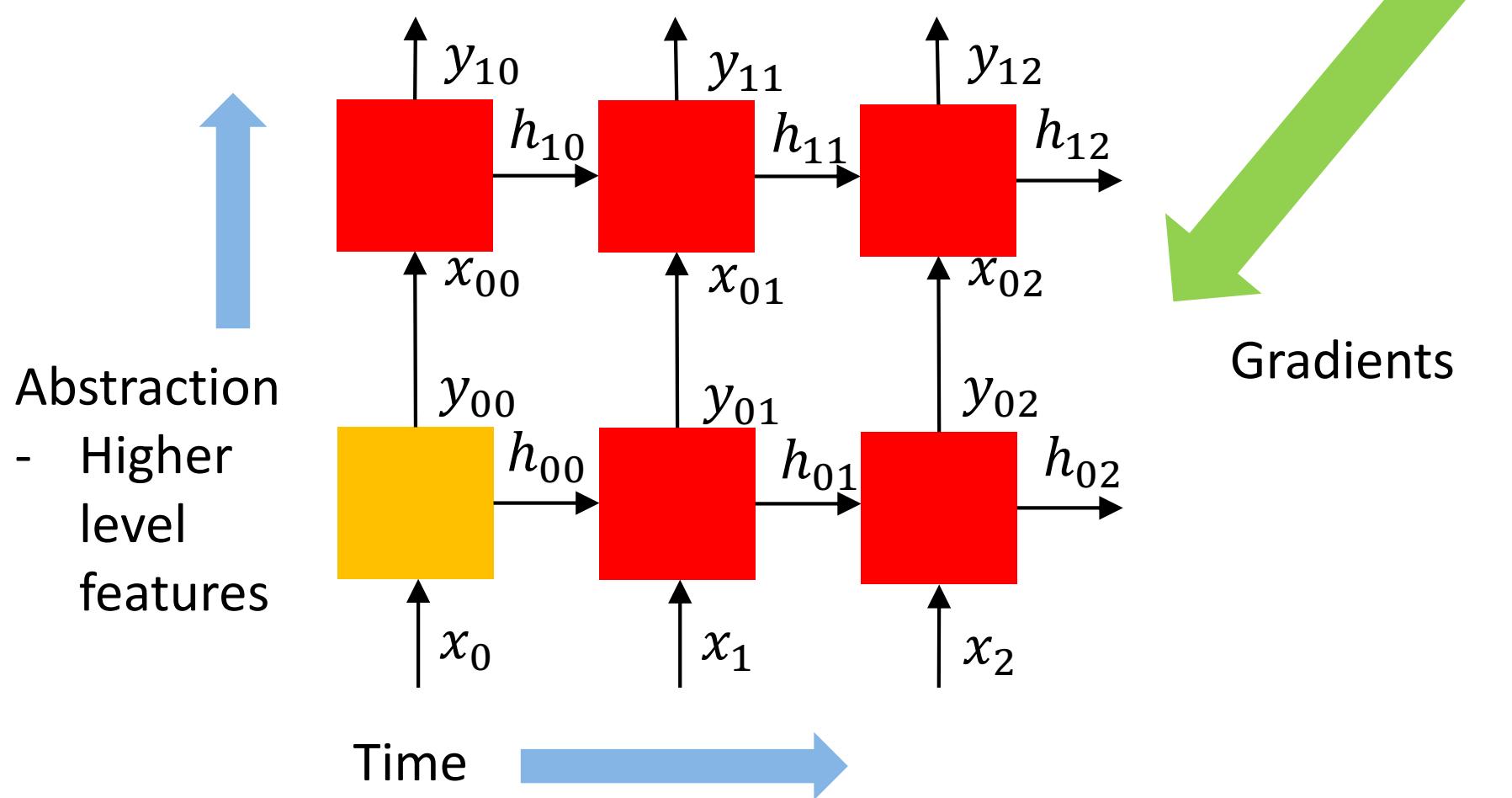
RNN structure

Backprop still works:



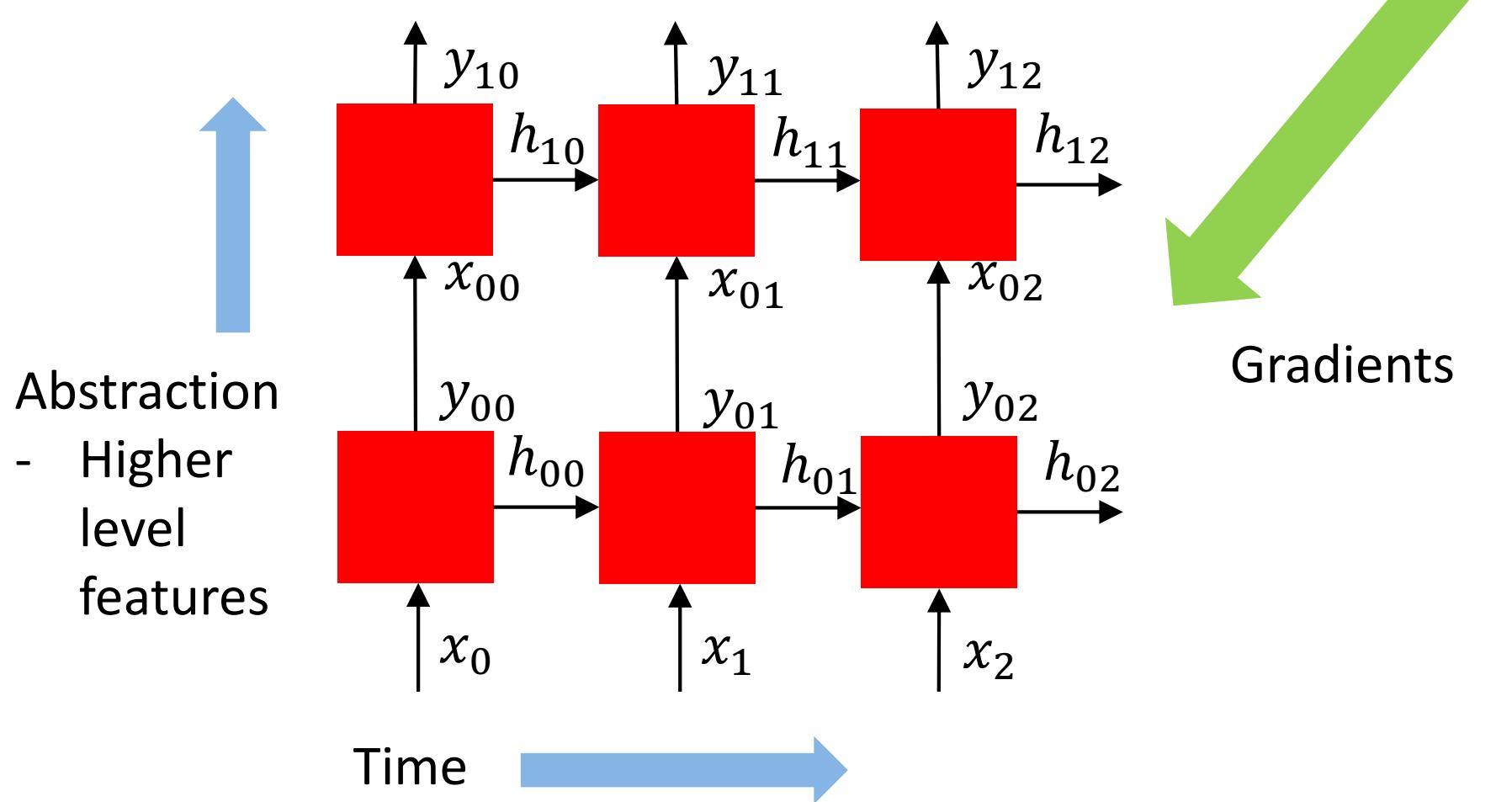
RNN structure

Backprop still works:



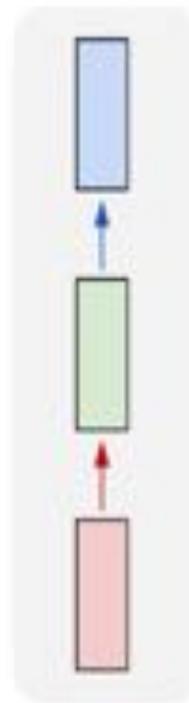
RNN structure

Backprop still works:

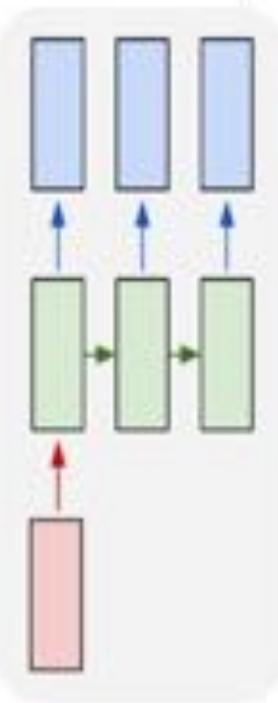


Recurrent Networks offer a lot of flexibility:

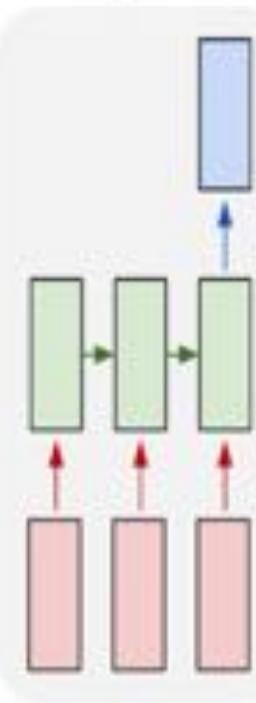
one to one



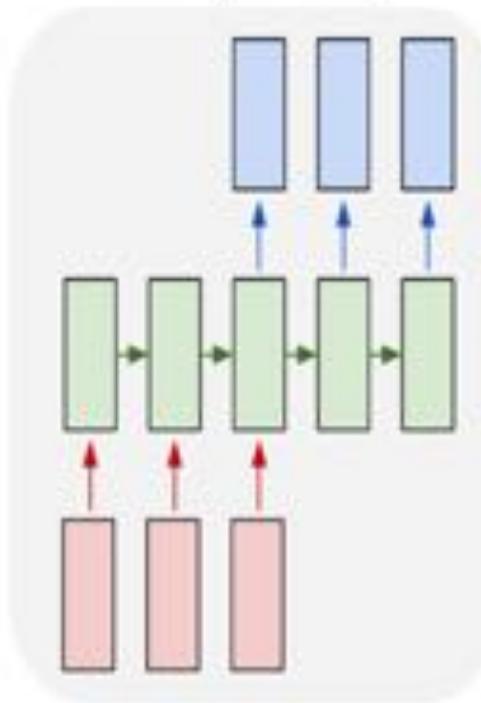
one to many



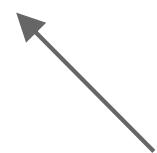
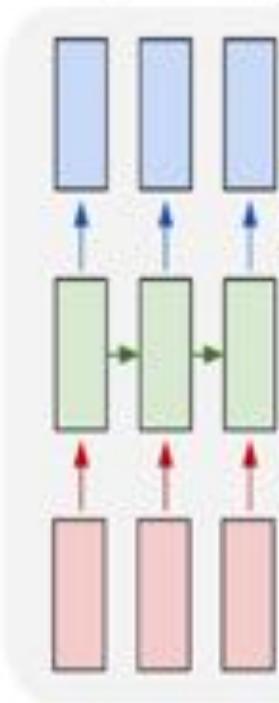
many to one



many to many



many to many

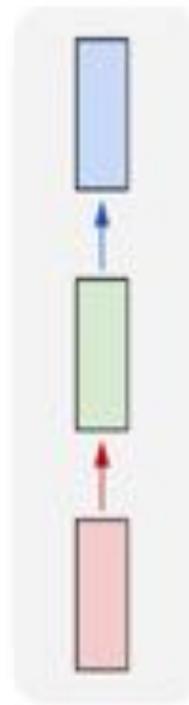


Vanilla Neural Networks

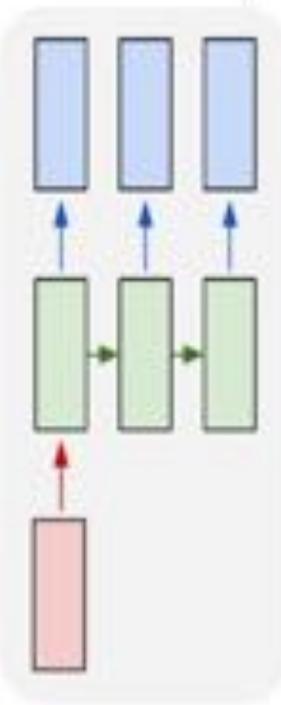


Recurrent Networks offer a lot of flexibility:

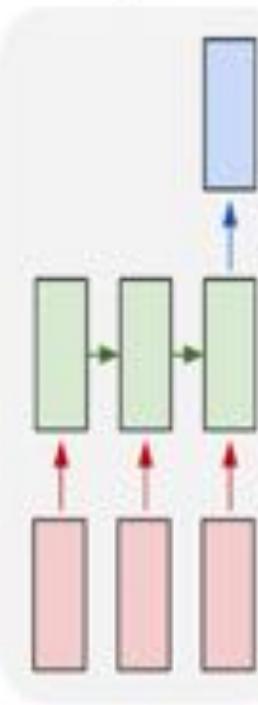
one to one



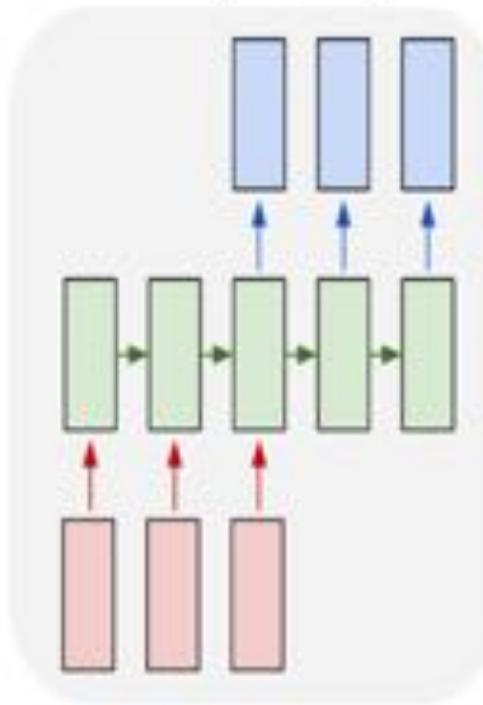
one to many



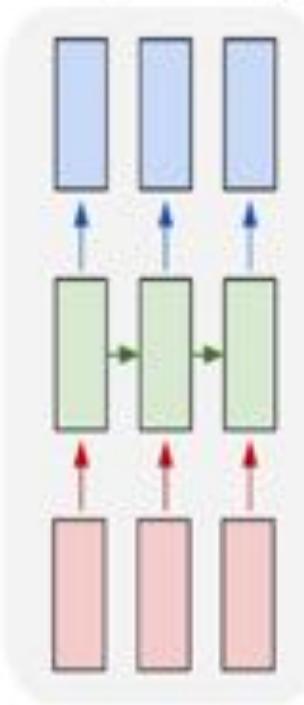
many to one



many to many



many to many

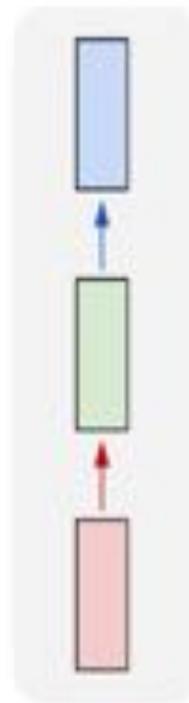


e.g. Image Captioning
image -> sequence of words

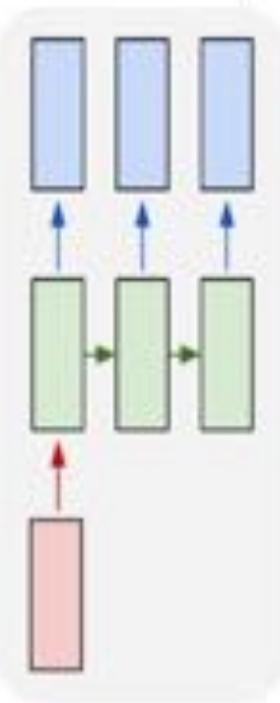


Recurrent Networks offer a lot of flexibility:

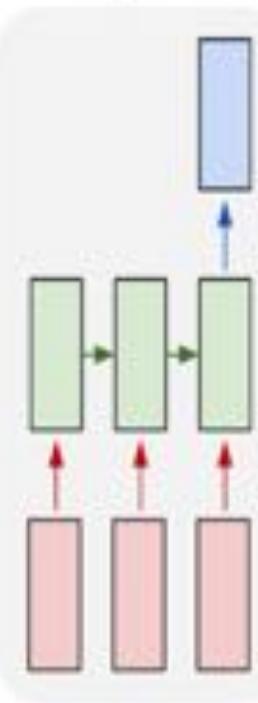
one to one



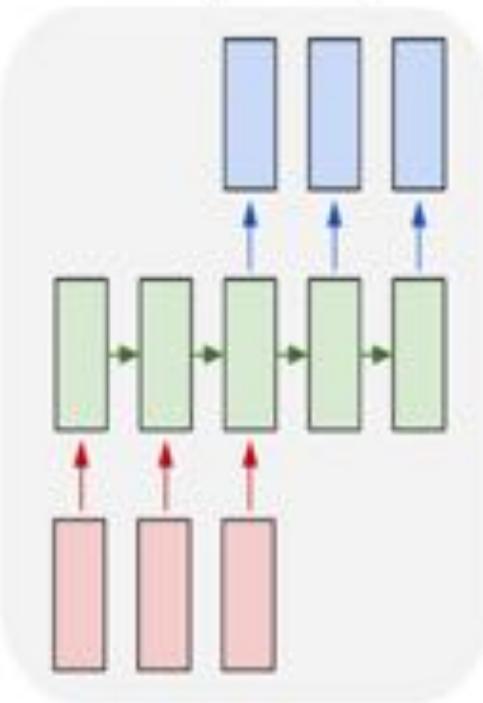
one to many



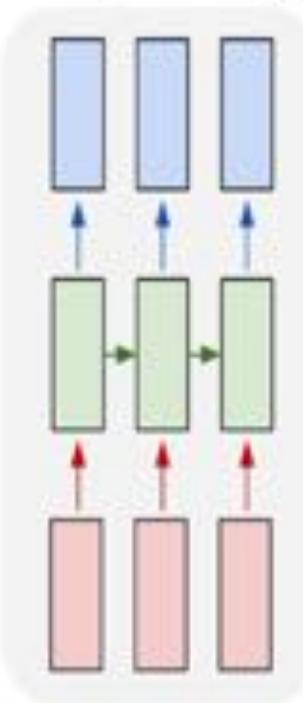
many to one



many to many



many to many

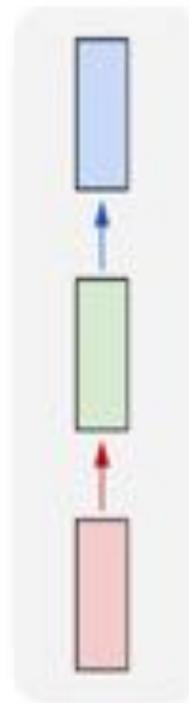


e.g. **Sentiment Classification**
sequence of words -> sentiment

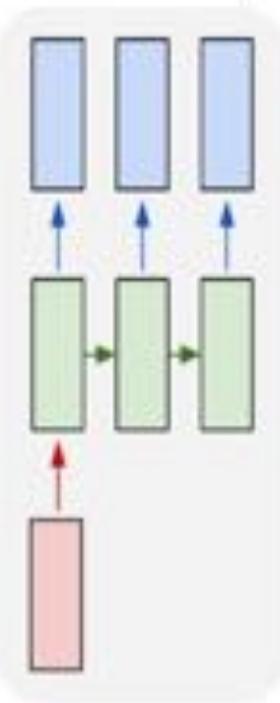


Recurrent Networks offer a lot of flexibility:

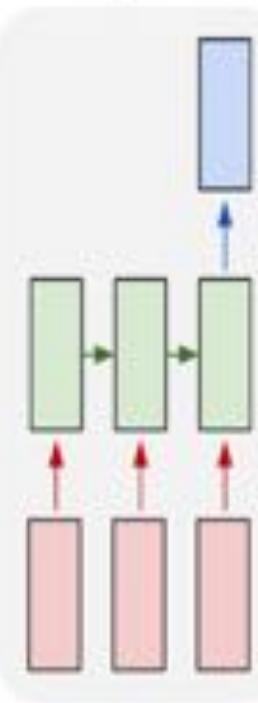
one to one



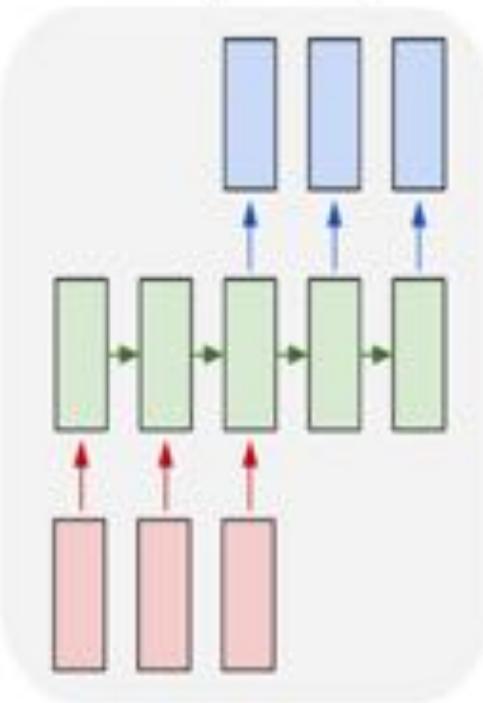
one to many



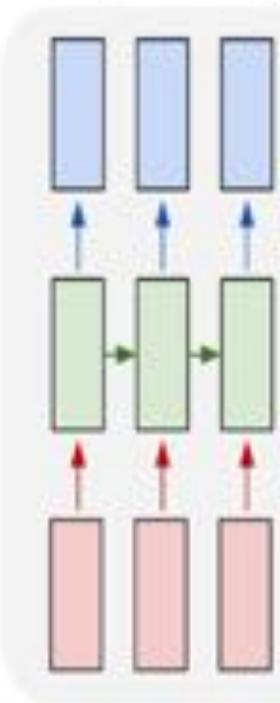
many to one



many to many



many to many

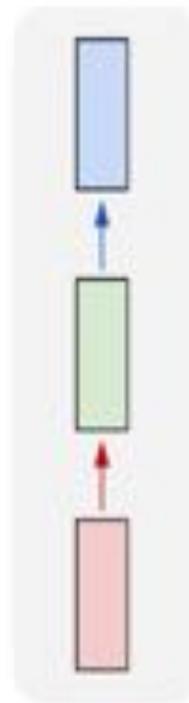


↑
e.g. Machine Translation
seq of words -> seq of words

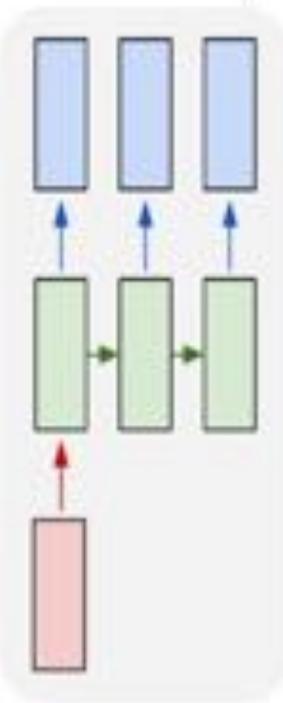


Recurrent Networks offer a lot of flexibility:

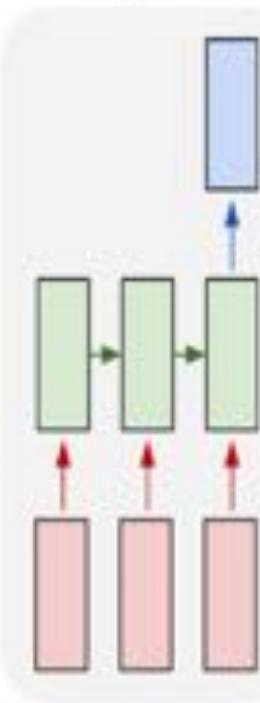
one to one



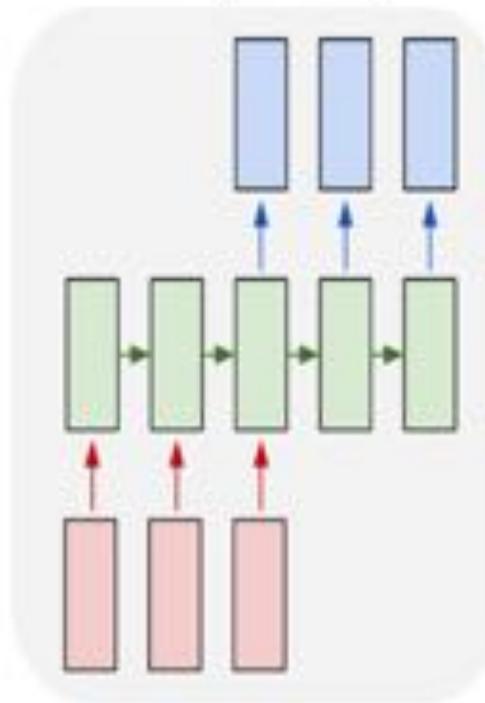
one to many



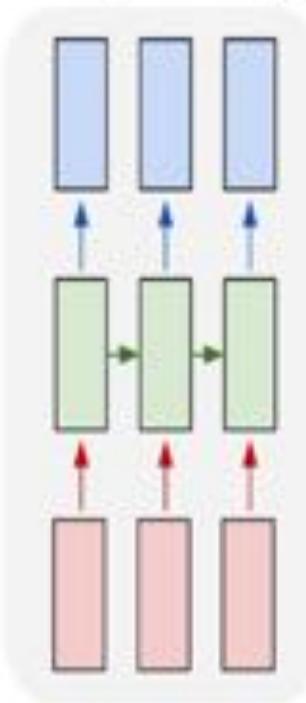
many to one



many to many



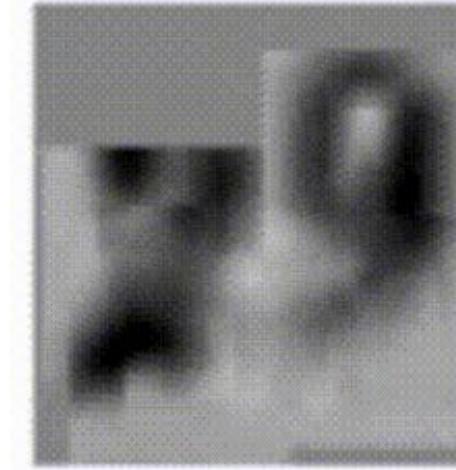
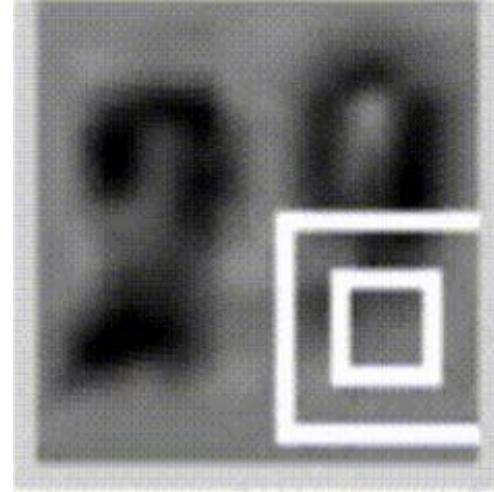
many to many



e.g. Video classification on frame level



Sequential Processing of fixed inputs

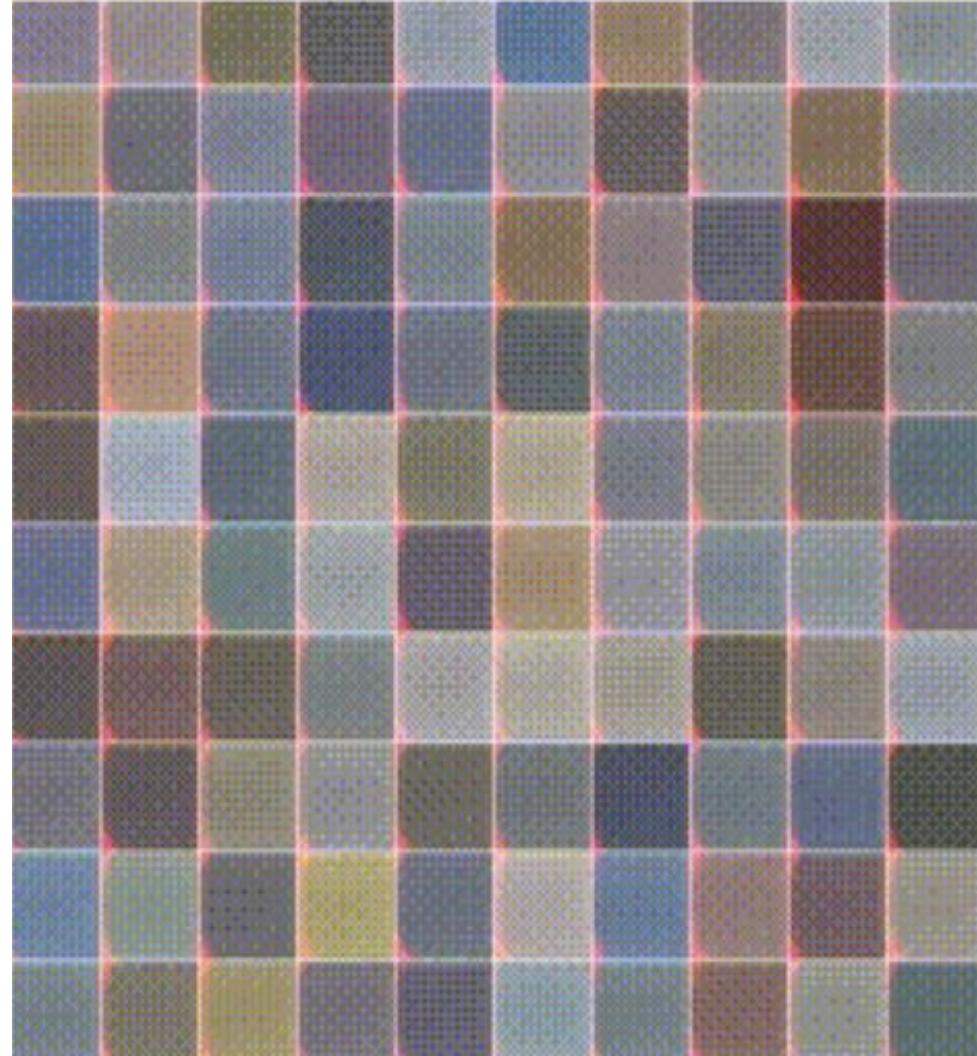


Multiple Object Recognition with
Visual Attention, Ba et al.

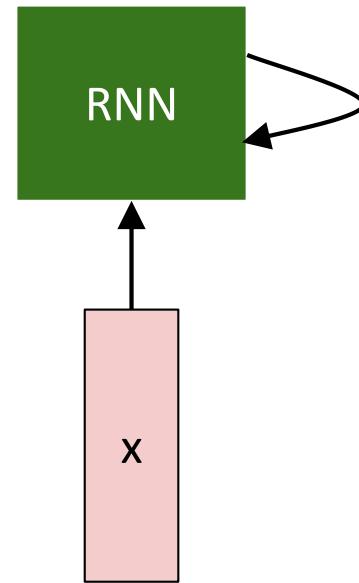


Sequential Processing of fixed inputs

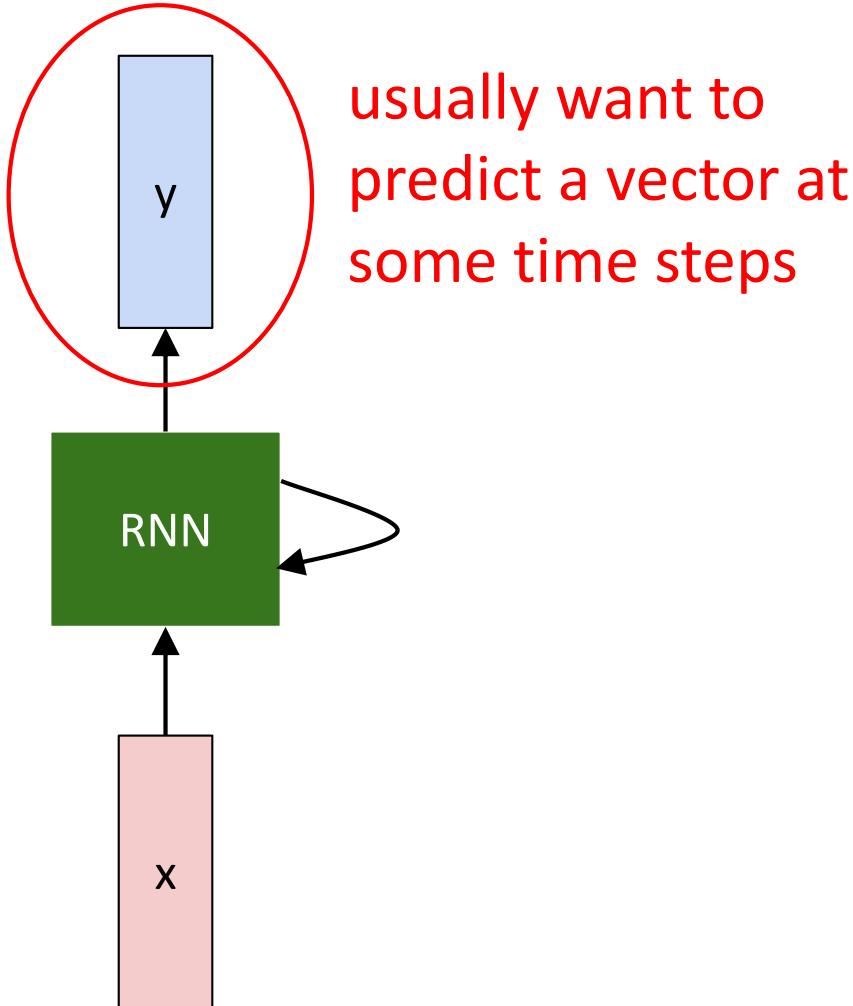
DRAW: A Recurrent
Neural Network For
Image Generation, Gregor
et al.



Recurrent Neural Network



Recurrent Neural Network

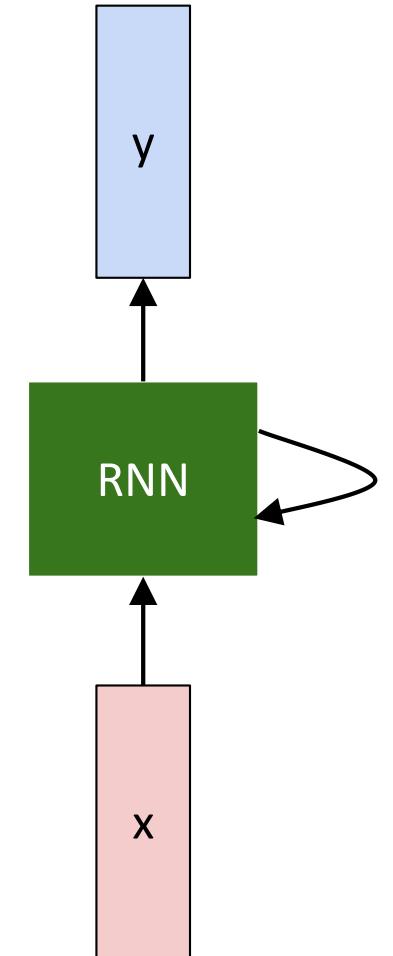


Recurrent Neural Network

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

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

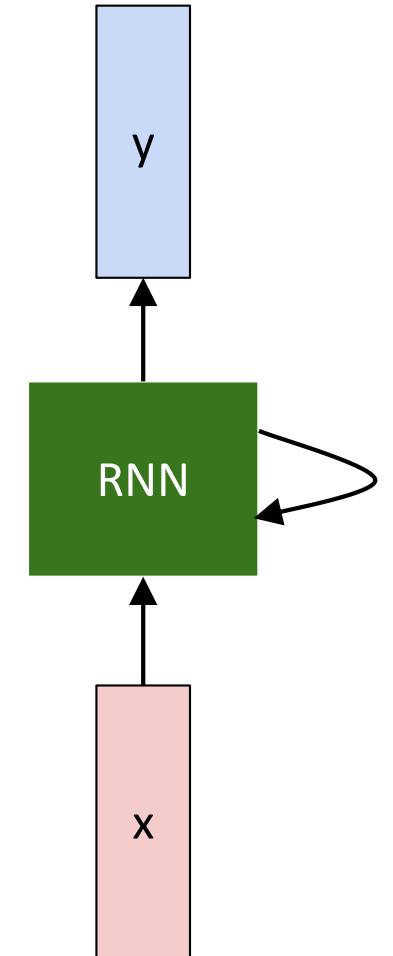
new state old state input vector at
 some function some time step
 with parameters W



Recurrent Neural Network

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

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

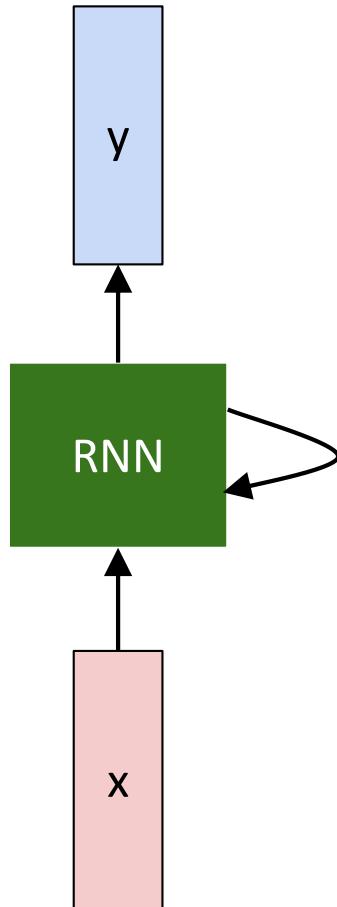


Notice: the same function and the same set of parameters are used at every time step.



(Vanilla) Recurrent Neural Network

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



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



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

$$y_t = W_{hy}h_t$$



Character-level language model example

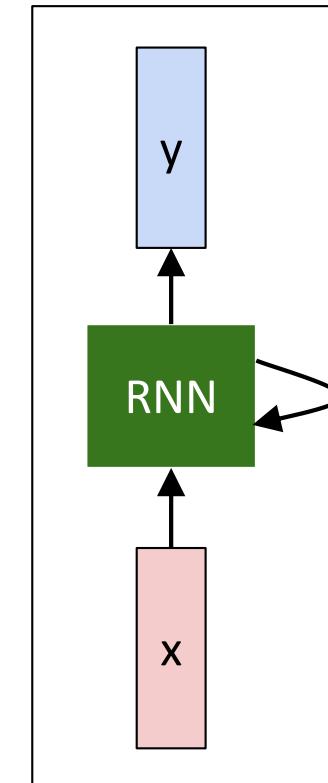
Vocabulary:

[h,e,l,o]

Example training

sequence:

“hello”



Character-level language model example

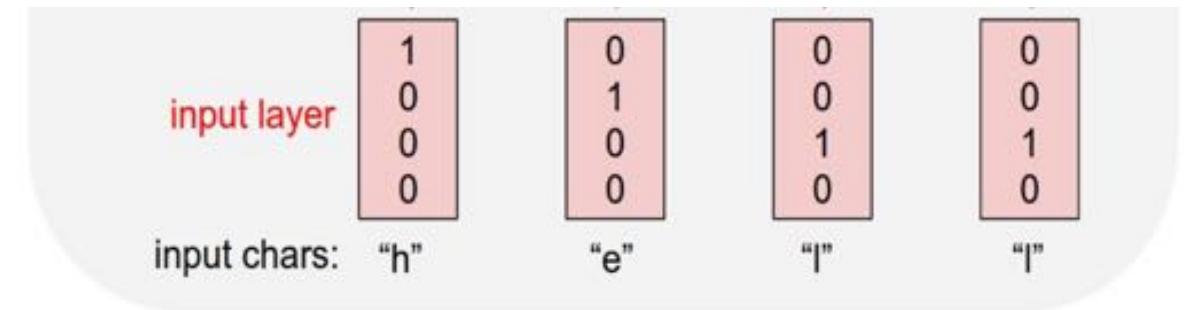
Vocabulary:

[h,e,l,o]

Example training

sequence:

“hello”



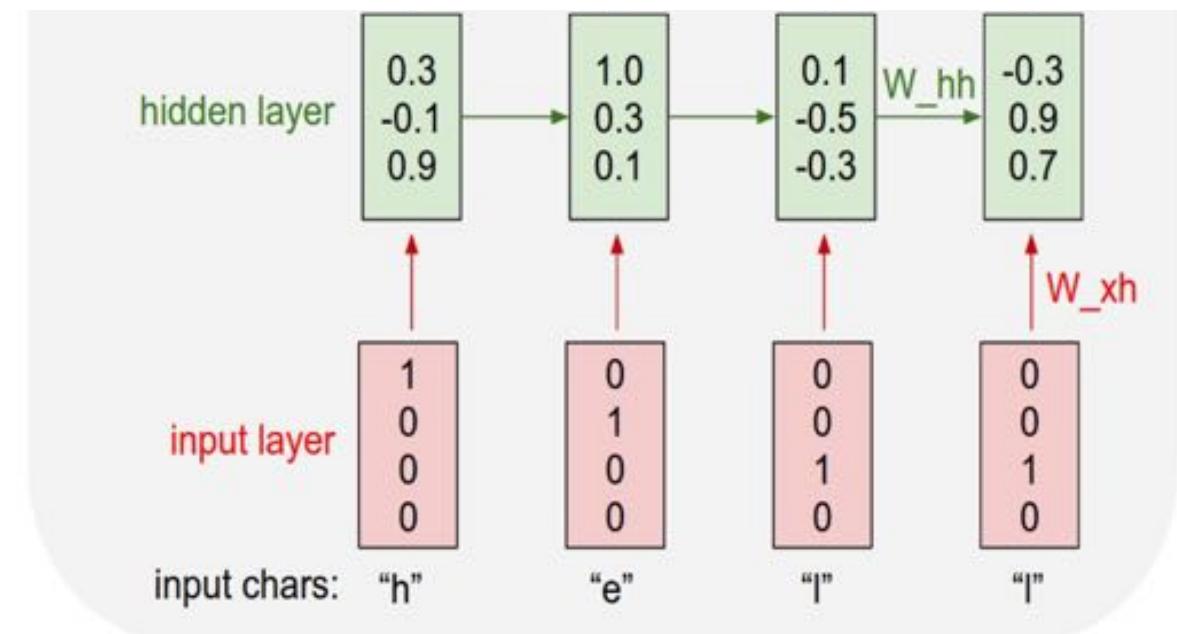
Character-level language model example

Vocabulary:

[h,e,l,o]

Example training
sequence:
“hello”

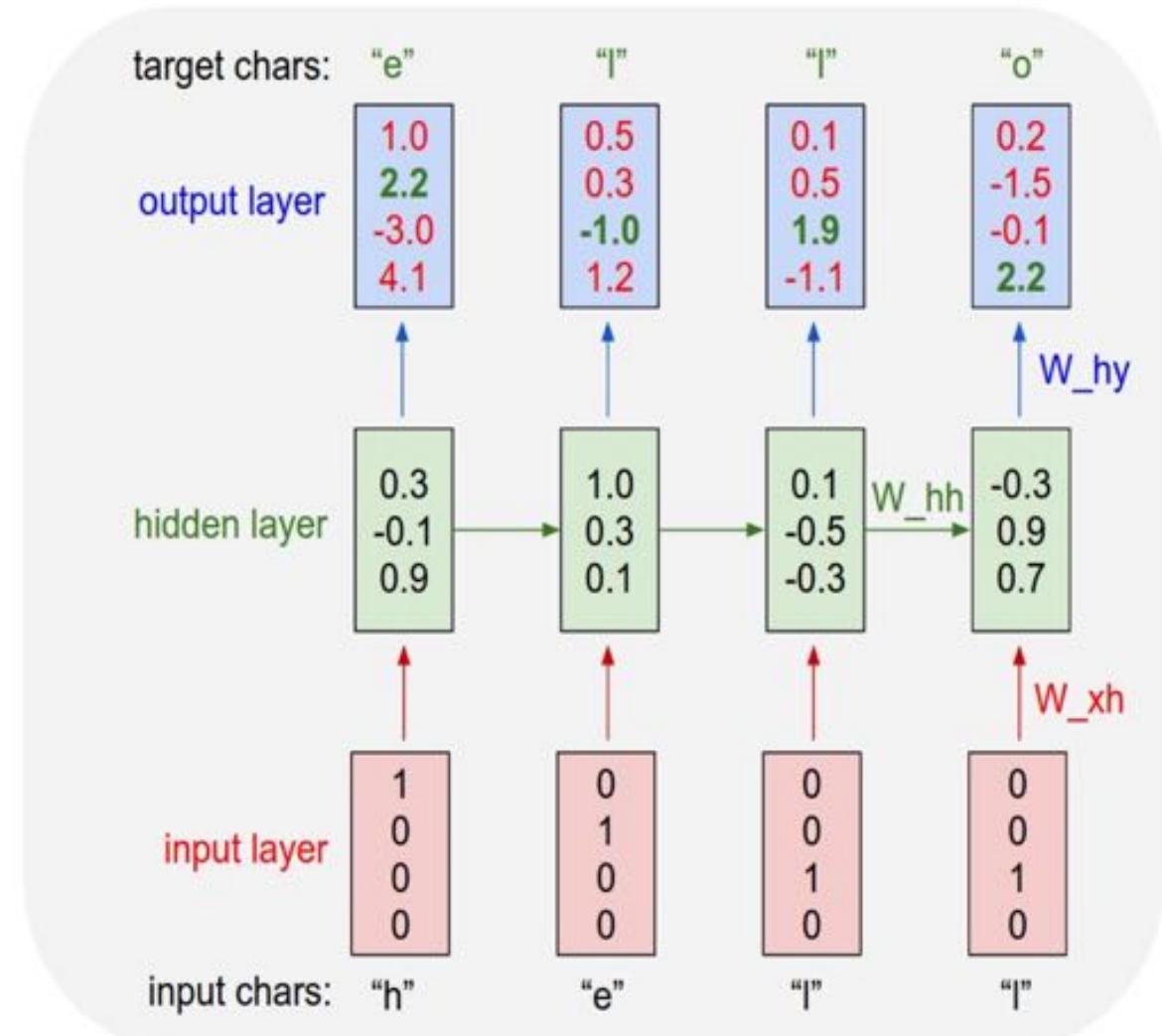
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”





[min-char-rnn.py](#) gist: 112 lines of Python

```
1 /**
2  * minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  * BSD license
4  */
5
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,ch in enumerate(chars) }
14 ix_to_char = { i:ch for i,ch in enumerate(chars) }
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # model parameters
22 wkh = np.random.rand(hidden_size, vocab_size)*0.02 + np.sqrt(0.01) # input to hidden
23 whh = np.random.rand(hidden_size, hidden_size)*0.01 + np.sqrt(0.01) # hidden to hidden
24 why = np.random.rand(vocab_size, hidden_size)*0.01 + np.sqrt(0.01) # hidden to output
25 bh = np.zeros([hidden_size, 1]) # hidden bias
26 by = np.zeros([vocab_size, 1]) # output bias
27
28 def lossFun(inputs, targets, hprev):
29     """
30     inputs,targets are both lists of integers.
31     hprev is NxD array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     xs, hs, ys, ps = {}, {}, {}, {}
35     hs[-1] = np.copy(hprev)
36     loss = 0
37     # forward pass
38     for t in range(len(inputs)):
39         xs[t] = np.zeros([vocab_size, 1]) + inputs[t] # become an 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(wkh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
42         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t])[targets[t]] # softmax (cross-entropy loss)
45
46         # backward pass: compute gradients going backwards
47         dprev, dhw, dwh, dyh = np.zeros_like(wkh), np.zeros_like(whh), np.zeros_like(why),
48         dbh, dy = np.zeros_like(bh), np.zeros_like(by)
49         dnext = np.zeros_like(hs[0])
50
51         for t in reversed(range(len(inputs))): # backward pass
52             dy = np.copy(ps[t])
53             dy[targets[t]] -= 1 # backprop into y
54             dyh += np.dot(dy, hs[t].T)
55             dy += dyh
56             dhw = (1 - hs[t]**2) * dy # backprop through tanh nonlinearity
57             dbh += dhw
58             dwh += np.dot(dhw, hs[t-1].T)
59             dnext = np.dot(whh.T, dhw)
60
61             for dparam in [dhw, dhw, dyh, dbh, dy]:
62                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
63
64     return loss, dhw, dwh, dyh, dbh, dy, hs[seq_length-1]
```

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



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

Data I/O



```

1 /**
2 Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be simple plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
12 char_to_ix = { ch:i for i,ch in enumerate(chars) }
13 ix_to_char = { i:ch for i,ch in enumerate(chars) }
14
15
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
21 # model parameters
22 Wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
23 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
24 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
25 bh = np.zeros(hidden_size, 1) # hidden bias
26 by = np.zeros(vocab_size, 1) # output bias
27
28 def loss(inputs, targets, hprev):
29     """
30     inputs, targets are both list of integers.
31     hprev is mx array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     x, h, y, hs = [0, 0, 0, 0]
35     hprev = np.copy(hprev)
36     loss = 0
37
38     # forward pass
39     for t in range(len(inputs)):
40         x = np.zeros((vocab_size, 1)) + encode_in_1-of-k(representation
41         x[inputs[t]] = 1)
42         h = hprev * np.dot(Wih, x) + np.dot(Whh, hprev) + bh
43         h[1:] = np.tanh(np.dot(Why, h[1:])) + by # hidden state
44         y[targets[t]] = np.exp(h[1:]) / np.sum(np.exp(h[1:])) # probabilities for next char
45         ps[t] = np.exp(y[1:])/np.sum(np.exp(y[1:])) # softmax (cross-entropy loss)
46         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
47         dloss = np.zeros_like(ps)
48         dloss[targets[t]] = 1 # backprop into y
49         dy = np.dot(Why, dloss) # backprop into y
50         dby += np.sum(dy[1])
51         dy = np.dot(dy[1:], np.transpose(Why))
52         dh = np.dot(Why, dy) # backprop into h
53         dh += np.tanh(np.dot(Whh, h[1:])) * np.dot(Whh, dh) # backprop through tanh nonlinearity
54         dWhh = np.dot(dh[1:], x[1:])
55         dWih = np.dot(dh[1:], np.transpose(x[1:]))
56         dbh = np.sum(dh[1:], axis=0)
57         dby = np.sum(dy[1:], axis=0)
58         dby += by
59
60     for param in [dWih, dWhh, dby, dbh, dby]:
61         np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
62     return loss, h, dWih, dWhh, dby, dbh, dby, hs[-1]
63
64 def sample(h, seed_ix, n):
65     """
66     sample a sequence of integers from the model
67     h is memory state, seed_ix is seed letter for first time step
68     """
69     if seed_ix is None:
70         x = np.zeros((vocab_size, 1))
71         x[seed_ix] = 1
72     else:
73         x = np.zeros((vocab_size, 1))
74         h = np.tanh(np.dot(Wih, x) + np.dot(Whh, h) + bh)
75         y = np.dot(Why, h) + by
76         p = np.exp(y) / np.sum(np.exp(y))
77         ix = np.random.choice(range(vocab_size), p=p.ravel())
78         x[0] = np.zeros((vocab_size, 1))
79         x[ix] = 1
80         seed_ix = ix
81
82     appendix.append(ix)
83     return seed_ix
84
85 n = 0, 0
86
87 x, dbh, dby = np.zeros_like(h), np.zeros_like(Whh), np.zeros_like(Why)
88 dbh, dby = np.zeros_like(h), np.zeros_like(Why) # memory variables for adaptive
89 smooth_loss = -np.log(1.0/vocab_size)*seq_length + loss at iteration 0
90 while True:
91     # sample from inputs (we're unrolling from left to right in steps seq_length long)
92     if print_length == len(data) or n == 0:
93         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
94         g = 0 # go from start of data
95         inputs = [char_to_ix[ch] for ch in data[g:g+seq_length]]
96         targets = [char_to_ix[ch] for ch in data[g+seq_length:g+2*seq_length]]
97
98     # sample from the model now and then
99     if n % 100 == 0:
100         sample_ix, inputs[g] = sample(h, inputs[g], seq_length)
101         txt = "...".join(ix_to_char[i] for i in sample_ix)
102         print "...".join(ix_to_char[i] for i in sample_ix)
103
104     # forward pass through characters in the seq_length window
105     loss, dh, dbh, dby, dbh, dby, hprev = loss_fn(inputs, targets, hprev)
106     smooth_loss = smooth_loss * 0.999 + loss * 0.001
107     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) + print progress
108
109     # perform parameter update with adam
110     for param, dparam, m, v in zip([h, wh, why, bh, by],
111                                   [dh, dWhh, dWhy, dbh, dby],
112                                   [dmh, dmWhh, dmWhy, dm_bh, dm_by],
113                                   [dvh, dvWhh, dvWhy, dbh, dby]):
114         m += dparam ** 2
115         param += dparam * adam_beta1 * m / (m + adam_beta2 * adam_epsilon)
116         param -= adam_beta2 * adam_beta1 * m / (m + adam_beta2 * adam_epsilon)
117
118     n += 1
119
120     g += seq_length + 1 # move data pointer
121     n += 1 # iteration counter

```

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

```

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

14 # Hyperparameters
15 hidden_size = 200 + size of hidden layer of neurons
16 seq_length = 25 # number of steps to unroll the RNN for learning_rate = 1e-1.

17 # Model parameters
18 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
19 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
20 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
21 bh = np.zeros((hidden_size, 1)) # hidden bias
22 by = np.zeros((vocab_size, 1)) # output bias

23 def lossfun(inputs, targets, hprev):
24     """
25     inputs,targets are both list of integers.
26     hprev is init array of initial hidden state
27     returns the loss, gradients on model parameters, and last hidden state
28     """
29     x, h, y, px = (0, 0, 0, 0)
30     hs = []
31     for t in range(len(inputs)):
32         x = inputs[t] # encode as 1-of-a representation
33         xh = np.zeros((vocab_size, 1)) # encode as 1-of-a representation
34         xh[inputs[t]] = 1
35         h = hprev * np.dot(Wxh, xh) + np.dot(Whh, h) + bh + hidden_bias
36         yh = np.dot(Why, h) / np.sum(np.exp(yh)) # probabilities for next char
37         ph = np.exp(yh) / np.sum(np.exp(ph)) # probabilities for next char
38         loss += -np.log(ph[targets[t]]) + softmax_cross_entropy_loss
39         dx = np.zeros_like(x) # backprop into x
40         dh = np.zeros_like(h) # backprop into h
41         dy = np.zeros_like(y) # backprop into y
42         dyt = np.zeros_like(yh) # backprop into yh
43         dhy = np.zeros_like(Why) # backprop into Why
44         dbh = np.zeros_like(bh) # backprop into bh
45         dby = np.zeros_like(by) # backprop into by
46         for t2 in reversed(range(len(inputs))):
47             dx += np.copy(dx[t2])
48             dyt += np.multiply(dy[t2], np.eye(vocab_size))
49             dh += np.multiply(dh[t2], np.eye(hidden_size))
50             dbh += np.multiply(dbh[t2], np.eye(hidden_size))
51             dby += np.multiply(dby[t2], np.eye(vocab_size))
52             for param in [dx, dh, dy, dbh, dby]:
53                 np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
54             return loss, dx, dh, dy, dbh, dby, np[seq[inputs[t]]-1]
55
56 sample=h, seed_ix, s:
57     """
58     sample a sequence of integers from the model
59     h is memory state, seed_ix is seed letter for first time step
60     """
61     x = np.zeros((vocab_size, 1))
62     if seed_ix > 1:
63         h = hprev
64     for t in range(n):
65         h = np.tanh(np.dot(Whh, h) + np.dot(Wxh, x) + np.dot(Why, y) + bh)
66         y = np.dot(Why, h) + by
67         p = np.exp(y) / np.sum(np.exp(y))
68         ix = np.random.choice(range(vocab_size), p=p.ravel())
69         x = np.zeros((vocab_size, 1))
70         x[ix] = 1
71         zero.append(ix)
72     return zero

73 n, p, g, R
74 m0, mbh, mhy = np.zeros_like(Whh), np.zeros_like(Wxh), np.zeros_like(Why)
75 mb, my = np.zeros_like(h), np.zeros_like(y) # memory variables for adapted softmax loss
76 smoth_loss = np.log(1/vocab_size)*seq_length + loss / n at iteration k
77 while True:
78     # forward pass (from left to right in steps seq_length long)
79     if p < seq_length - 1:
80         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
81         g = 0 # go from start of data
82         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
83         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
84
85         # sample from the model now and then
86         if n % 100 == 0:
87             sample_ix = sample(hprev, inputs[0], 200)
88             txt = '\n'.join(ix_to_char[x] for x in sample_ix)
89             print('...'.join(ix_to_char[x] for x in sample_ix))

90         # forward pass through characters through the net and fetch gradients
91         loss, dx, dh, dy, dbh, dy, dby, dhy = lossfun(inputs, targets, hprev)
92         smooth_loss = smooth_loss * 0.999 + loss * 0.001
93         if n % 100 == 0: print('iter %d, loss: %f, (%f)' % (n, smooth_loss))
94         print progress

95         # perform parameter update with Adagrad
96         for param, gradvars, mem in zip([Whh, Wxh, Why, h], [dh, dx, dy, dbh], [mbh, mb, mhy, my]):
97             mem += gradvars * gradvars
98             param -= learning_rate * gradvars / np.sqrt(mem + 1e-8) # integral update
99
100         n += 1 # increase data pointer
101         i += 1 # iteration counter
102
103         p += seq_length + move data pointer
104         n += 1 # iteration counter

```

Initializations



```

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

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

25 def lossfun(inputs, targets, hprev):
26     """
27     inputs,targets are both list of integers.
28     hprev is init array of initial hidden state
29     returns the loss, gradients on model parameters, and last hidden state
30     """
31     x, h, y, px = (0, 0, 0, 0)
32     hs = []
33     for t in range(len(inputs)):
34         x = inputs[t] # encode as 1-of-a representation
35         xh = np.zeros((vocab_size, 1)) # encode as 1-of-a representation
36         xh[inputs[t]] = 1
37         h = hprev * np.dot(Wxh, xh) + np.dot(Whh, h) + bh + hidden_bias
38         yh = np.dot(Why, h) / np.sum(np.exp(yh)) # probabilities for next char
39         ph = np.exp(yh) / np.sum(np.exp(ph)) # probabilities for next char
40         loss += -np.log(ph[targets[t]]) + softmax_cross_entropy_loss
41         dx = np.zeros_like(x) # backprop into x
42         dh = np.zeros_like(h) # backprop into h
43         dy = np.zeros_like(y) # backprop into y
44         dyt = np.zeros_like(yh) # backprop into yh
45         dhy = np.zeros_like(Why) # backprop into Why
46         dbh = np.zeros_like(bh) # backprop into bh
47         dby = np.zeros_like(by) # backprop into by
48         for t2 in reversed(range(len(inputs))):
49             dx += np.copy(dx[t2])
50             dyt += np.multiply(dy[t2], np.eye(vocab_size))
51             dh += np.multiply(dh[t2], np.eye(hidden_size))
52             dbh += np.multiply(dbh[t2], np.eye(hidden_size))
53             dby += np.multiply(dby[t2], np.eye(vocab_size))
54             for param in [dx, dh, dy, dbh, dby]:
55                 np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
56             return loss, dx, dh, dy, dbh, dby, np[seq[inputs[t]]-1]
57
58 sample=h, seed_ix, s:
59     """
60     sample a sequence of integers from the model
61     h is memory state, seed_ix is seed letter for first time step
62     """
63     x = np.zeros((vocab_size, 1))
64     if seed_ix > 1:
65         h = hprev
66     for t in range(n):
67         h = np.tanh(np.dot(Whh, h) + np.dot(Wxh, x) + np.dot(Why, y) + bh)
68         y = np.dot(Why, h) + by
69         p = np.exp(y) / np.sum(np.exp(y))
70         ix = np.random.choice(range(vocab_size), p=p.ravel())
71         x = np.zeros((vocab_size, 1))
72         zero.append(ix)
73     return zero

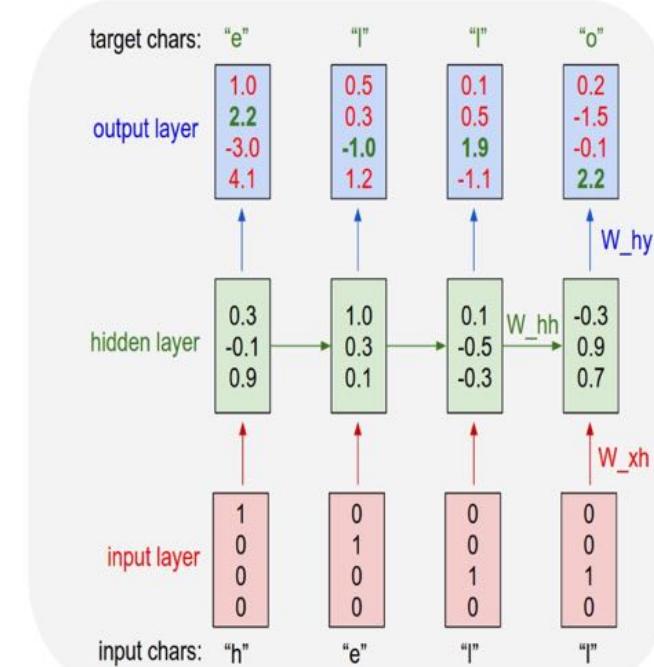
74 n, p, g, R
75 m0, mbh, mhy = np.zeros_like(Whh), np.zeros_like(Wxh), np.zeros_like(Why)
76 mb, my = np.zeros_like(h), np.zeros_like(y) # memory variables for adapted softmax loss
77 smoth_loss = np.log(1/vocab_size)*seq_length + loss / n at iteration k
78 while True:
79     # forward pass (from left to right in steps seq_length long)
80     if p < seq_length - 1:
81         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
82         g = 0 # go from start of data
83         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
84         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
85
86         # sample from the model now and then
87         if n % 100 == 0:
88             sample_ix = sample(hprev, inputs[0], 200)
89             txt = '\n'.join(ix_to_char[x] for x in sample_ix)
90             print('...'.join(ix_to_char[x] for x in sample_ix))

91         # forward pass through characters through the net and fetch gradients
92         loss, dx, dh, dy, dbh, dy, dhy = lossfun(inputs, targets, hprev)
93         smooth_loss = smooth_loss * 0.999 + loss * 0.001
94         if n % 100 == 0: print('iter %d, loss: %f, (%f)' % (n, smooth_loss))
95         print progress

96         # perform parameter update with Adagrad
97         for param, gradvars, mem in zip([Whh, Wxh, Why, h], [dh, dx, dy, dbh], [mbh, mb, mhy, my]):
98             mem += gradvars * gradvars
99             param -= learning_rate * gradvars / np.sqrt(mem + 1e-8) # integral update
100
101         n += 1 # increase data pointer
102         i += 1 # iteration counter
103
104         p += seq_length + move data pointer
105         n += 1 # iteration counter
106
107         g += 1 # iteration counter

```

recall:



min-char-rnn.py gist

```
1  #!/usr/bin/python
2  #
3  # Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
4  # BSD License
5  #
6  import numpy as np
7  import cPickle
8  import random
9  import time
10
11  # data I/O
12  data = open('input.txt', 'r').read() # should be simple plain text file
13  chars = list(set(data))
14  data_size, vocab_size = len(data), len(chars)
15  print 'data has %d characters, %d unique.' % (data_size, vocab_size)
16  char_to_ix = { ch:i for i,ch in enumerate(chars) }
17  ix_to_char = { i:ch for i,ch in enumerate(chars) }
18
19
20  # hyperparameters
21  hidden_size = 200 # size of hidden layer of neurons
22  seq_length = 25 # number of steps to unroll the RNN for
23  learning_rate = 0.1
24
25  # model parameters
26  Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
27  Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
28  Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
29  bh = np.zeros((hidden_size, 1)) # hidden bias
30  by = np.zeros((vocab_size, 1)) # output bias
31
32  def lossFun(inputs, targets, hprev):
33      """ 
34      inputs,targets are both list of integers.
35      hprev is init array of hidden state
36      returns the loss, gradients on model parameters, and last hidden state
37      """
38      x, h, y, px, ph = (0, 0, 0, 0, 0)
39      npx = 0
40      for t in range(len(inputs)):
41          x = inputs[t] # encode 27-of-a representation
42          xh = np.zeros((hidden_size, 1)) # encode 27-of-a representation
43          xh[0][inputs[t]] = 1
44          h = (np.dot(Wxh, xh) + np.dot(Whh, h) + bh) * hidden_size
45          yh = np.dot(Why, h) / np.sum(np.exp(yh)) # probabilities for next char
46          ph = np.dot(by, np.ones((1, 1))) + np.exp(-yh) # softmax (cross-entropy loss)
47          loss += -np.log(ph[targets[t], 0]) + softmax_cross_entropy_loss
48          npx += np.sum(px[targets[t], :])
49          # backprop next step's hidden state through softmax
50          dph = (why * np.zeros_like(ph)).dot(px[targets[t], :])
51          dbh = np.zeros_like(bh).dot(dph)
52          dby = np.zeros_like(by).dot(dph)
53          dh = np.zeros_like(h).dot(dph)
54          for param in [Wxh, Whh, Why, bh, by]:
55              np.clip(dparam, -5, 5) # clip to mitigate exploding gradients
56          return loss, npx, dph, dbh, dby, np.sum(inputs[:t])
57
58  def sample(h, seed_ix, n):
59      """ 
60      sample a sequence of integers from the model
61      h is memory state, seed_ix is seed letter for first time step
62      """
63      x = np.zeros((vocab_size, 1))
64      x[seed_ix] = 1
65
66      for t in range(n):
67          h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
68          y = np.dot(Why, h) + by
69          p = np.exp(y) / np.sum(np.exp(y))
70          ix = np.random.choice(range(vocab_size), p=p.ravel())
71          x = np.zeros((vocab_size, 1))
72          x[ix] = 1.0
73          h.append(ix)
74      return h
75
76  h, px, ph = np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((vocab_size, 1))
77  npx = 0
78  for i in range(100000):
79      if i % 100 == 0:
80          print 'iter %d' % i
81
82      n, p = 0, 0
83      mWxh, mWhh, mWhy = np.zeros_like(Wxh), 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
87      while True:
88          # prepare inputs (we're sweeping from left to right in steps seq_length long)
89          if p+seq_length+1 >= len(data) or n == 0:
90              hprev = np.zeros((hidden_size, 1)) # reset RNN memory
91              p = 0 # go from start of data
92
93              inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
94              targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
95
96
97          # sample from the model now and then
98          if n % 100 == 0:
99              sample_ix = sample(hprev, inputs[0], 200)
100             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
101             print '----\n %s \n----' % (txt, )
102
103
104          # forward seq_length characters through the net and fetch gradient
105          loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
106          smooth_loss = smooth_loss * 0.999 + loss * 0.001
107
108          if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
109
110
111          # perform parameter update with Adagrad
112          for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
113                                         [dWxh, dWhh, dWhy, dbh, dby],
114                                         [mWxh, mWhh, mWhy, mbh, mby]):
115
116              mem += dparam * dparam
117              param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
118
119
120              p += seq_length # move data pointer
121
122              n += 1 # iteration counter
123
124
125  # perform parameter update with Adagrad
126  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
127                               [dWxh, dWhh, dWhy, dbh, dby],
128                               [mWxh, mWhh, mWhy, mbh, mby]):
129
130      mem += dparam * dparam
131      param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
132
133
134  p += seq_length # move data pointer
135
136  n += 1 # iteration counter
137
```

Main loop



TECHNISCHE
UNIVERSITÄT
DARMSTADT



min-char-rnn.py gist

```
1  #!/usr/bin/python
2  #
3  # Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
4  # BSD License
5  #
6  # Import RNNPy API up
7  #
8  # DATA I/O
9  #
10 # data = open('input.txt', 'r').read() # should be simple plain text file
11 chars = list(set(data))
12 data_size, vocab_size = len(data), len(chars)
13 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
14 char_to_ix = { ch:i for i,ch in enumerate(chars) }
15 ix_to_char = { i:ch for i,ch in enumerate(chars) }
16
17 # hyperparameters
18 hidden_size = 200 # size of hidden layer of neurons
19 seq_length = 25 # number of steps to unroll the RNN for
20 learning_rate = 1e-1
21
22 # model parameters
23 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
24 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
25 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
26 bh = np.zeros((hidden_size, 1)) # hidden bias
27 by = np.zeros((vocab_size, 1)) # output bias
28
29 def lossFun(inputs, targets, hprev):
30     """
31     inputs,targets are both list of integers.
32     hprev is old array of initial hidden state
33     returns the loss, gradients on model parameters, and last hidden state
34     """
35     x, h, y, px, p = (0, 0, 0, 0, 0)
36     hprev = np.copy(hprev)
37     loss = 0
38     npx = 0
39     for t in range(len(inputs)):
40         x = inputs[t] # encode 27-of-a representation
41         xh = np.zeros((hidden_size, 1)) + encode_ix_1_of_k_representation(x)
42         nh = xh + h
43         wh = np.tanh(np.dot(Whh, nh[t]) + np.dot(Wxh, xh[t]) + bh[t]) + hidden_bias
44         yh = np.dot(Why, wh[t]) / np.sum(np.exp(yh[t])) + softmax_crossentropy_for_one_char(yh[t], targets[t])
45         loss += -np.log(yh[t]) * targets[t] + softmax_crossentropy_loss
46         px = np.exp(yh[t]) / np.sum(np.exp(yh[t])) # probabilities for next char
47         p = np.copy(px) # sample from model now and then
48         if t < len(inputs)-1:
49             dy = np.copy(px[t])
50             dy[targets[t]] -= 1 # backprop into y
51             dy += np.multiply(dy, nh[t])
52             dh = np.dot(Why.T, dy) + dhprev # backprop into h
53             dhraw = (1 - nh[t]**2) * nh[t] * dh # backprop through tanh nonlinearity
54             dh = dhraw
55             dWhh += np.outer(dhraw, nh[t].T)
56             dbh += np.outer(dhraw, bh[t].T)
57             dhprev = np.dot(Whh.T, dhraw)
58         for dparam in [dWhh, dbh, dWxh, dh]:
59             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
60     return loss, dWhh, dbh, dWxh, dh, dy, np.sum(inputs)-1
61
62     # sample(i, seed_ix, s):
63     #
64     # sample a sequence of integers from the model
65     # h is memory state, seed_ix is seed letter for first time step
66     #
67     x = np.zeros((vocab_size, 1))
68     if seed_ix > 1:
69         h = hprev
70     else:
71         h = np.zeros((hidden_size, 1))
72         h = np.tanh(np.dot(Whh, h) + np.dot(Wxh, x) + np.dot(bh, 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] = 1.0
78         hprev.append(ix)
79     return hprev
80
81     h, 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         if n % 100 == 0:
95             sample_ix = sample(hprev, inputs[0], 200)
96             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97             print '----\n%s\n----' % (txt, )
98
99         # forward seq_length characters through the net and fetch gradient
100        loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101        smooth_loss = smooth_loss * 0.999 + loss * 0.001
102        if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104        # perform parameter update with Adagrad
105        for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                      [dWxh, dWhh, dWhy, dbh, dby],
107                                      [mWxh, mWhh, mWhy, mbh, mby]):
108            mem += dparam * dparam
109            param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111        p += seq_length # move data pointer
112        n += 1 # iteration counter
113
114    # perform parameter update with Adagrad
115    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
116                                  [dWxh, dWhh, dWhy, dbh, dby],
117                                  [mWxh, mWhh, mWhy, mbh, mby]):
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    n += 1 # iteration counter
123
```

Main loop



TECHNISCHE
UNIVERSITÄT
DARMSTADT

min-char-rnn.py gist

```
1 Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
2 BSD License
3
4 import numpy as np
5
6 # DATA I/O
7
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 = 0.1
19
20 # model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     x, h, y, p = 0, 0, 0, 0
34     hprev = np.copy(hprev)
35     loss = 0
36     npx = 0
37     for t in range(len(inputs)):
38         x = inputs[t] # encode as 1-of-a representation
39         xh = np.zeros((hidden_size, 1)) + encode_ix_1_of_k_representation(x)
40         nh = h * np.tanh(Wxh*xh + Whh*h) + bh # hidden state
41         yh = np.dot(Why, nh) + by # unnormalized log probability for next char
42         ph = np.exp(yh) / np.sum(np.exp(yh)) # probabilities for next char
43         loss += -np.log(ph[targets[t]]) # softmax (cross-entropy loss)
44         npx += 1 # accumulate stats for softmax gradient
45         dnx = -ph[targets[t]] # softmax gradient
46         dny = dnx * Why # output layer gradient
47         dbh = dny * np.zeros_like(bh) # hidden bias gradient
48         dnh = dny * np.zeros_like(nh) # hidden state gradient
49         dWxh = dnh * np.zeros_like(Wxh) # input layer gradient
50         dWhh = dnh * np.zeros_like(Whh) # hidden layer gradient
51         dWhy = dny * np.zeros_like(Why) # output layer gradient
52         dbh += dny # accumulate gradients
53         dnh += dny # accumulate gradients
54         dWxh += dnx * np.zeros_like(Wxh) # input layer gradient
55         dWhh += dnh * np.zeros_like(Whh) # hidden layer gradient
56         dWhy += dny # output layer gradient
57     for param in [Wxh, Whh, Why, bh, by]:
58         np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
59     return loss, dnx, dnh, dbh, dny, dbh, dny, nh[seq_length-1]
60
61 def sample(h, seed_ix, n):
62     """
63     sample a sequence of integers from the model
64     h is memory state, seed_ix is seed letter for first time step
65     """
66     x = np.zeros((vocab_size, 1))
67     x[seed_ix] = 1
68
69     for t in range(n):
70         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
71         y = np.dot(Why, h) + by
72         p = np.exp(y) / np.sum(np.exp(y))
73         ix = np.random.choice(range(vocab_size), p=p.ravel())
74         x = np.zeros((vocab_size, 1))
75         x[ix] = 1.0
76         h.append(ix)
77     return h
78
79 h, dbh, dny = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
80 dbh, dny = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
81 smooth_loss = smooth_loss * 0.999 + loss * 0.001
82
83 if n % 100 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
84
85 # forward seq_length characters through the net and fetch gradient
86 loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
87 smooth_loss = smooth_loss * 0.999 + loss * 0.001
88
89 # perform parameter update with Adagrad
90 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
91                             [dWxh, dWhh, dWhy, dbh, dby],
92                             [mWxh, mWhh, mWhy, mbh, mbty]):
93     mem += dparam * dparam
94     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
95
96     p += seq_length # move data pointer
97     n += 1 # iteration counter
```

Main loop



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mbty = 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
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     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102     smooth_loss = smooth_loss * 0.999 + loss * 0.001
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                 [dWxh, dWhh, dWhy, dbh, dby],
107                                 [mWxh, mWhh, mWhy, mbh, mbty]):
108         mem += dparam * dparam
109         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
```

```

1  #!/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 = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is old array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     x, h, y, px, ph = (0, 0, 0, 0, 0)
34     hs1 = np.zeros((hidden_size, 1))
35     loss = 0
36     for t in range(len(inputs)):
37         x = np.zeros((vocab_size, 1)) # encode 1-of-k representation
38         x[inputs[t]] = 1
39         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh) # hidden state
40         y = np.dot(Why, h) + by # unnormalized log probability for next char
41         ph = np.exp(y) / np.sum(np.exp(y)) # probabilities for next char
42         loss += -np.log(ph[targets[t]]) # softmax (cross-entropy loss)
43         px = np.copy(ph[targets[t]])
44         ph[targets[t]] = 0 # backprop into x
45         ph += np.multiply(px, y)
46         dy = np.dot(Why.T, ph) # backprop into y
47         dh = np.dot(Whh.T, dy) + dh # backprop through tanh nonlinearity
48         dbh = np.sum(dy)
49         dWxh = np.outer(dh, x) # backprop into x
50         dWhh = np.outer(dh, h) # backprop into h
51         dWhy = np.outer(ph, h.T) # backprop into h
52         dbh = np.sum(dbh, 1) # average over all characters in batch
53         dWxh = np.clip(dWxh, -5, 5) # clip to mitigate exploding gradients
54         dWhh = np.clip(dWhh, -5, 5)
55         dWhy = np.clip(dWhy, -5, 5)
56         return loss, dx, dh, dbh, dWxh, dWhh, dWhy, np.zeros(inputs[1:])
57
58     def sample(h, seed_ix, n):
59         """
60         sample a sequence of integers from the model
61         h is memory state, seed_ix is seed letter for first time step
62         """
63         x = np.zeros((vocab_size, 1))
64         hs1 = np.zeros((hidden_size, 1))
65         for t in range(n):
66             h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
67             y = np.dot(Why, h) + by
68             p = np.exp(y) / np.sum(np.exp(y))
69             ix = np.random.choice(range(vocab_size), p=p.ravel())
70             x = np.zeros((vocab_size, 1))
71             x[ix] = 1.0
72             hs1.append(h)
73         return hs1
74
75     h, dx, dbh, dWxh, dWhh, dWhy = np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1))
76     dbh, dWxh, dWhh, dWhy = np.zeros_like(dbh), np.zeros_like(dWxh), np.zeros_like(dWhh), np.zeros_like(dWhy)
77     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
78
79     for p in range(seq_length): # forward pass
80         h, dx, dbh, dWxh, dWhh, dWhy, hprev = lossFun(inputs[p], targets[p], hprev)
81         smooth_loss += lossFun(inputs[p+1:p+seq_length+1], targets[p+1:p+seq_length+1], h)
82
83     smooth_loss /= seq_length # loss at iteration 0
84
85     while True:
86         # prepare inputs (we're sweeping from left to right in steps seq_length long)
87         if p+seq_length+1 >= len(data) or n == 0:
88             hprev = np.zeros((hidden_size, 1)) # reset RNN memory
89             p = 0 # go from start of data
90             inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91             targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93         # sample from the model now and then
94         if n % 100 == 0:
95             sample_ix = sample(hprev, inputs[0], 200)
96             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97             print '----\n%s\n----' % (txt, )
98
99         # forward seq_length characters through the net and fetch gradient
100 loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101 smooth_loss = smooth_loss * 0.999 + loss * 0.001
102 if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104 # perform parameter update with Adagrad
105 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                               [dWxh, dWhh, dWhy, dbh, dby],
107                               [mWxh, mWhh, mWhy, mbh, mbay]):
108     mem += dparam * dparam
109     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
113
114 # perform parameter update with Adagrad
115 for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
116                               [dWxh, dWhh, dWhy, dbh, dby],
117                               [mWxh, mWhh, mWhy, mbh, mbay]):
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     n += 1 # iteration counter
123
124

```

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, mbay = 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         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
105
106         # perform parameter update with Adagrad
107         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
108                                       [dWxh, dWhh, dWhy, dbh, dby],
109                                       [mWxh, mWhh, mWhy, mbh, mbay]):
110             mem += dparam * dparam
111             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
112
113             p += seq_length # move data pointer
114             n += 1 # iteration counter
115
116         # perform parameter update with Adagrad
117         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
118                                       [dWxh, dWhh, dWhy, dbh, dby],
119                                       [mWxh, mWhh, mWhy, mbh, mbay]):
120             mem += dparam * dparam
121             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
122
123             p += seq_length # move data pointer
124             n += 1 # iteration counter
125
126

```

```

1  #!/usr/bin/python
2  # Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  # BSD License
4  #
5  # Import NumPy and IPython
6  #
7  # DATA I/O
8  #
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, ch in enumerate(chars) }
14 # ix_to_char = { i: ch for i, ch in enumerate(chars) }
15 #
16 # Hyperparameters
17 hidden_size = 200 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # Model parameters
22 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
23 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
24 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
25 bh = np.zeros((hidden_size, 1)) # hidden bias
26 by = np.zeros((vocab_size, 1)) # output bias
27
28 def lossFun(inputs, targets, hprev):
29     """
30     Inputs: targets are both list of integers.
31     hprev is not array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     xh, yt, ph, ps = (0, 0, 0, 0)
35     hs1 = []
36     for t in range(len(inputs)):
37         xh[t] = np.zeros((vocab_size, 1)) # encode 2D 1-of-K representation
38         xt[t] = inputs[t].copy() + 1
39         xh[t] = np.dot(Wxh, xt[t]) + np.dot(bh, hs[t-1]) + ph
40         yt[t] = np.dot(Why, xh[t]) + by # unnormalized log probability for next char
41         ph = np.exp(yt[t]) / np.sum(np.exp(yt[t])) # probabilities for next char
42         loss += -np.log(ph[targets[t], 0]) + softmax_cross_entropy_loss
43         # backward pass: compute gradients going backwards
44         dph = np.zeros_like(ph)
45         dph[targets[t], 0] = 1 # backprop into y
46         dy = np.zeros_like(dy)
47         dy += np.dot(Why.T, dph) # backprop into h
48         dhs = (1 - hs[t] * hs[t]) * dy # backprop through tanh nonlinearity
49         dph = np.dot(dhhs, dx[t].T)
50         dph += np.dot(dhhs, bh[1:, T])
51         dhhs = np.dot(Whh, dph)
52         for dh in [dph, dhs, dph, dy, dhs]:
53             clip_dparam, _ = clip_to_maxnorm(dh)
54             clip_dparam, _, _ = clip_to_maxnorm(clip_dparam)
55         return loss, hs1, ph, dy, hs[t], inputs[t-1]
56     net.sample(hs[-1], s)
57     """
58     sample a sequence of integers from the model
59     h is memory state, seed_ix is seed letter for first time step
60     """
61     x = np.zeros((vocab_size, 1))
62     if seed_ix is None:
63         s = 1
64     else:
65         for i in range(n):
66             h = np.tanh(np.dot(Whh, x) + np.dot(bh, h) + bh)
67             y = np.dot(Why, h) + by
68             p = np.exp(y) / np.sum(np.exp(y))
69             ix = np.random.choice(range(vocab_size), p=p.ravel())
70             x = np.zeros((vocab_size, 1))
71             x[ix] = 1
72             s.append(ix)
73     return s
74
75 A, p = R, 0
76 mbh, mby = np.zeros_like(bh), np.zeros_like(bh), np.zeros_like(by)
77 mbh, mby = np.zeros_like(Why), np.zeros_like(Why) # memory variables for Adagrad
78 smooth_loss = smooth_loss * 0.999 + loss * 0.001
79 while True:
80     # forward sweep (unroll unrolled sequence from left to right in steps seq_length long)
81     n, p = 0, 0
82     mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83     mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84     smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85     while True:
86         # prepare inputs (we're sweeping from left to right in steps seq_length long)
87         if p+seq_length+1 >= len(data) or n == 0:
88             hprev = np.zeros((hidden_size,1)) # reset RNN memory
89             p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93         # sample from the model now and then
94         if n % 100 == 0:
95             sample_ix = sample(hprev, inputs[0], 200)
96             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97             print '----\n%s\n----' % (txt, )
98
99         # forward seq_length characters through the net and fetch gradient
100        loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101        smooth_loss = smooth_loss * 0.999 + loss * 0.001
102        if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104        # perform parameter update with Adagrad
105        for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                     [dWxh, dWhh, dWhy, dbh, dby],
107                                     [mWxh, mWhh, mWhy, mbh, mby]):
108            mem += dparam * dparam
109            param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111        p += seq_length # move data pointer
112        n += 1 # iteration counter

```

Main loop



```

1 Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
2 BSD license
3
4 import numpy as np
5
6 # DATA I/O
7 data = open('input.txt', 'r').read() # should be simple plain text file
8 chars = list(set(data))
9 print('data has %d characters, %d unique.' % (data_size, vocab_size))
10 char_to_ix = { ch:i for i,ch in enumerate(chars) }
11 ix_to_char = { i:ch for i,ch in enumerate(chars) }
12
13 # hyperparameters
14 hidden_size = 500 # size of hidden layer of neurons
15 seq_length = 25 # number of steps to unroll the RNN for
16 learning_rate = 1e-1
17
18 # model parameters
19 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
20 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
21 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
22 bh = np.zeros((hidden_size, 1)) + hidden_size*0.01 # hidden bias
23 by = np.zeros((vocab_size, 1)) + output_bias
24
25 def lossFun(inputs, targets, hprev):
26     """
27     inputs,targets are both list of integers.
28     hprev is init array of initial hidden state
29     returns the loss, gradients on model parameters, and last hidden state
30     """
31     x, h, y, px = (0, 0, 0, 0)
32     hs = []
33     for t in xrange(len(inputs)):
34         x = np.zeros((vocab_size,1)) # encode in 1-of-k representation
35         x[inputs[t]] = 1
36         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, hprev) + bh) # hidden state
37         y = np.dot(why, h) + by # unnormalized log probabilities for next chars
38         p = np.exp(y) / np.sum(np.exp(y)) # probabilities for next chars
39         loss += -np.log(p[targets[t]]) # softmax (cross-entropy loss)
40         dx, dh = np.zeros_like(x), np.zeros_like(h) # backward pass: compute gradients going backwards
41         dwhy = np.zeros_like(why), np.zeros_like(by)
42         dbh = np.zeros_like(bh), np.zeros_like(bh)
43         dloss = np.zeros_like(h)
44         for dy in [dx, dh, dwhy, dbh, dloss]:
45             np.clip(dy, -5, 5, out=dy) # clip to mitigate exploding gradients
46         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
47         return loss, dx, dh, dwhy, dbh, dy, hs[-1]
48
49 def sample(h, seed_ix, n):
50     """
51     sample a sequence of integers from the model
52     h is memory state, seed_ix is seed letter for first time step
53     """
54     x = np.zeros((vocab_size, 1))
55     x[seed_ix] = 1
56     for t in xrange(n):
57         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
58         y = np.dot(why, h) + by
59         p = np.exp(y) / np.sum(np.exp(y))
60         ix = np.random.choice(range(vocab_size), p=p.ravel())
61         x = np.zeros((vocab_size, 1))
62         x[ix] = 1
63         hprev.append(h)
64     return hprev
65
66 h, p = None, None
67 min, mwh, mbh, mwhy, mby = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(bh)
68 mbh, mwhy = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
69 smoth_loss = np.log(1/vocab_size)/seq_length * loss / seq_length
70 while True:
71     # forward pass (unrolled over time)
72     # print progress (every 1000 iterations from left to right in steps seq_length long)
73     if p+seq_length > len(data) or n == 0:
74         hprev = np.zeros((hidden_size,1)) # reset RNN memory
75         g = 0 # go from start of data
76         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
77         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
78
79     # sample from the model now and then
80     if n % 100 == 0:
81         sample_ix = sample(hprev, inputs[0], 200)
82         txt = "'".join(ix_to_char[i] for i in sample_ix)
83         print('...An %d ...' % (g))
84
85     # forward unrolled characters through the net and fetch gradients
86     loss, dx, dh, dwhy, dbh, dy, hprev = lossFun(inputs, targets, hprev)
87     smooth_loss = smooth_loss * 0.999 + loss * 0.001
88     if n % 100 == 0: print('iter %d, loss %f, smooth loss %f' % (n, smooth_loss)) # print progress
89
90     # perform parameter update with Adagrad
91     for param, dparam, mem in zip([h, wh, bh, why, by],
92                                  [dh, dwhh, dbh, dwhy, dy],
93                                  [mwh, mbh, mwhy, mby]):
94         mem += dparam * dparam # element-wise square of gradients
95         param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # integral update
96
97     n += 1 # iteration counter
98     m += 1 # iteration counter

```

Loss function

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



```

27     def lossFun(inputs, targets, hprev):
28         """
29             inputs,targets are both list of integers.
30             hprev is Hx1 array of initial hidden state
31             returns the loss, gradients on model parameters, and last hidden state
32             """
33             xs, hs, ys, ps = {}, {}, {}, {}
34             hs[-1] = np.copy(hprev)
35             loss = 0
36             # forward pass
37             for t in xrange(len(inputs)):
38                 xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39                 xs[t][inputs[t]] = 1
40                 hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41                 ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
42                 ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43                 loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44
45             # backward pass: compute gradients going backwards
46             dwxh, dwhh, dwhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(why)
47             dbh, dby = np.zeros_like(bh), np.zeros_like(by)
48             dhnext = np.zeros_like(hs[0])
49             for t in reversed(xrange(len(inputs))):
50                 dy = np.copy(ps[t])
51                 dy[targets[t]] -= 1 # backprop into y
52                 dwhy += np.dot(dy, hs[t].T)
53                 dhnext += np.dot(dwhy, hs[t].T)
54                 dy[targets[t]] += 1 # backprop into y
55                 dby += np.dot(dy, hs[t].T)
56                 dwhh += np.dot(dby, hs[t-1].T)
57                 dbh += np.dot(dby, hs[0].T)
58                 dwxh += np.dot(dwhh, xs[t].T)
59                 dwhh += np.dot(dwhh, hs[t-1].T)
60                 dhnext = np.dot(why.T, dhnext) + dhnext # backprop into h
61                 dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
62                 dbh += dhraw
63             # perform parameter update with Adagrad
64             for param, dparam, mem in zip([h, wh, bh, why, by],
65                                         [dh, dwhh, dbh, dwhy, dy],
66                                         [mwh, mbh, mwhy, mby]):
67                 mem += dparam * dparam # element-wise square of gradients
68                 param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # integral update
69
70             return loss, dwxh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]

```

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 RNNPy API 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 print('data has %d characters, %d unique.' % (data_size, vocab_size))
11 char_to_ix = { ch:i for i,ch in enumerate(chars) }
12 ix_to_char = { i:ch for i,ch in enumerate(chars) }

13 # hyperparameters
14 hidden_size = 500 # size of hidden layer of neurons
15 seq_length = 25 # number of steps to unroll the RNN for
16 learning_rate = 1e-1

17 # model parameters
18 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
19 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
20 why = np.random.rand(vocab_size, hidden_size)*0.01 # hidden to output
21 bh = np.zeros((vocab_size, 1)) + hidden_size*0.01 # hidden bias
22 by = np.zeros((vocab_size, 1)) # output bias

23 def lossFun(inputs, targets, hprev):
24     """
25     inputs,targets are both list of integers.
26     hprev is np array of initial hidden state
27     returns the loss, gradients on model parameters, and last hidden state
28     """
29
30     xs, hs, ys, ps = {}, {}, {}, {}
31     hs[-1] = np.copy(hprev)
32     loss = 0
33     # forward pass
34     for t in xrange(len(inputs)):
35         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
36         xs[t][inputs[t]] = 1
37         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
38         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
39         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
40
41     # backward pass
42     for t in reversed(xrange(len(inputs))):
43         dy = np.copy(ps[t])
44         dy[targets[t]] -= 1 # backprop into y
45         dhy += np.multiply(dy, why.T)
46         dWxh += np.dot(xs[t].T, dy) # backprop into input-to-hidden
47         dWhh += np.dot(hs[t-1].T, dy) * np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # backprop through tanh nonlinearity
48         dbh += np.sum(dy, axis=0)
49         dby += np.sum(dy, axis=0)
50         dhs[t] = np.dot(why.T, dy) # backprop into previous hidden state
51         for param in [dWxh, dWhh, dbh, dby]:
52             np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
53     return loss, xs, hs, dhs, dhy, dbh, dby, ps[seq(inputs)-1]

54 def sample(h, seed_ix, n):
55     """
56     sample a sequence of integers from the model
57     h is memory state, seed_ix is seed letter for first time step
58     """
59     x = np.zeros((vocab_size, 1))
60     x[seed_ix] = 1
61
62     for t in range(n):
63         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
64         y = np.dot(why, h) + by
65         p = np.exp(y) / np.sum(np.exp(y))
66         ix = np.random.choice(range(vocab_size), p=p.ravel())
67         x = np.zeros((vocab_size, 1))
68         x[ix] = 1
69         seed_ix = ix
70     return seed_ix

71 n, p = 0, 0
72 min_n, max_n, min_y, max_y = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
73 dbh, dby = np.zeros_like(wh), np.zeros_like(why) # memory variables for Adagrad
74 smoth_loss = np.log(1/vocab_size)*seq_length + loss at iteration 0
75 while True:
76     # forward pass (iterative)
77     for p in range(n, seq_length):
78         hprev = h # remember h
79         if p < seq_length - 1:
80             hprev = np.zeros((hidden_size, 1)) # reset RNN memory
81             g = 0 # go from start of data
82             inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
83             targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
84
85         # sample from the model now and then
86         if n % 100 == 0:
87             sample_ix = sample(hprev, inputs[0], 200)
88             txt = '\n'.join(ix_to_char[i] for i in sample_ix)
89             print('...'.join(ix_to_char[i] for i in sample_ix))

90         # forward one length characters through the net and fetch gradients
91         loss, dhs, dhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
92         smooth_loss = smooth_loss * 0.999 + loss * 0.001
93         if n % 100 == 0:
94             print('iter %d, loss: %f' % (n, smooth_loss))
95             print progress
96
97         # perform parameter update with Adagrad
98         for param, dparam, mem in zip([Wxh, wh, why, bh],
99                                       [dWxh, dWhh, dwhy, dbh],
100                                      [min_n, max_n, min_y, max_y]):
101             mem += dparam * dparam # integral update
102             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # integral update
103
104         n += 1 # iteration counter
105         p += 1 # iteration counter

```



```

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

```

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

$$y_t = W_{hy}h_t$$

Softmax classifier

min-char-rnn.py gist

```

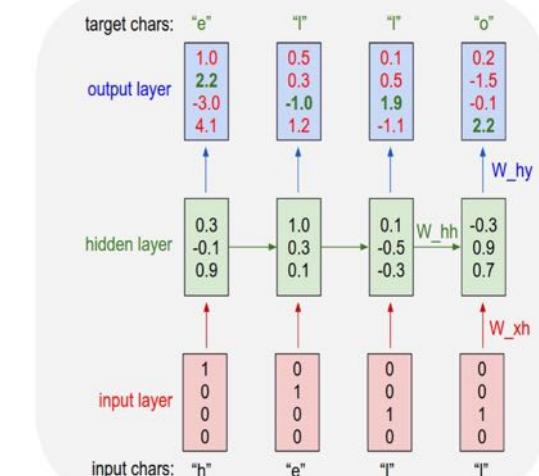
1  #
2  Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  #
5  import numpy as np
6
7  # DATA I/O
8
9  data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print('data has %d characters, %d unique.' % (data_size, vocab_size))
13 char_to_ix = { ch:i for i,ch in enumerate(chars) }
14 ix_to_char = { i:ch for i,ch in enumerate(chars) }
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 0.1
20
21 # model parameters
22 Wih = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
23 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
24 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
25 bi = np.zeros((hidden_size, 1)) # hidden bias
26 bo = np.zeros((vocab_size, 1)) # output bias
27
28 def lossfun(inputs, targets, hprev):
29     """
30     inputs,targets are both list of integers.
31     hprev is init array of initial hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     x, h, y, ps = (0, 0, 0, 0)
35     hs = []
36     for t in xrange(len(inputs)):
37         xi = np.zeros((vocab_size, 1)) # encode 1-of-k representation
38         xi[inputs[t]] = 1
39         hi = np.tanh(np.dot(Wih, xi) + np.dot(Whh, h) + bi) # hidden state
40         yi = np.dot(Why, hi) + bo # probabilities for next char
41         pi = yi / np.exp(yi) # softmax for next char
42         loss += -np.log(pi[targets[t], 0]) # softmax cross-entropy loss
43         loss += -np.log(pi[targets[t], 0]) # softmax cross-entropy loss
44
45         dhi, dWih, dWhy = np.zeros_like(hi), np.zeros_like(Wih), np.zeros_like(Why)
46         dWih = np.zeros_like(Wih)
47         dWhy = np.zeros_like(Why)
48         dbi = np.zeros_like(bi)
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
54             dh = np.dot(Why.T, dy) + dhnxt # backprop into h
55             ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
56             dWih += np.dot(ddraw, xs[t].T)
57             dWih += np.dot(dhraw, hs[t-1].T)
58             dhnxt = np.dot(Whh.T, dhraw)
59             for dparam in [dWih, dWhh, dWhy, dWih, dWhy]:
60                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61             return loss, dWih, dWhh, dWhy, dWih, dWhy, hs[len(inputs)-1]
62
63 def sample(hprev, 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
71     for t in xrange(n):
72         h = np.tanh(np.dot(Wih, x) + np.dot(Whh, h) + bi)
73         y = np.dot(Why, h) + bo
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         hprev.append(ix)
79     return hprev
80
81 n, p = 0, 0
82 max_ih, min_ih = np.zeros_like(Whh), np.zeros_like(Whh), np.zeros_like(Why)
83 ih, hy = np.zeros_like(Whh), np.zeros_like(Why) # memory variables for adapted
84 smooth_loss = np.log(1.0/vocab_size)*seq_length + loss at iteration 0
85 while True:
86     if p > seq_length: break # we're moving from left to right in steps seq_length long
87     if p+seq_length > len(data) or n > 100:
88         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
89         g = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p-p:seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[i] for i in sample_ix)
97         print('...An %d ...' % (n))
98
99     # forward pass: compute gradients going forwards
100    loss, dWih, dWhh, dWhy, dWih, dWhy, hprev = lossfun(inputs, targets, hprev)
101    smooth_loss = smooth_loss * 0.999 + loss * 0.001
102    if n % 100 == 0: print('iter %d, loss %f, smooth loss %f' % (n, smooth_loss))
103
104    # perform parameter update with Adagrad
105    for param, dparam, mem in zip([Wih, Whh, Why, bi],
106                                 [dWih, dWhh, dWhy, dbi],
107                                 [max_ih, min_ih, mem, dWih]):
108        mem += dparam * dparam
109        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # integral update
110
111    n += seq_length + 1 # move data pointer
112    n += 1 # iteration counter

```

```

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

```



recall:

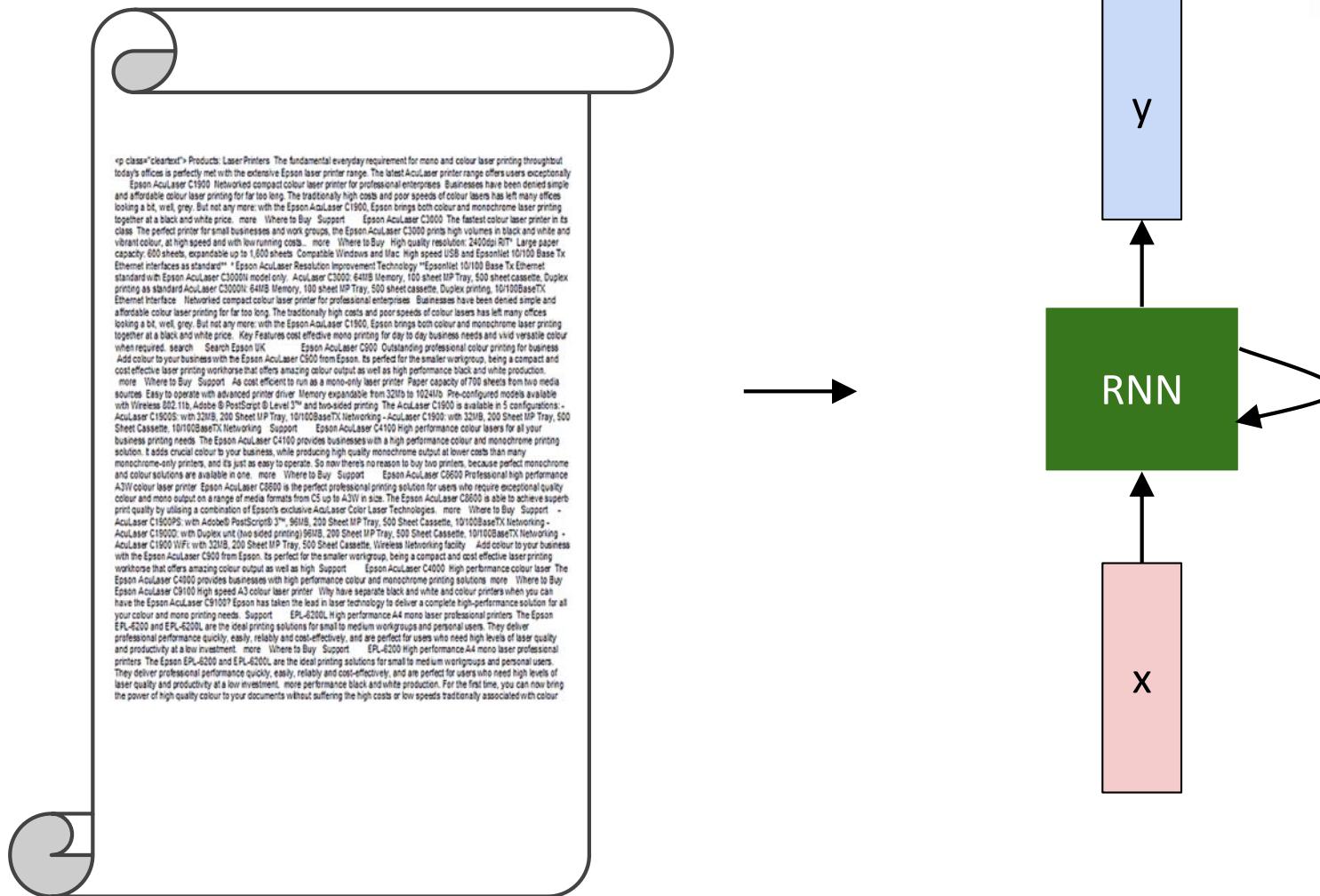
min-char-rnn.py gist

```
1  #!/usr/bin/python
2  #
3  # Minimal character-level vanilla RNN model. Written by Andrej Karpathy (@karpathy)
4  # BSD License
5  #
6  # Import RNNPy API up
7  #
8  # data I/O
9  #
10 # data = open("input.txt", "r").read() # should be simple plain text file
11 # chars = list(set(data))
12 # data_size, vocab_size = len(data), len(chars)
13 # print('data has %d characters, %d unique.' % (data_size, vocab_size))
14 # char_to_ix = { ch:i for i,ch in enumerate(chars) }
15 # ix_to_char = { i:ch for i,ch in enumerate(chars) }
16 #
17 #
18 # Hyperparameters
19 # hidden_size = 100 # size of hidden layer of neurons
20 # seq_length = 25 # number of steps to unroll the RNN for
21 # learning_rate = 0.1
22 #
23 # Model parameters
24 # Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
25 # Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
26 # Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
27 # bh = np.zeros((hidden_size, 1)) # hidden bias
28 # by = np.zeros((vocab_size, 1)) # output bias
29 #
30 def lossFun(inputs, targets, hprev):
31     """
32     Inputs, targets are both list of integers.
33     hprev is old array of initial hidden state
34     returns the loss, gradients on model parameters, and last hidden state
35     """
36     x, h, y, px, ph = (0, 0, 0, 0, 0)
37     hs1 = []
38     for t in xrange(len(inputs)):
39         x = np.zeros((vocab_size, 1)) # encode 1-of-k representation
40         x[inputs[t]] = 1
41         h = np.dot(Wxh, x) + np.dot(Whh, h) + bh # hidden state
42         y = np.dot(Why, h) + by # unnormalized log probability for next char
43         ph = np.exp(y) / np.sum(np.exp(y)) # probabilities for next char
44         loss += -np.log(ph[targets[t]]) # softmax (cross-entropy loss)
45         dx = np.zeros_like(x) # backprop into x
46         dh = np.zeros_like(ph) # backprop into h
47         dy = np.zeros_like(by) # backprop into by
48         dby = np.zeros_like(by)
49         dloss = np.zeros_like(by)
50         for t2 in reversed(xrange(len(inputs))):
51             dx = np.copy(dx)[t2]
52             dy[targets[t2]] -= 1 # backprop into y
53             dby += np.multiply(dy, ph1)
54             dy *= dy
55             dh = np.dot(Why.T, dy) + dh # backprop into h
56             dhw = (1 - h * h) * dh # backprop through tanh nonlinearity
57             dhw *= dhw
58             dWxh += np.dot(dhw, dx.T)
59             dWhh += np.dot(dhw, dh.T)
60             dby += np.dot(dhw, dby)
61             dloss += np.dot(dhw, dloss)
62             for param in [dWxh, dWhh, dby, dloss]:
63                 np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
64             return loss, dx, dh, dby, dloss, ph, y, px[inputs[t]-1]
65     return None
66     """
67     sample a sequence of integers from the model
68     h is memory state, seed_ix is seed letter for first time step
69     """
70     x = np.zeros((vocab_size, 1))
71     h = hprev
72     ixes = []
73     for t in xrange(n):
74         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
75         y = np.dot(Why, h) + by
76         p = np.exp(y) / np.sum(np.exp(y))
77         ix = np.random.choice(range(vocab_size), p=p.ravel())
78         x = np.zeros((vocab_size, 1))
79         x[ix] = 1
80         ixes.append(ix)
81     return ixes
82
83 n = 500
84
85 h0, ph0, m0h, m0y = np.zeros_like(Wxh), np.zeros_like(Whh),
86 np.zeros_like(Why), np.zeros_like(by) # memory variables for unrolled
87 smoth_loss = np.log(1.0/vocab_size)*seq_length + loss at iteration 0
88 while True:
89     # generate points (ixes) unrolled from left to right in steps seq_length long
90     if prev_length == len(data) or n == 0:
91         hprev = np.zeros((hidden_size, 1)) # reset RNN memory
92         g = 0 # g is from start of data
93         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
94         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
95
96     # sample from the model now and then
97     if n < 200 or n % 200 == 0:
98         sample_ix = sample(hprev, inputs[0], 200)
99         txt = '\n'.join(ix_to_char[i] for i in sample_ix)
100        print('...'.join(ix_to_char[i] for i in sample_ix))
101
102    # forward one length characters through the net and fetch gradients
103    loss, dx, dh, dby, db, dy, pnew = lossFun(inputs, targets, hprev)
104    smooth_loss = smooth_loss * 0.999 + loss * 0.001
105    if n < 200 == 0: print('iter %d, loss: %f' % (n, smooth_loss)) # print progress
106
107    # perform parameter update with Adagrad
108    for param, param_s, mem in zip([Wxh, Whh, Why, by],
109                                  [dWxh, dWhh, dWhy, dby],
110                                  [mem0h, mem0y, mem0y, mem0y]):
111        mem += np.multiply(param_s, param_s)
112        param -= learning_rate * param / np.sqrt(mem + 1e-8) # integral update
113
114    p += seq_length + 1 # move data pointer
115    n += 1 # iteration counter
```



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





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-iafhatawiaoahrdemot 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, ammerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

train more

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

train more

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



PANDARUS:

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

Second Senator:

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

DUKE VINCENTIO:

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

Second Lord:

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

Clown:

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

VIOLA:

I'll drink it.

VIOLA:

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

KING LEAR:

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



open source textbook on algebraic geometry

The Stacks Project

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

Browse chapters

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

Parts

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

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

Latex source





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

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

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

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fppf} and $U \rightarrow U$ is the fibre category of S in U in Section, T7 and the fact that any U affine, see Morphisms, Lemma T7. 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 $\mathcal{O}_{X,S}$ is a scheme where $x, x', x'' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}_{X',x'}$ is separated. By Algebra, Lemma T7 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 X' , and 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

$$\widehat{\mathcal{M}}^* = T^* \otimes_{\text{Spec}(R)} \mathcal{O}_{S,S} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

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

and

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

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 T7. It may replace S by $X_{\text{spaces},\text{étale}}$ which gives an open subspace of X and T equal to S_{Zar} , see Descent, Lemma T7. Namely, by Lemma T7 we see that R is geometrically regular over S .

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

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

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

When in this case of to show that $\mathcal{Q} \rightarrow \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition T7 (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 retrocomposes of this implies that $\mathcal{F}_{i_0} = \mathcal{F}_{i_0} = \mathcal{F}_{X,S}$.

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $I = J_1 \subset T_n$. Since $T^n \subset T^n$ are nonzero over $i_0 \leq p$ is a subset of $J_{n,0} \circ \tilde{A}_2$ works.

Lemma 0.3. In Situation T7. Hence we may assume $q' = 0$.

Proof. We will use the property we see that p is the next functor (T7). On the other hand, by Lemma T7 we see that

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

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



Proof. Omitted. \square

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

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

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

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

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

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 $\mathcal{U} \subset X$ 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

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

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. \square

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

$$\begin{array}{ccc}
 S & \longrightarrow & \\
 \downarrow & & \\
 \xi & \longrightarrow & \mathcal{O}_X \\
 \downarrow \text{gr}_x & & \uparrow \\
 & & \mathcal{O}_X \\
 & & \downarrow \text{gr}_x \\
 & & \mathcal{O}_X \\
 \downarrow \text{id} \alpha' & \longrightarrow & \downarrow \text{id} \alpha \\
 & & \downarrow \text{id} \alpha \\
 & & \mathcal{O}_X \\
 \text{Spec}(K_0) & \longrightarrow & \text{Mor}_{\mathcal{O}_X} \\
 & & \downarrow \text{id}(\mathcal{O}_{X_{\text{state}}}, \mathcal{G}) \\
 & & X
 \end{array}$$

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_x . 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 . \square

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 C . The functor \mathcal{F} is a field

$$\mathcal{O}_{X,S} \rightarrow \mathcal{F}_T \dashv (\mathcal{O}_{X_{\text{state}}}) \rightarrow \mathcal{O}_{X,T}^* \mathcal{O}_{X,S}(\mathcal{O}_{X,T}^*)$$

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

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

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

If \mathcal{F} is a finite direct sum \mathcal{O}_{X_1} is a closed immersion, see Lemma ??.
This is a sequence of \mathcal{F} is a similar morphism.





TECHNISCHE
UNIVERSITÄT
DARMSTADT



This repository Search

Explore Gist Blog Help

karpathy + ⌂ ⌂ ⌂ ⌂



torvalds / linux

Watch 3,711

Star 23,054

Fork 9,141

Linux kernel source tree

520,037 commits

1 branch

420 releases

5,039 contributors



branch: master + linux / +

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

torvalds authored 9 hours ago

latest commit 4b1786927d

Documentation

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

6 days ago

arch

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

a day ago

block

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

9 days ago

crypto

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

10 days ago

drivers

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

9 hours ago

firmware

firmware/hex2hw.c: restore missing default in switch statement

2 months ago

fs

vfs: read file_handle only once in handle_to_path

4 days ago

include

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

a day ago

init

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

a month ago

mm

mm: remove 'fix' from 'fix()' macro

in seconds now

Code

74
Pull requests

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 << 1))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 0x000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

Generated C code





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

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

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





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

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

#define SHAP_ALLOCATE(nr)      (e)
#define emulate_sigs() arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %%esp, %0, %1" : : "r" (0));  \
    if (__type & DO_READ)

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

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

```



Searching for interpretable cells

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

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



Searching for interpretable cells

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

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

quote detection cell



Searching for interpretable cells

Cell sensitive to position in line:

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

line length tracking cell



Searching for interpretable cells

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

if statement cell



Searching for interpretable cells

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

quote/comment cell



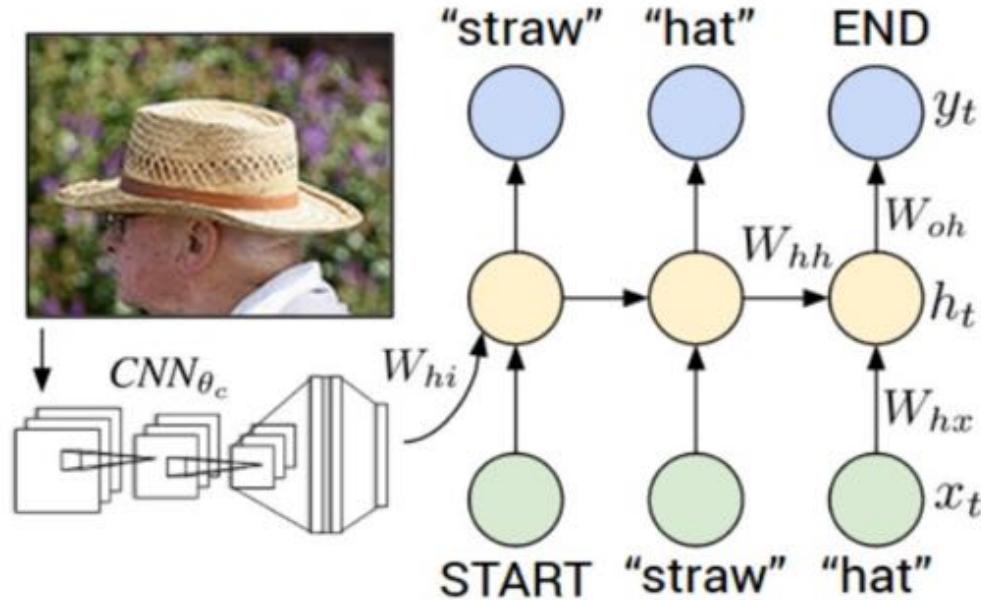
Searching for interpretable cells

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

code depth cell



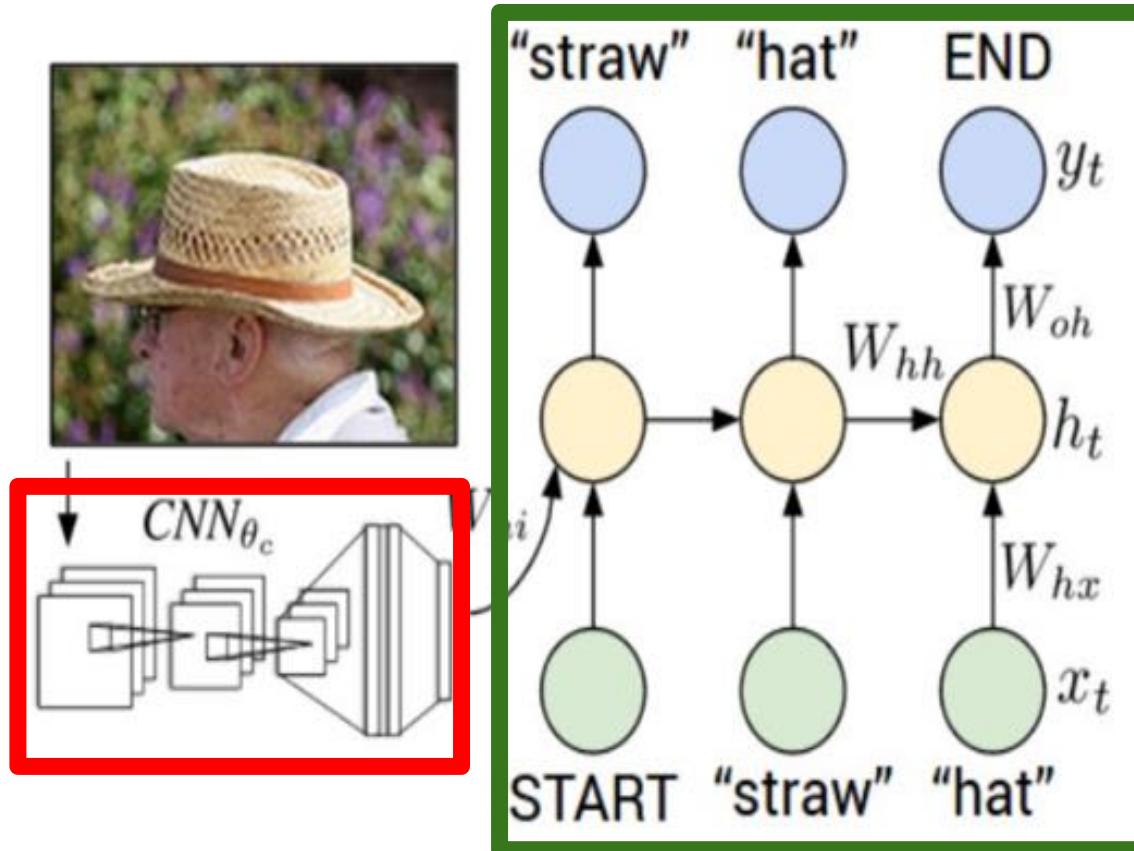
Image Captioning



- Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
- Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
- Show and Tell: A Neural Image Caption Generator, Vinyals et al.
- Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
- Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick



Image Captioning Recurrent Neural Network



Convolutional Neural Network



Image Captioning



test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

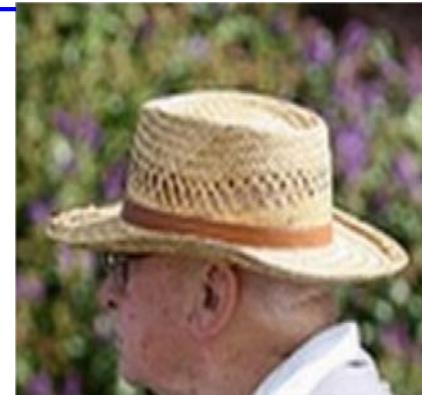


test image



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



<START>

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

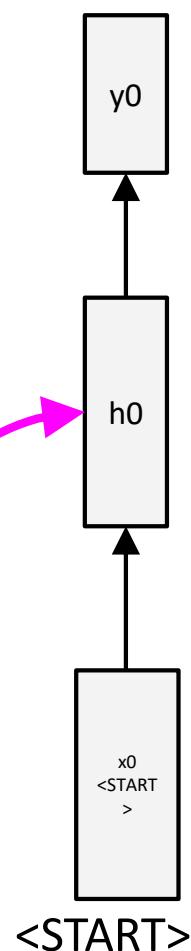
FC-4096

FC-4096

v



test image



before:

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

now:

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

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

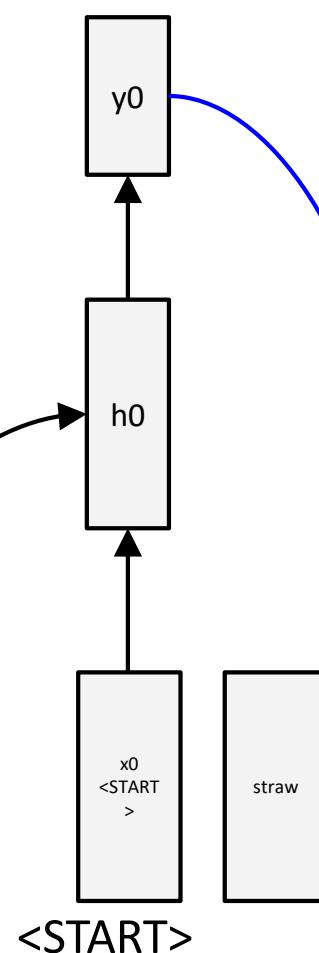
maxpool

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

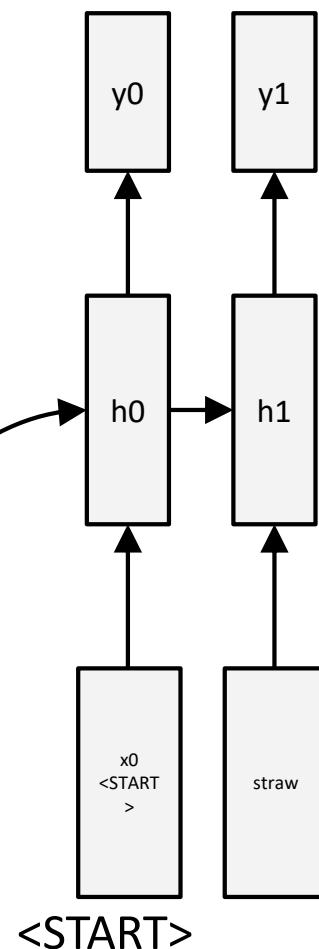
conv-512

conv-512

maxpool

FC-4096

FC-4096



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

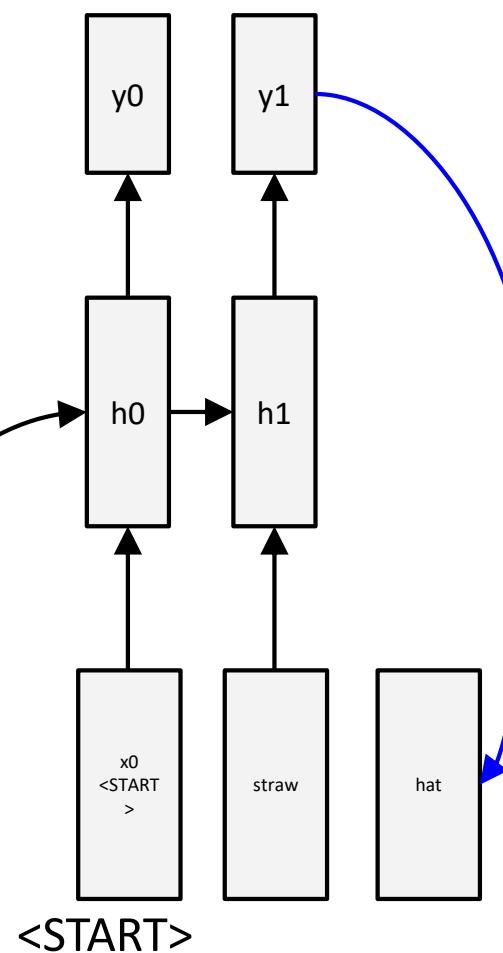
conv-512

conv-512

maxpool

FC-4096

FC-4096



sample!

image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

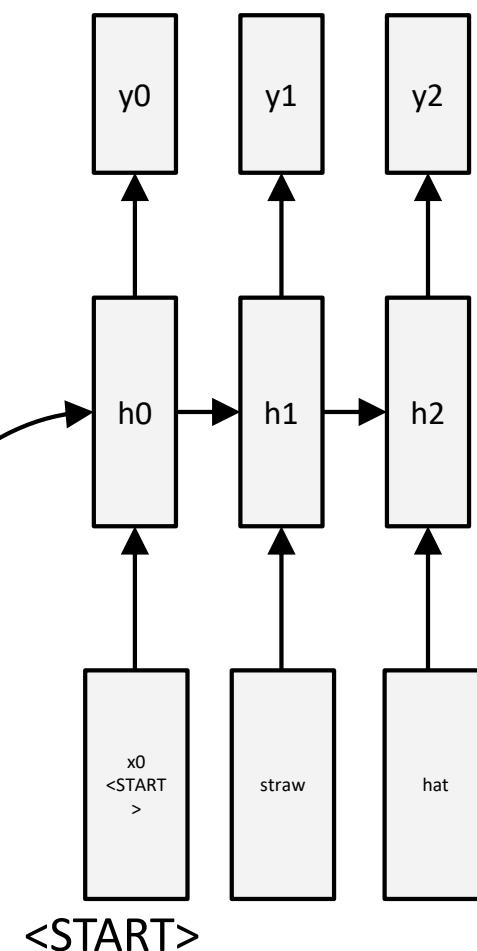
conv-512

conv-512

maxpool

FC-4096

FC-4096



image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

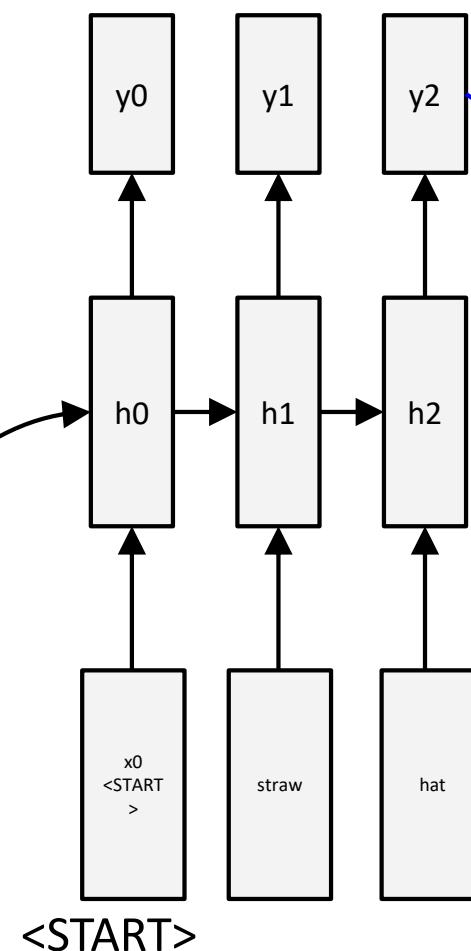
maxpool

FC-4096

FC-4096



test image



sample
<END> token
=> finish.

Image Sentence Datasets

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



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

currently:
~120K images
~5 sentences each





TECHNISCHE
UNIVERSITÄT
DARMSTADT



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."





"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



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



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

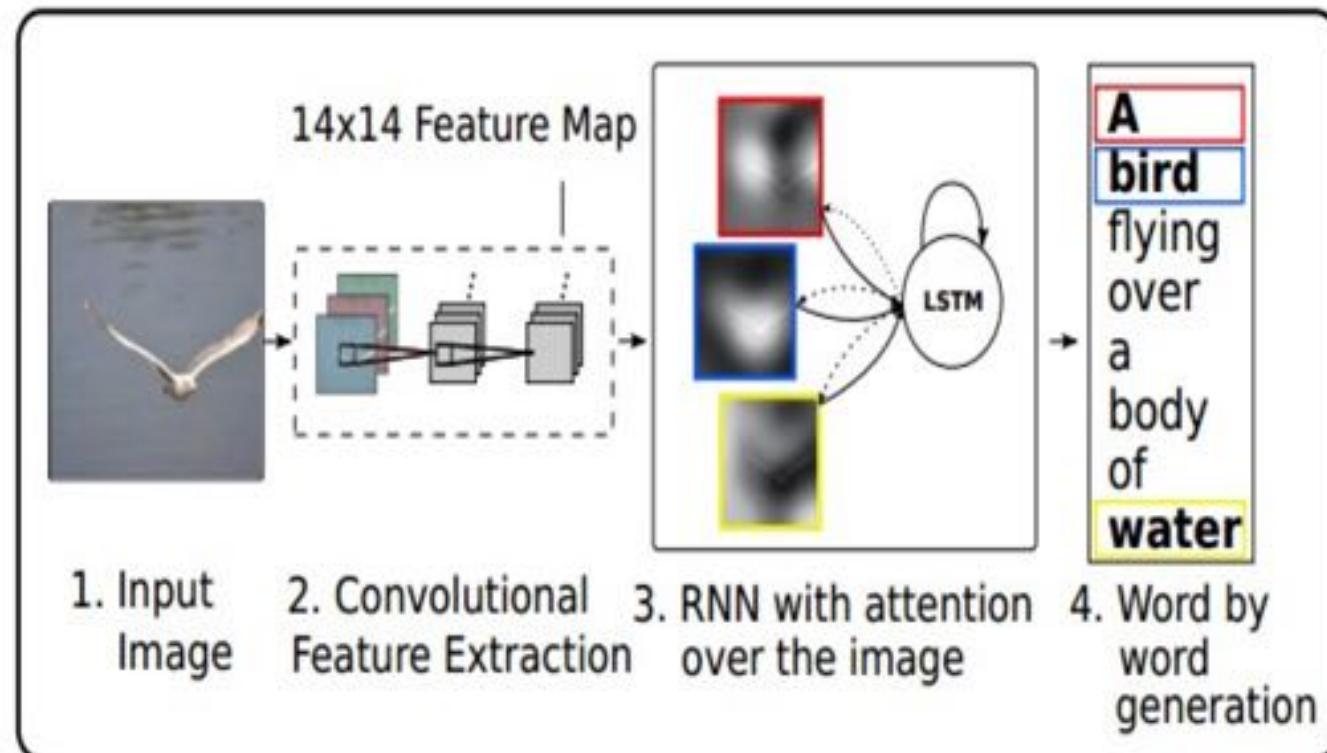


"a horse is standing in the middle of a road."



Preview of fancier architectures

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



Show Attend and Tell, Xu et al., 2015

LSTM: Long Short-Term Memory

Vanilla RNN:

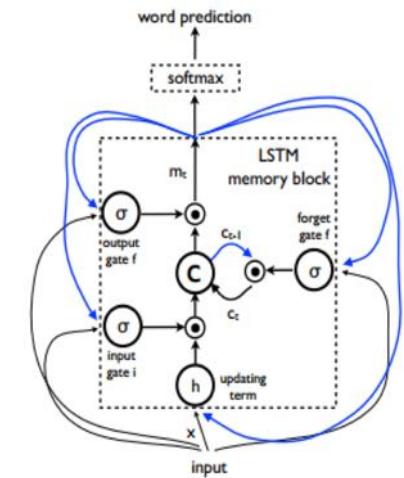
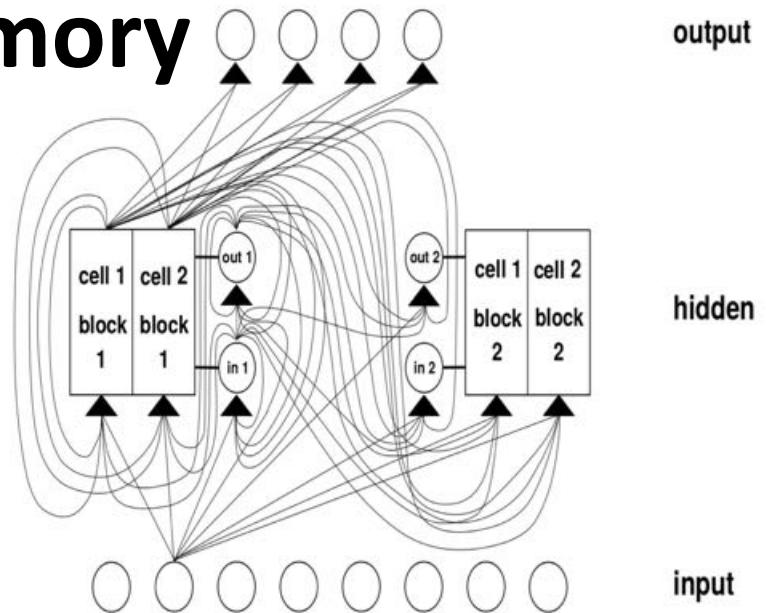
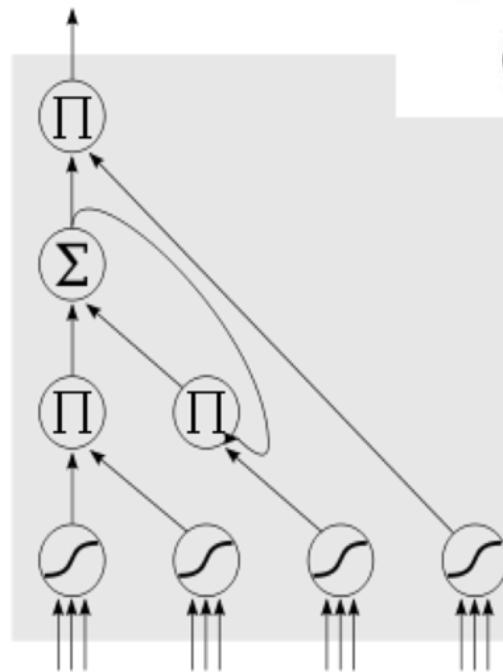
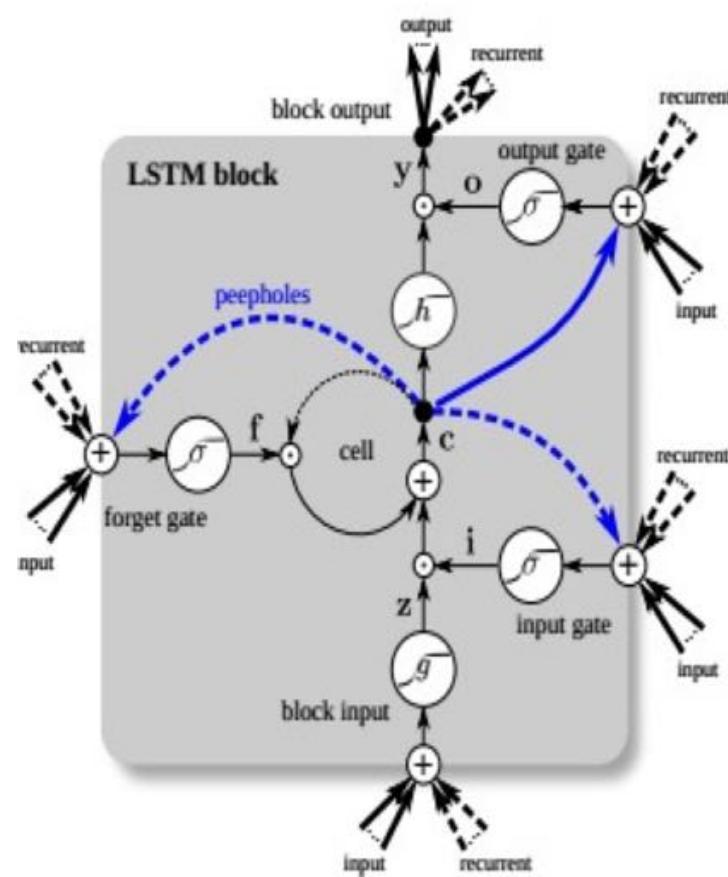
$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$h \in \mathbb{R}^n \quad W^l [n \times 2n]$$

LSTM:

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

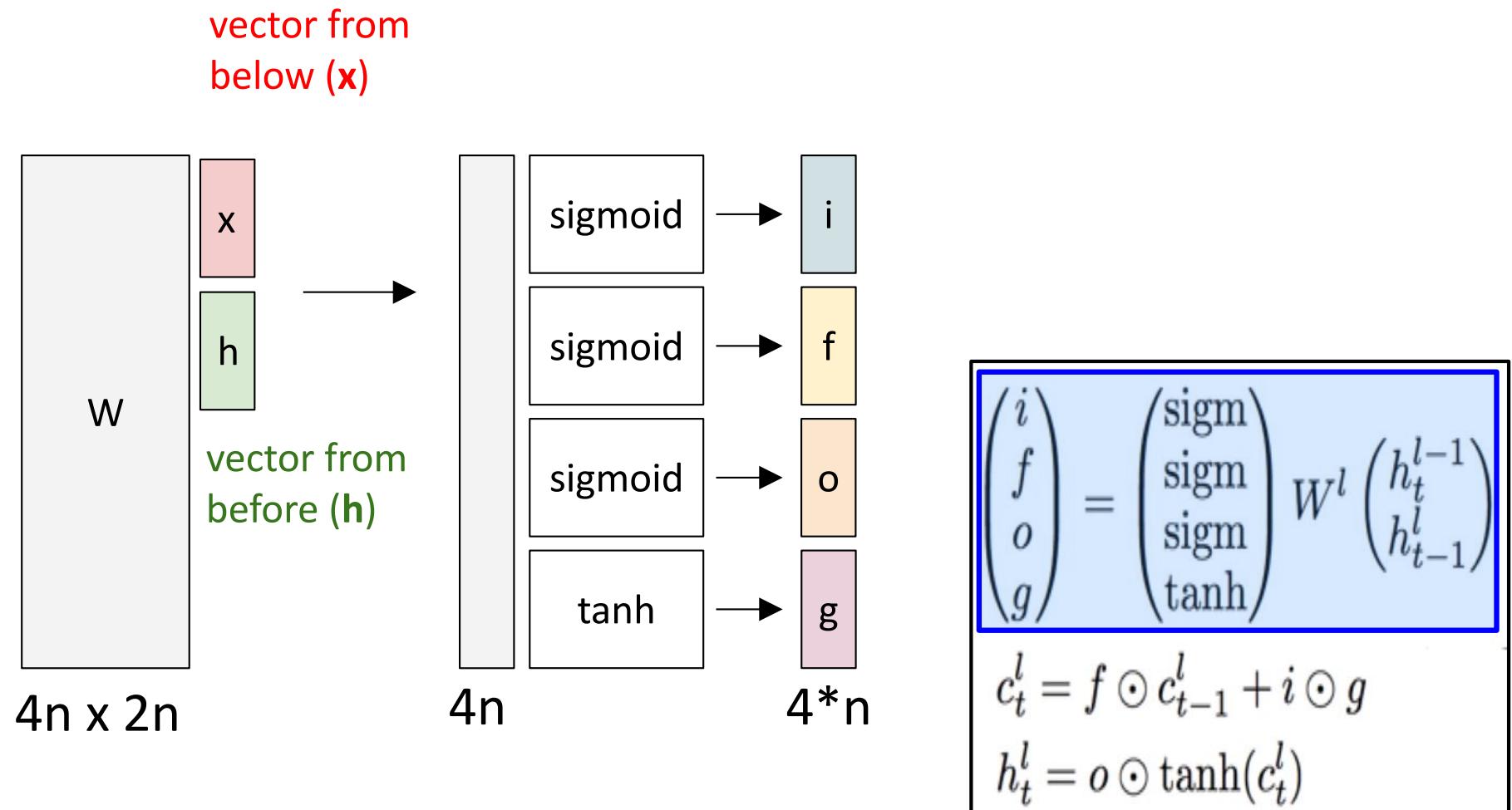


LSTM: Long Short-Term Memory



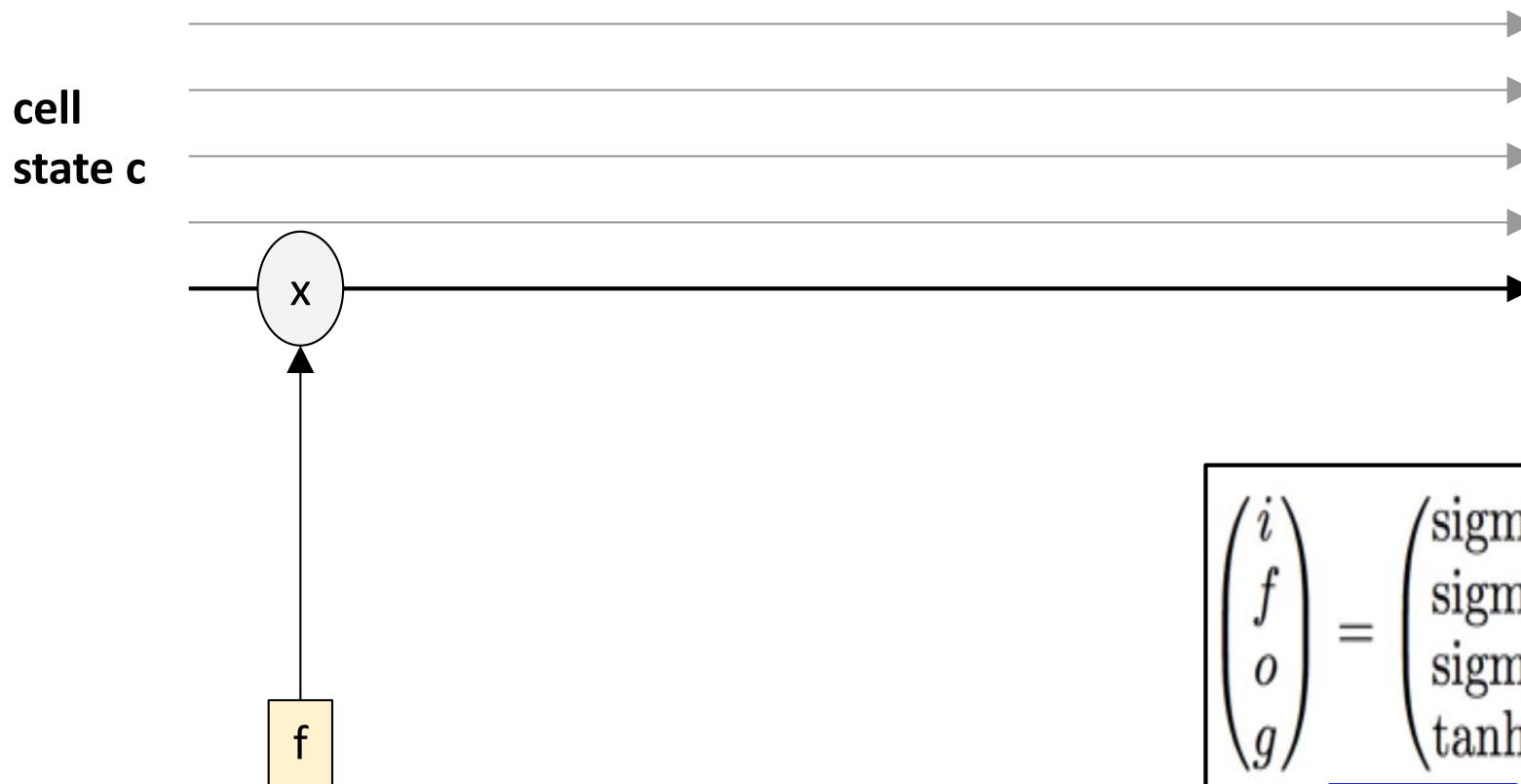
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



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

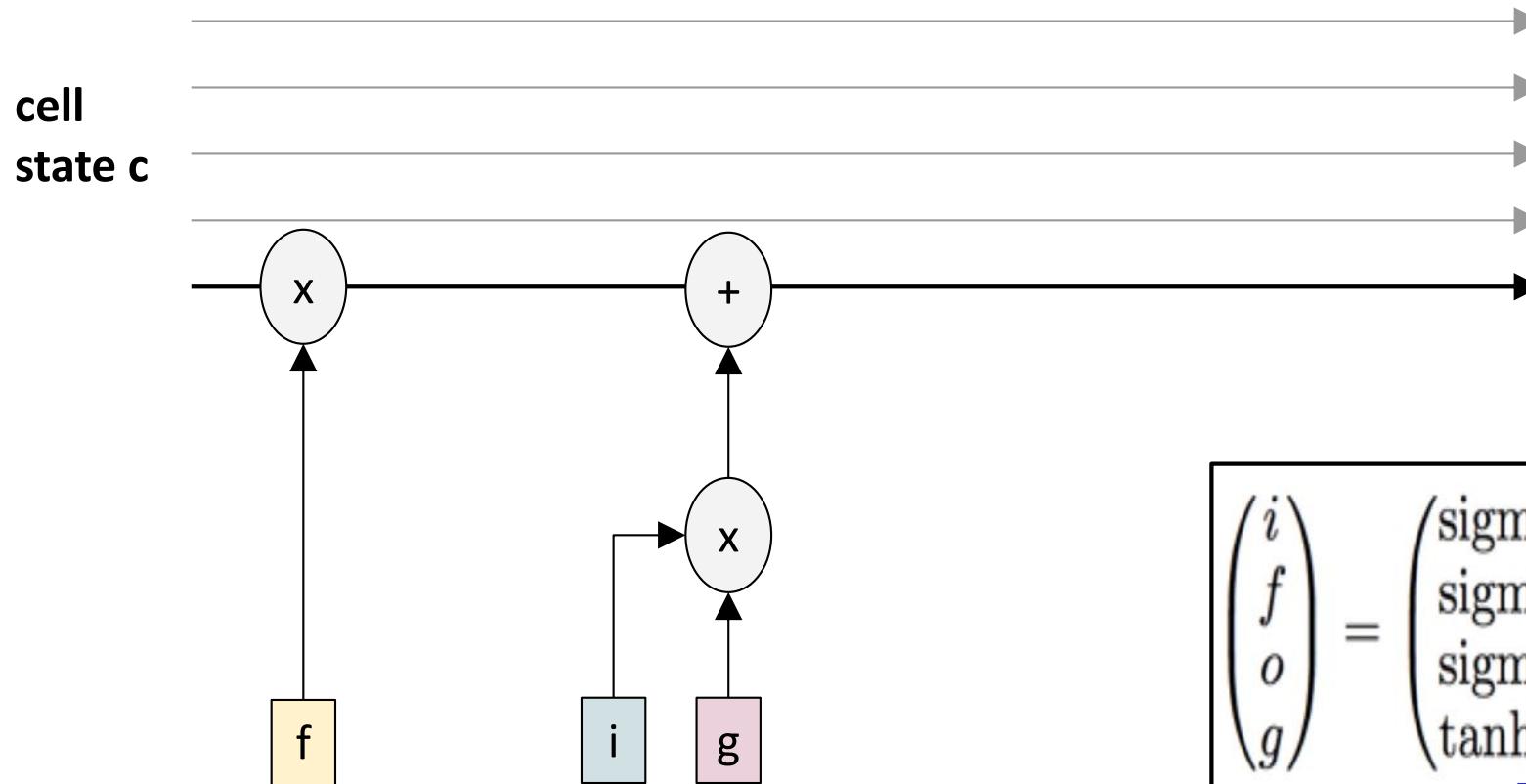
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

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



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



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

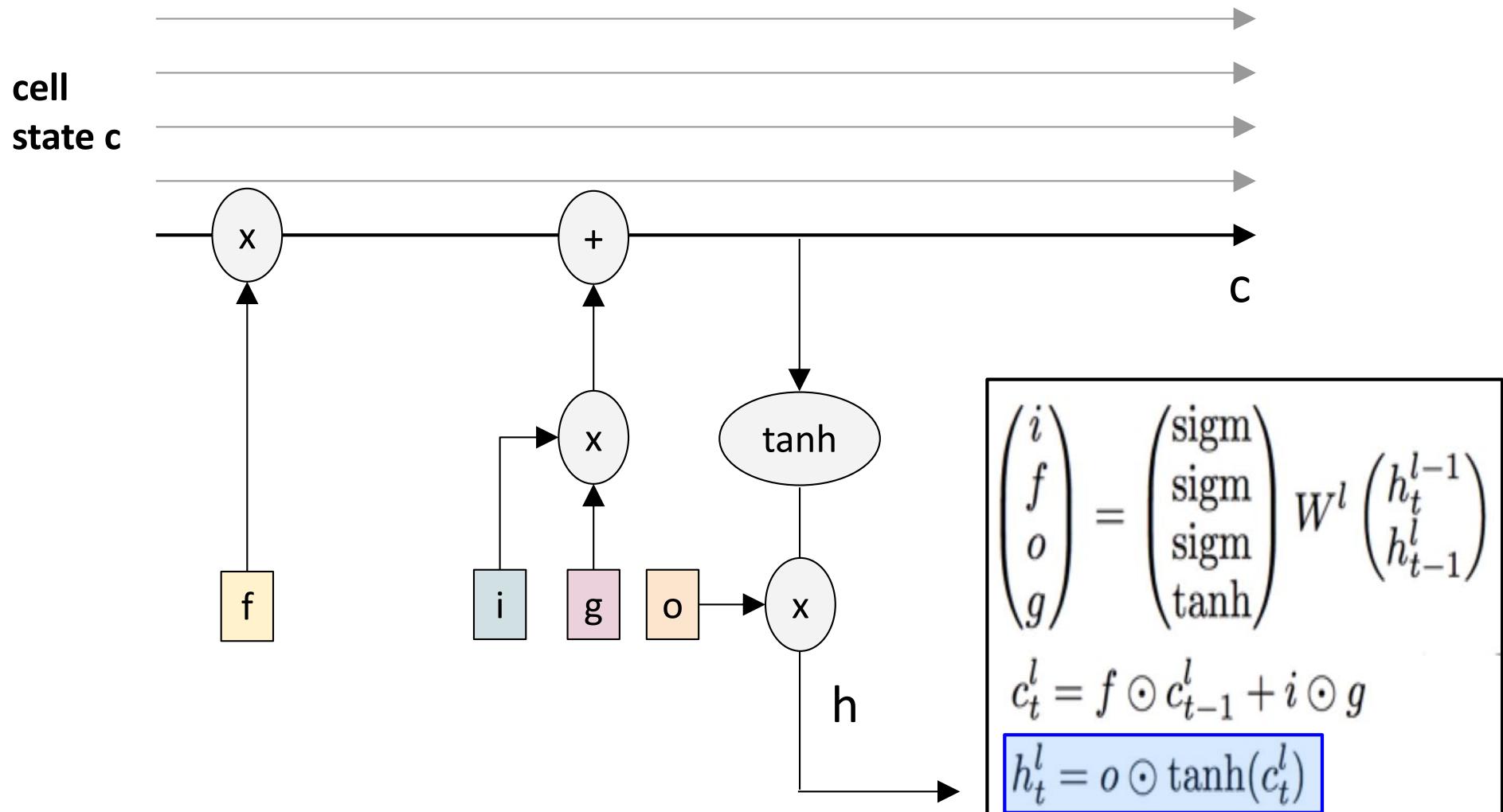
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

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



Long Short Term Memory (LSTM)

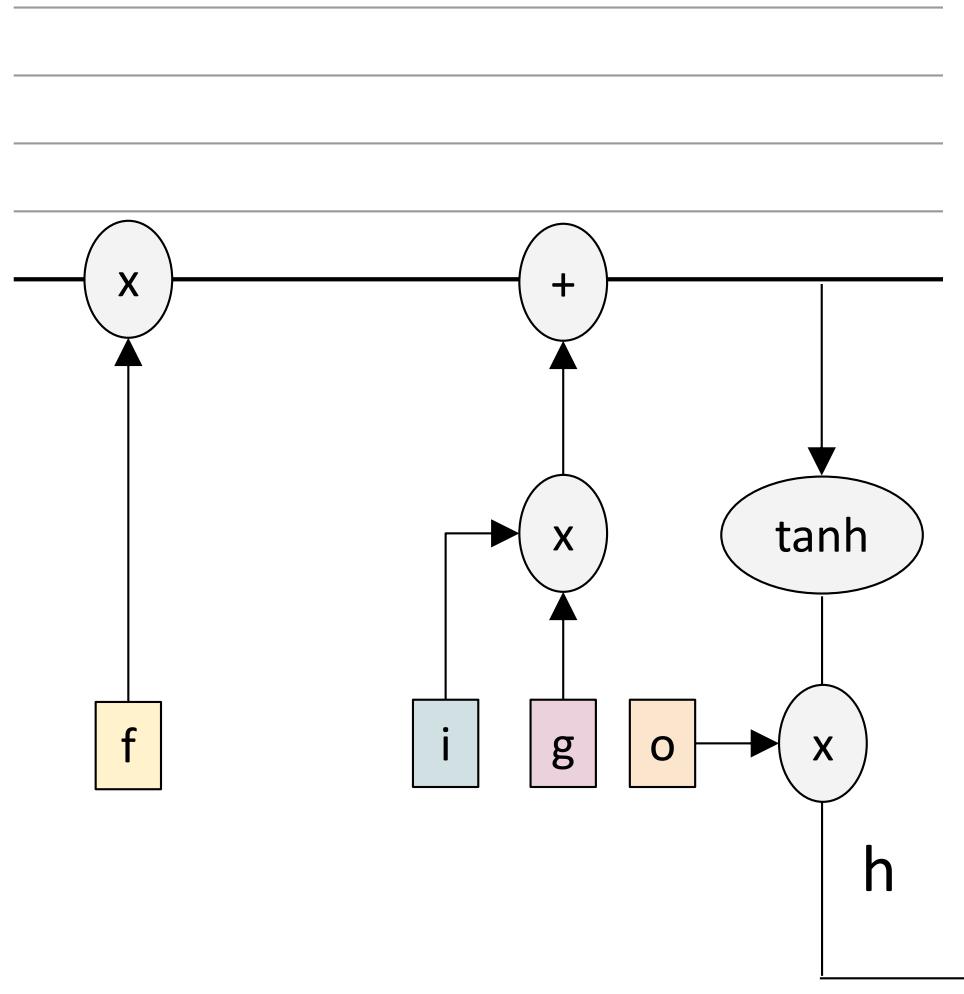
[Hochreiter et al., 1997]



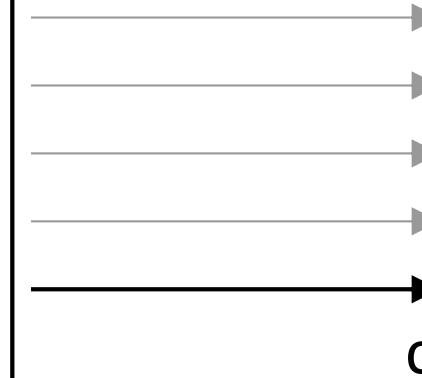
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

cell
state c



higher layer, or prediction



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

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

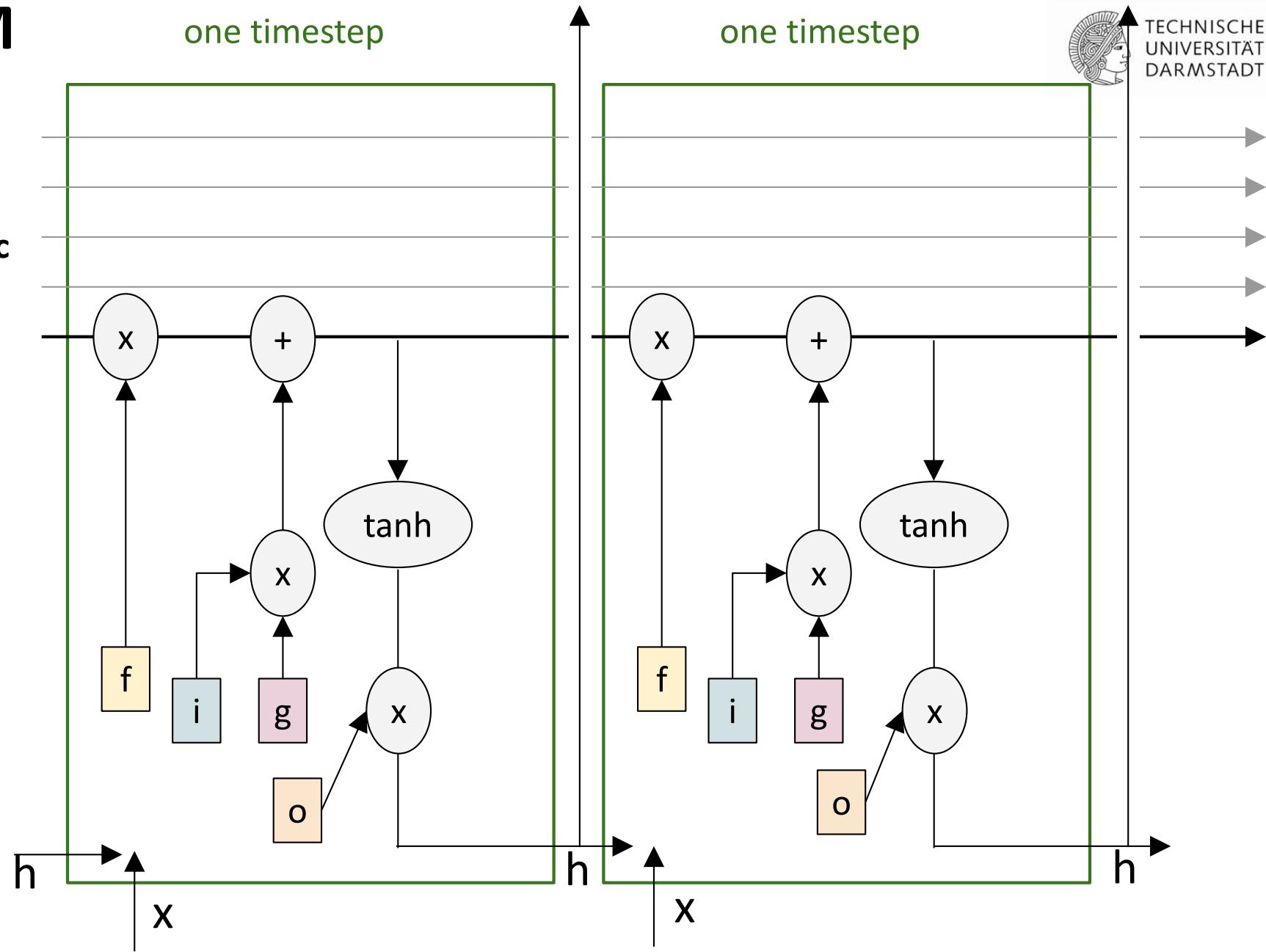
$$h_t^l = o \odot \tanh(c_t^l)$$



LSTM

one timestep

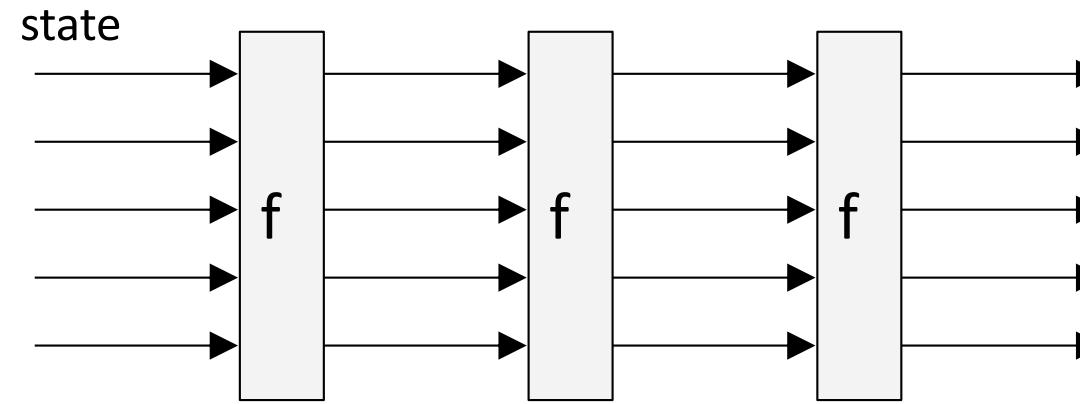
cell
state c



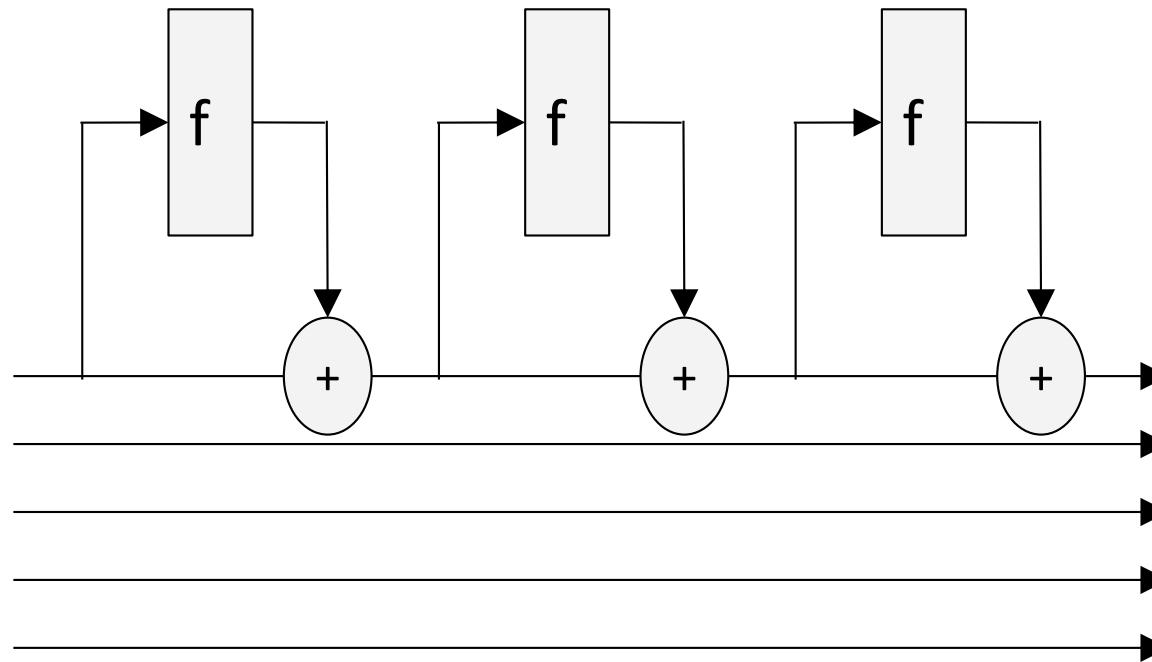
TECHNISCHE
UNIVERSITÄT
DARMSTADT



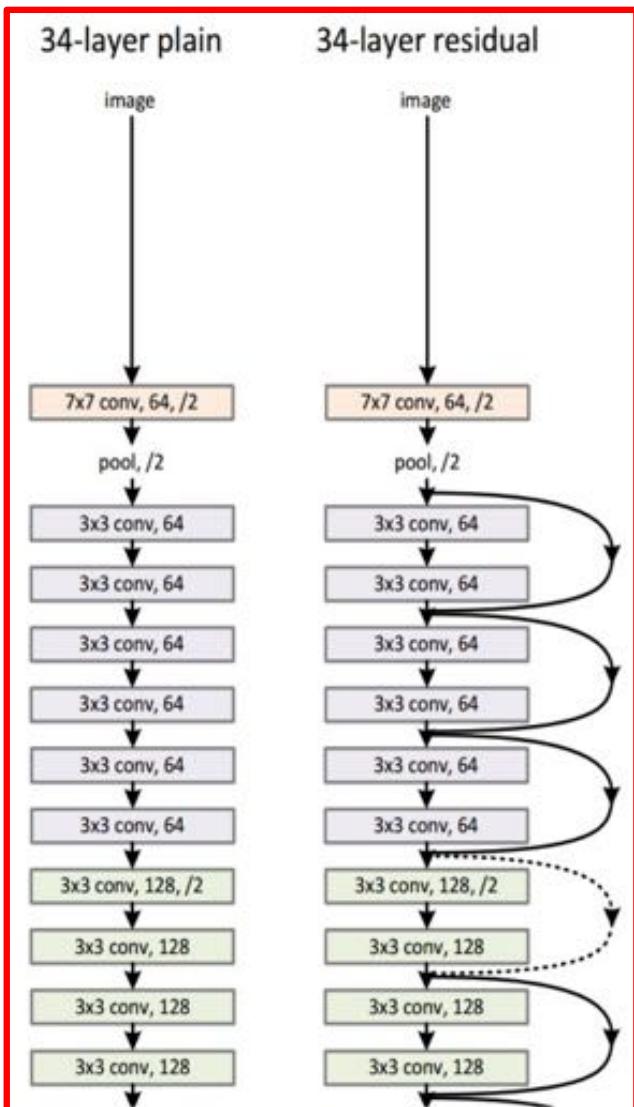
RNN



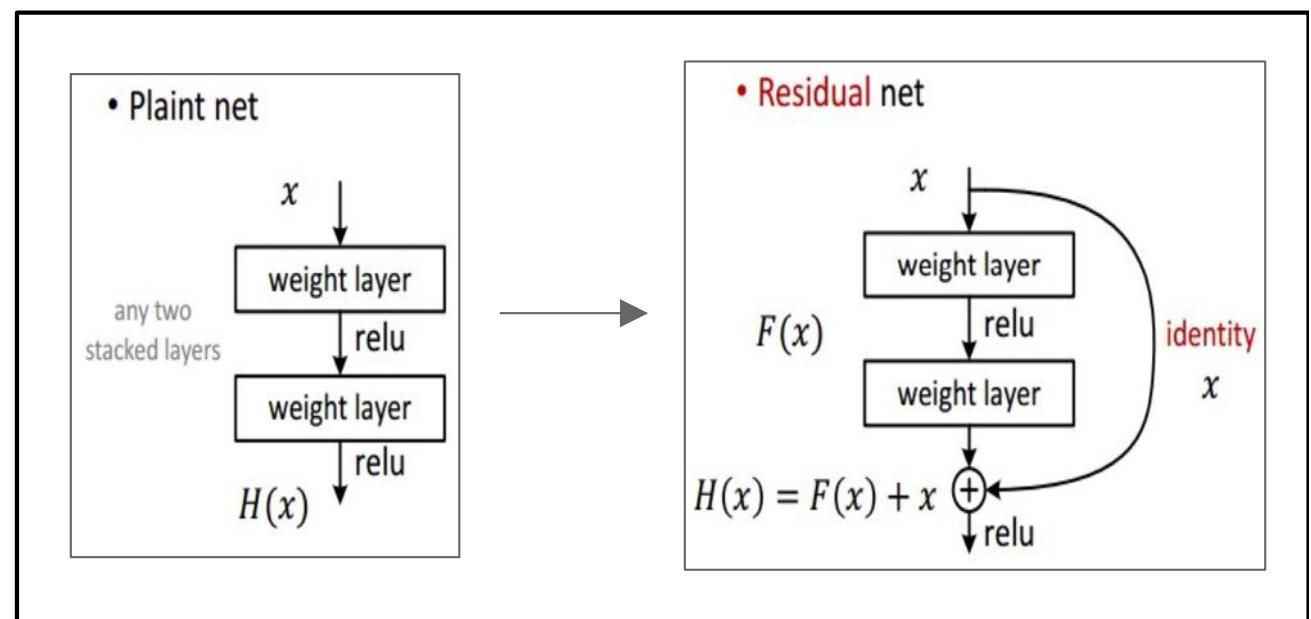
LSTM (ignoring forget gates)



Recall: “PlainNets” vs. ResNets



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



Understanding gradient flow dynamics



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

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

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

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



Understanding gradient flow dynamics



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

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

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

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

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



Understanding gradient flow dynamics



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

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

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

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

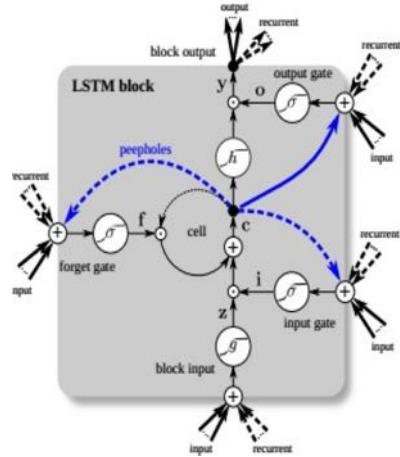
can control exploding with gradient clipping
can control vanishing with LSTM

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



LSTM variants and friends

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



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

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

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

MUT1:

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

MUT2:

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

MUT3:

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



What have you learnt?

- 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.

