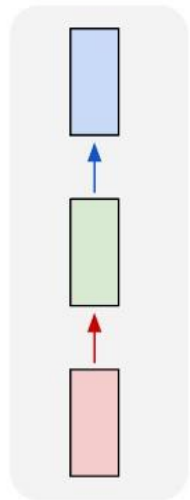


Lecture 16:

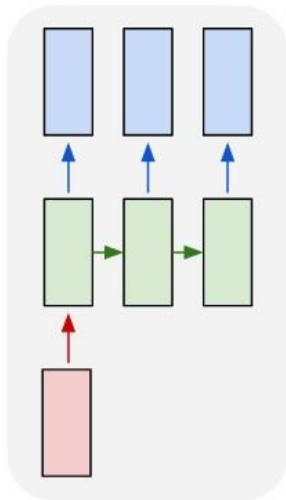
Recurrent Neural Networks

Recurrent Networks offer a lot of flexibility:

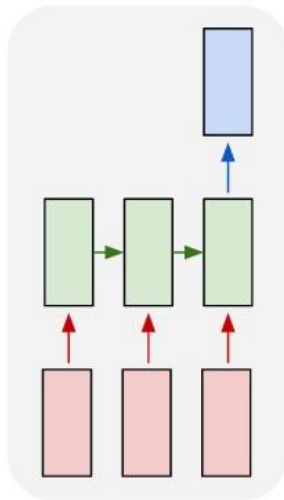
one to one



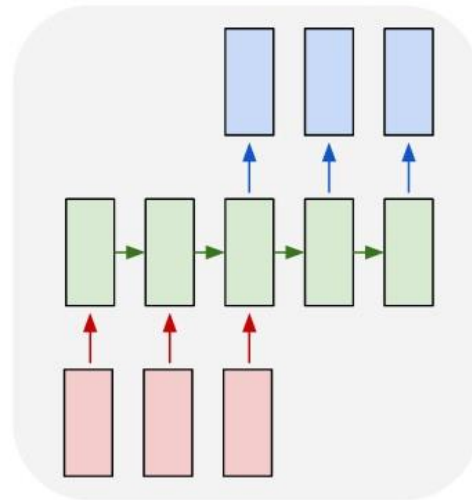
one to many



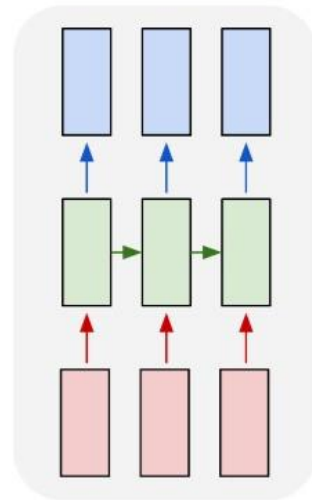
many to one



many to many



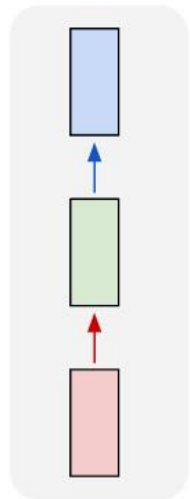
many to many



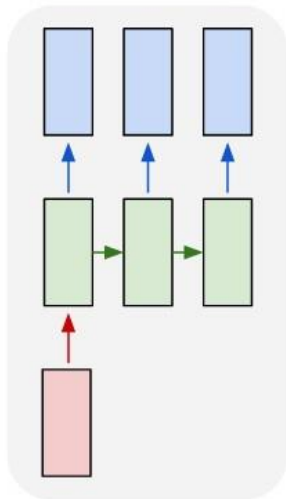
↙ **Vanilla Neural Networks**

Recurrent Networks offer a lot of flexibility:

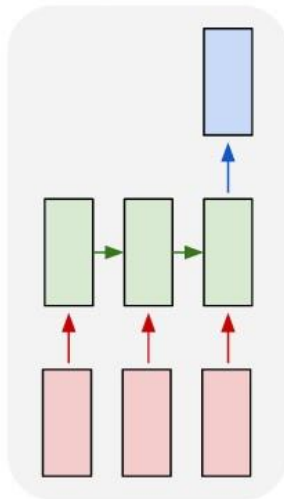
one to one



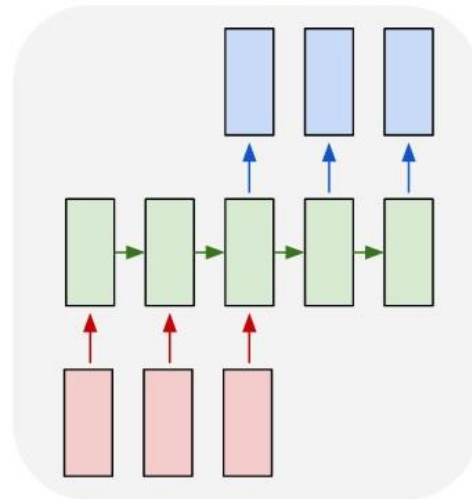
one to many



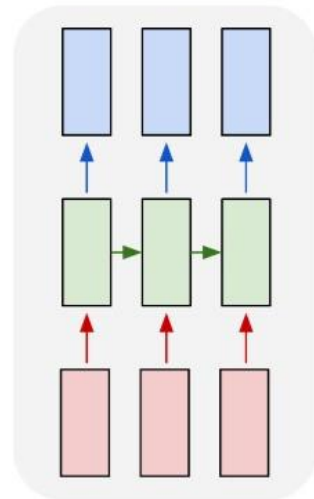
many to one



many to many



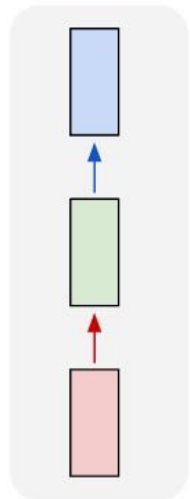
many to many



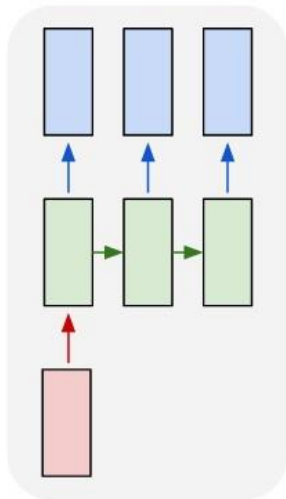
e.g. **Image Captioning**
image -> sequence of words

Recurrent Networks offer a lot of flexibility:

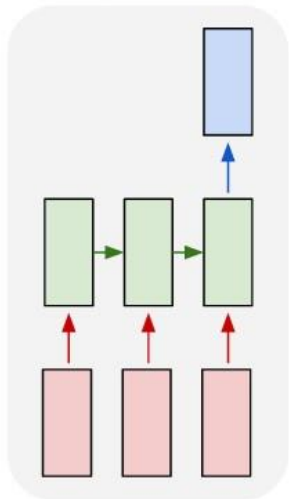
one to one



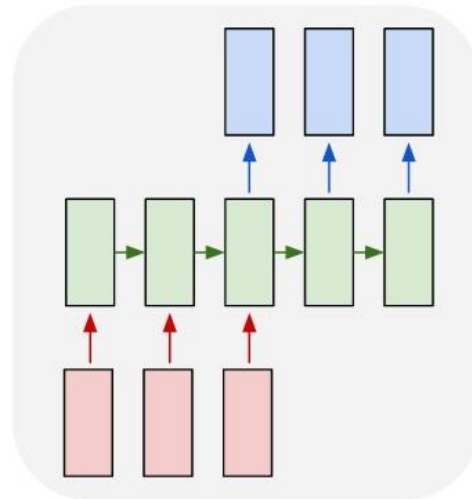
one to many



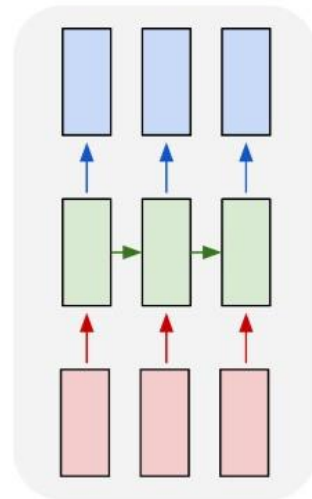
many to one



many to many



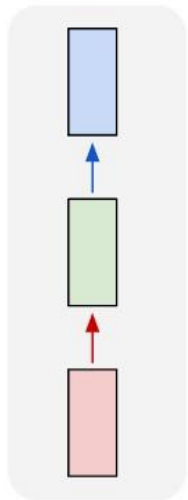
many to many



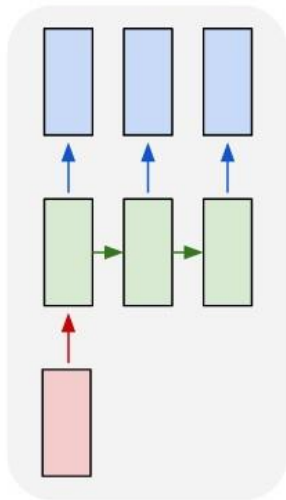
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

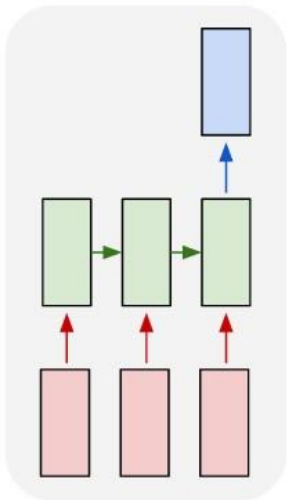
one to one



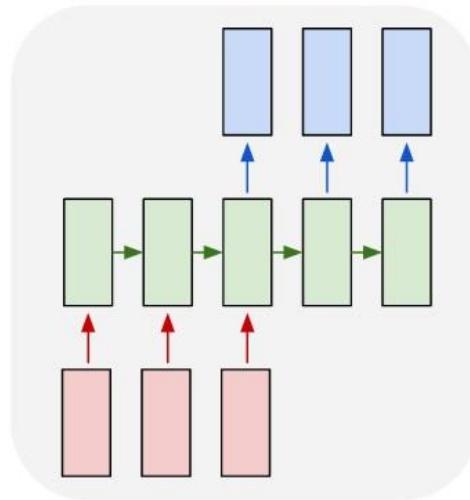
one to many



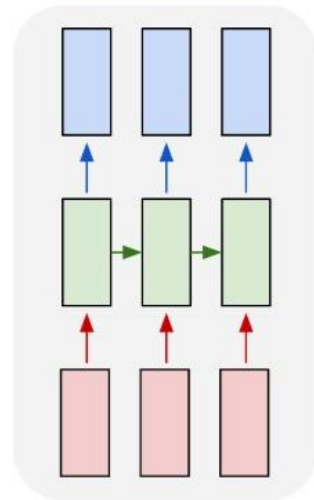
many to one



many to many



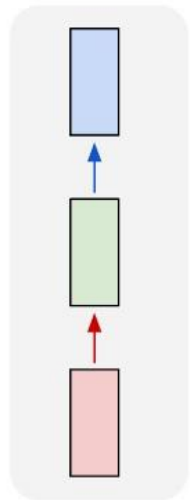
many to many



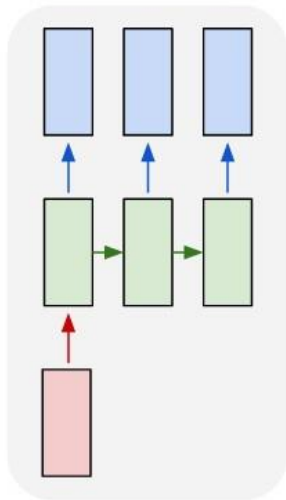
↖ e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Networks offer a lot of flexibility:

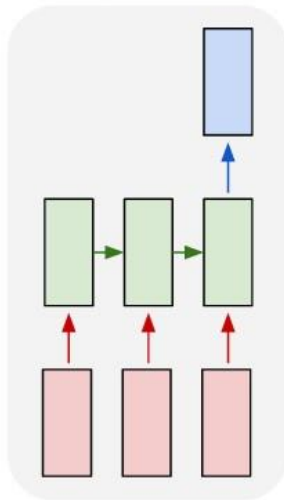
one to one



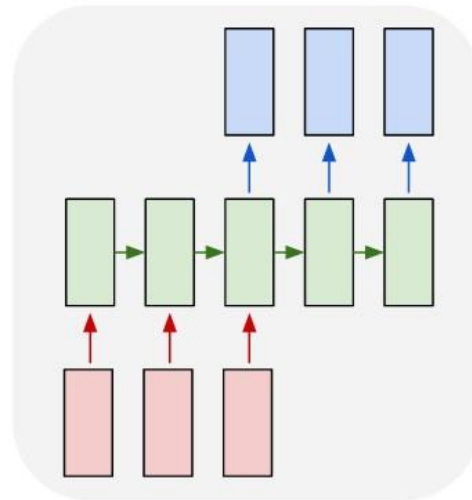
one to many



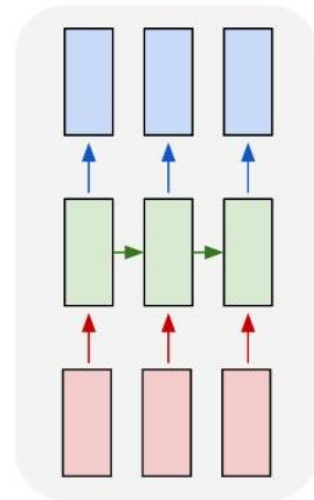
many to one



many to many



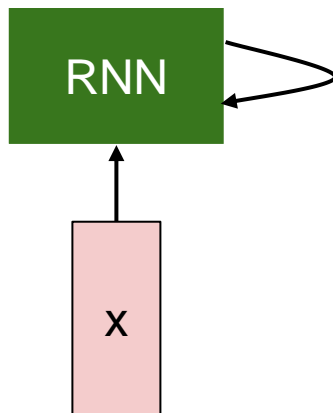
many to many



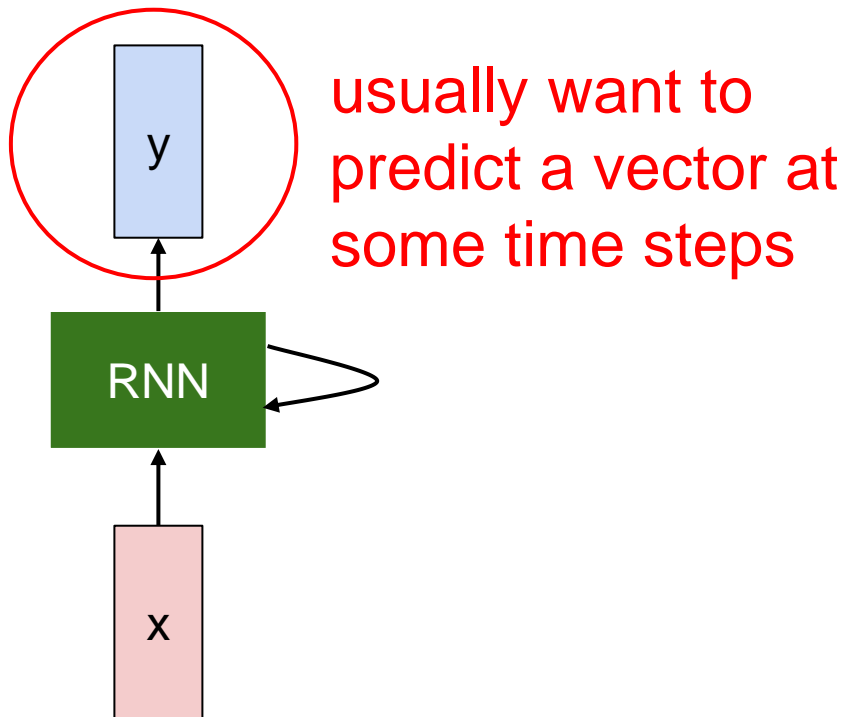
e.g. **Video classification on frame level**



Recurrent Neural Network



Recurrent Neural Network



Recurrent Neural Network

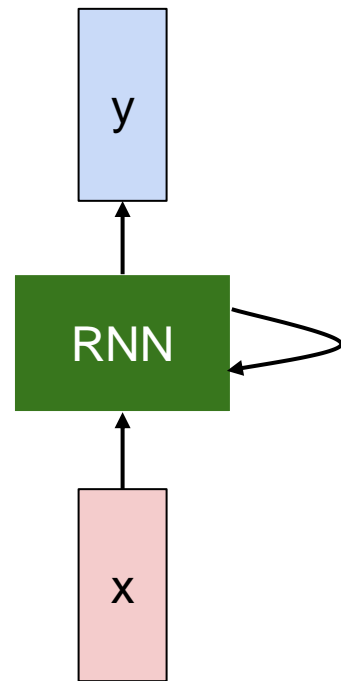
We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

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

new state / some function with parameters W

old state

input vector at some time step

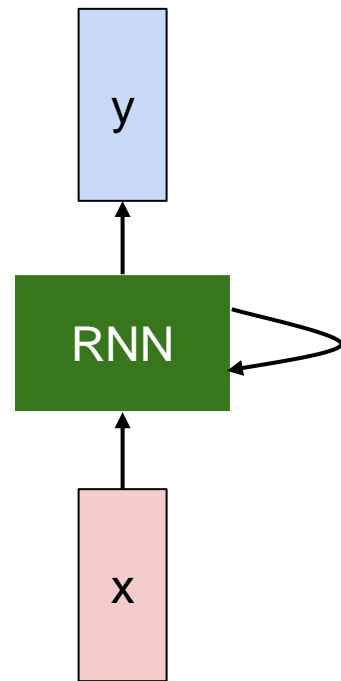


Recurrent Neural Network

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

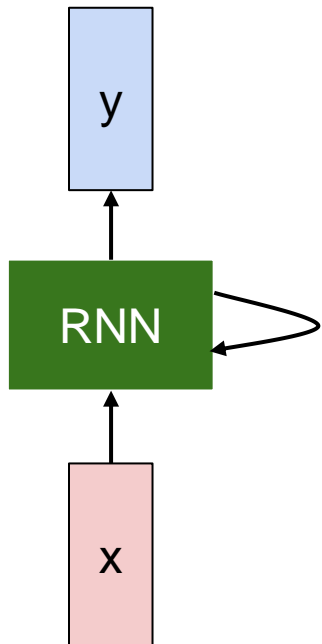
$$h_t = f_W(h_{t-1}, x_t)$$

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



(Vanilla) Recurrent Neural Network

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



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



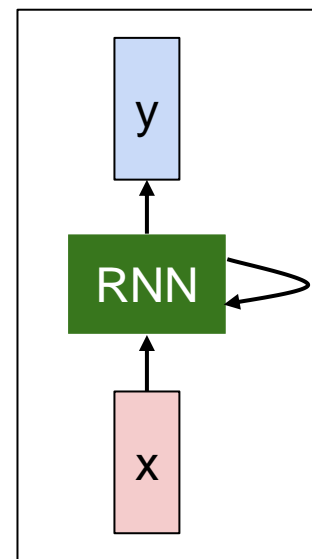
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Character-level language model example

Vocabulary:
[h,e,l,o]

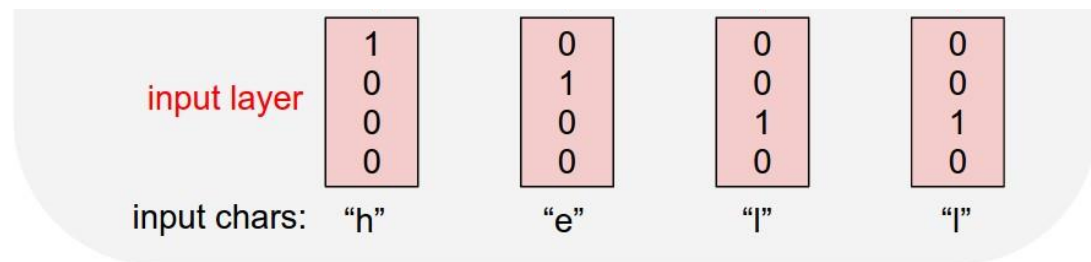
Example training sequence:
“**hello**”



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

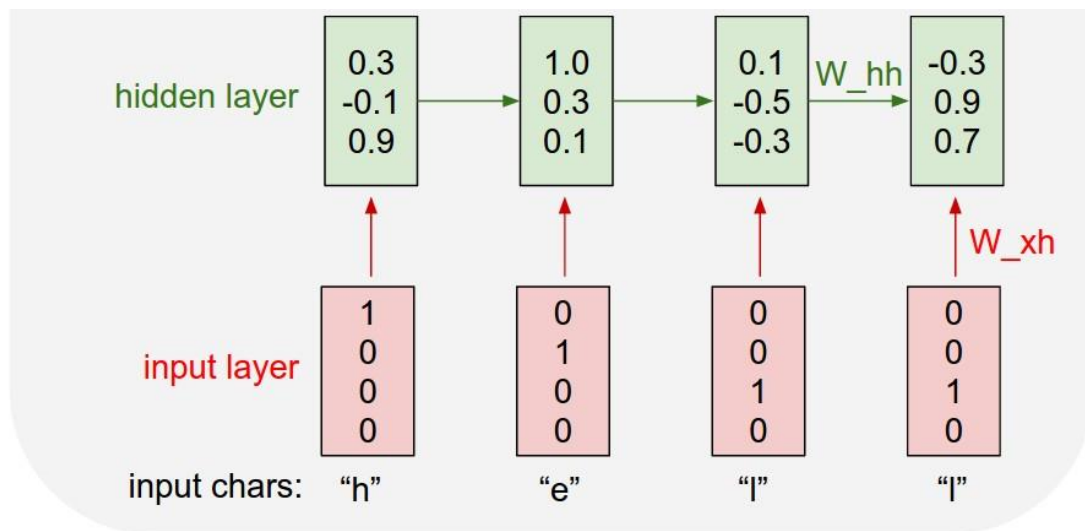


Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

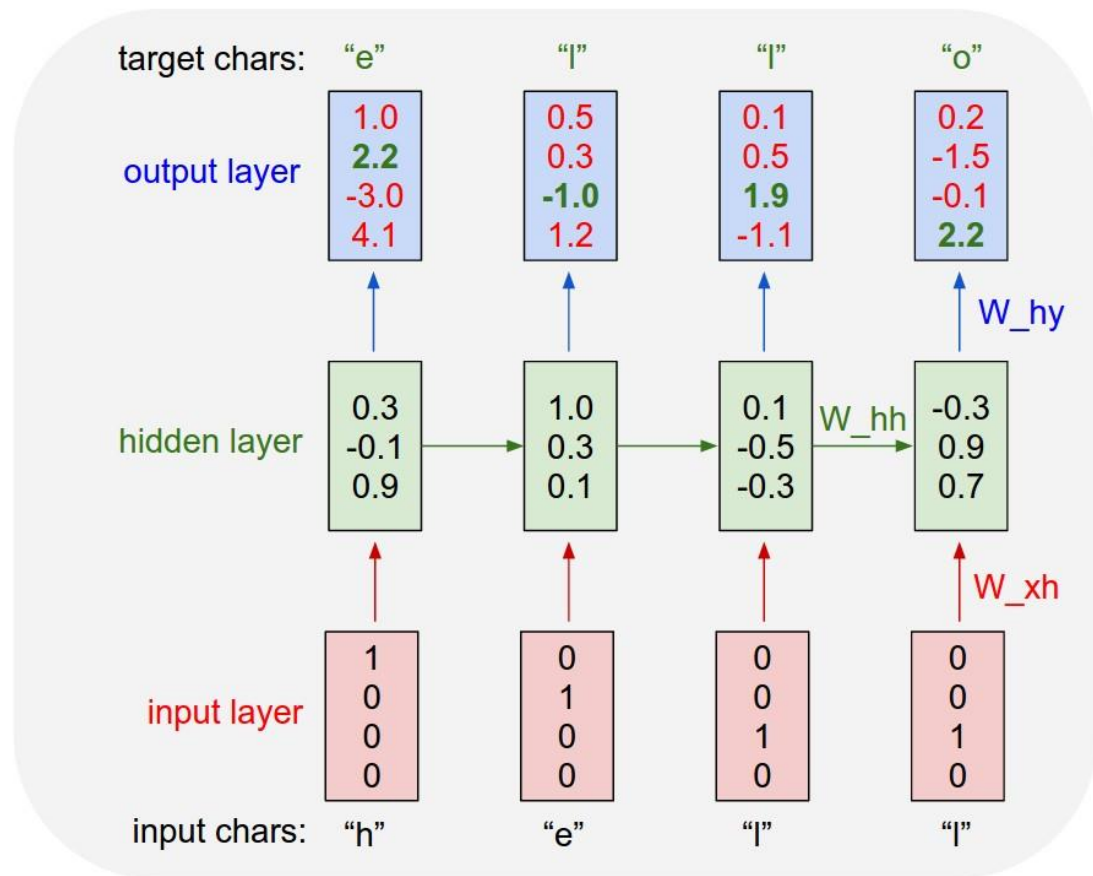
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



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

```
1 """
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4 """
5 import numpy as np
6
7 # data I/O
8 data = open('input.txt', 'r').read() # should be simple plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
12 char_to_ix = { ch:i for i,ch in enumerate(chars) }
13 ix_to_char = { i:ch for i,ch in enumerate(chars) }
14
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(wxh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44     # backward pass: compute gradients going backwards
45     dwxh, dwhh, dwhy = np.zeros_like(wxh), np.zeros_like(whh), np.zeros_like(why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnext = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49         dy = np.copy(ps[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dwhy += np.dot(dy, hs[t].T)
52         dby += dy
53         dh = np.dot(why.T, dy) + dhnext # backprop into h
54         dhraw = (1 - hs[t]**2) * hs[t]**2 * dh # backprop through tanh nonlinearity
55         dbh += dhraw
56         dxh = np.dot(dhraw, xs[t].T)
57         dwhh += np.dot(dhraw, hs[t-1].T)
58         dhnext = np.dot(whh.T, dhraw)
59     for dparam in [dwxh, dwhh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dwxh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```

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

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

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 data = open('input.txt', 'r').read() # should be simple plain text file
8 chars = list(set(data))
9 data_size, vocab_size = len(data), len(chars)
10 print('data has %d characters, %d unique.' % (data_size, vocab_size))
11 char_to_ix = { ch:i for i, ch in enumerate(chars) }
12 ix_to_char = { i:ch for i, ch in enumerate(chars) }
13
14 # Hyperparameters
15 hidden_size = 100 # size of hidden layer of neurons
16 seq_length = 25 # number of steps to unroll the RNN for
17 learning_rate = 1e-1
18
19 # Model parameters
20 wh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
21 wh_ = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
22 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
23 bh = np.zeros((hidden_size, 1)) # hidden bias
24 by = np.zeros((vocab_size, 1)) # output bias
```

```
25 def lossFun(inputs, targets, hprev):
26     """
27     inputs, targets are both list of integers.
28     hprev is Hx1 array of initial hidden state
29     returns the loss, gradients on model parameters, and last hidden state
30     """
31     xs, hs, ys, ps = {}, {}, {}, {}
32     hs[-1] = np.copy(hprev)
33     loss = 0
34     # forward pass
35     for t in xrange(len(inputs)):
36         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
37         xs[t][inputs[t]] = 1
38         hs[t] = np.tanh(np.dot(whx, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
39         ys[t] = np.dot(why, hs[t]) + by # unnormalized log probabilities for next chars
40         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
41     loss = -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
42     # backward pass: compute gradients going backwards
43     dhs = dhs = np.zeros_like(hs), np.zeros_like(whx), np.zeros_like(whh), np.zeros_like(why)
44     dhs[-1] = np.zeros_like(hs[-1])
45     dhs[-1] = np.zeros_like(hs[-1])
46     for t in reversed(xrange(len(inputs))):
47         dy = np.copy(ys[t])
48         dy[targets[t]] -= 1 # backprop into y
49         dhs[t] = np.dot(dy, why)
50         dhs[t] = dy
51         dh = np.dot(whh, T, dy) + dhs[t] # backprop into h
52         dhraw = (-1 + dh**2) * whx[t] # dh = backprop through tanh nonlinearity
53         dhs[t] = dhraw
54         dhs[t] = np.dot(dhraw, whx[t])
55         dhs[t] = np.dot(dhraw, whx[t])
56         dhs[t] = np.dot(whh, T, dhraw)
57         for param in [wh, wh_, why, bh, dhs]:
58             np.clip(param, -5, 5, out=param) # clip to mitigate exploding gradients
59     return loss, dhs, dwh, dwh_, dwhy, dbh, dby, hs[len(inputs)-1]
```

```
60 def sample(next_ix, h):
61     """
62     sample a sequence of integers from the model
63     h is memory state, next_ix is seed letter for first time step
64     """
65     x = np.zeros((vocab_size, 1))
66     ix_to_ix = h
67     ix = next_ix
68     for i in xrange(1):
69         x[ix] = 1
70         ix = np.argmax(np.dot(whx, x) + np.dot(whh, h) + bh)
71         h = np.tanh(np.dot(whh, h) + np.dot(whx, x) + bh)
72         ix = np.argmax(np.dot(why, h) + by)
73         p = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
74         ix = np.random.choice(vocab_size, p=p, random_state=None)
75         x[ix] = 1
76     return ix
77
78 # Run
79 p = 0.0
80 mwh, mwh_, mwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
81 mwh, mwh_, mwhy = np.zeros_like(wh), np.zeros_like(whh), np.zeros_like(why)
82 smooth_loss = -np.log(1.0/vocab_size)/seq_length # loss at iteration 0
83 while True:
84     # Prepare inputs (we're sampling from left to right in steps seq_length)
85     if prev_length is None or n == 0:
86         hprev = np.zeros((hidden_size, 1)) # reset rnn memory
87         p = 0 # p from start of data
88         inputs = [char_to_ix[ch] for ch in data[:prev_length]]
89         targets = [char_to_ix[ch] for ch in data[prev_length:]]
90     # Sample from the model and then
91     if n % 100 == 0:
92         sample_ix = sample(hprev, inputs[-1], 100)
93         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
94         print('....%s\n' % txt, )
95     # Forward seq_length characters through the net and fetch gradients
96     loss, dhs, dwh, dwh_, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
97     smooth_loss = smooth_loss * 0.999 + loss * 0.001
98     if n % 100 == 0: print('iter %d, loss %f' % (n, smooth_loss)) # print progress
99
100 # Perform parameter update with adagrad
101 for param, dparam, new in zip([wh, wh_, why, bh, dhs],
102                             [dwh, dwh_, dwhy, dbh, dby],
103                             [mwh, mwh_, mwhy, dbh, dby]):
104     param -= learning_rate * dparam / np.sqrt(new + 1e-8) # adagrad update
105
106 # seq_length = move data pointer
107 n += 1 # iteration counter
```

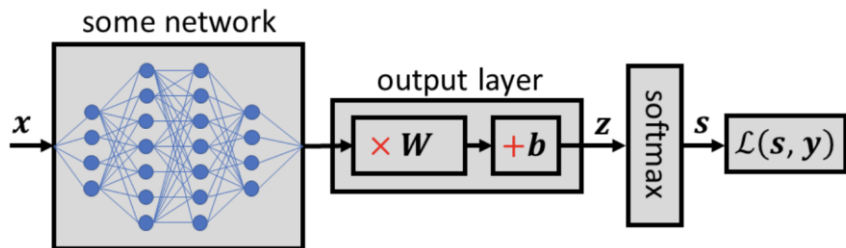
```
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)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

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

Softmax classifier

Derivative of Softmax and Categorical Cross-Entropy Loss

<https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>



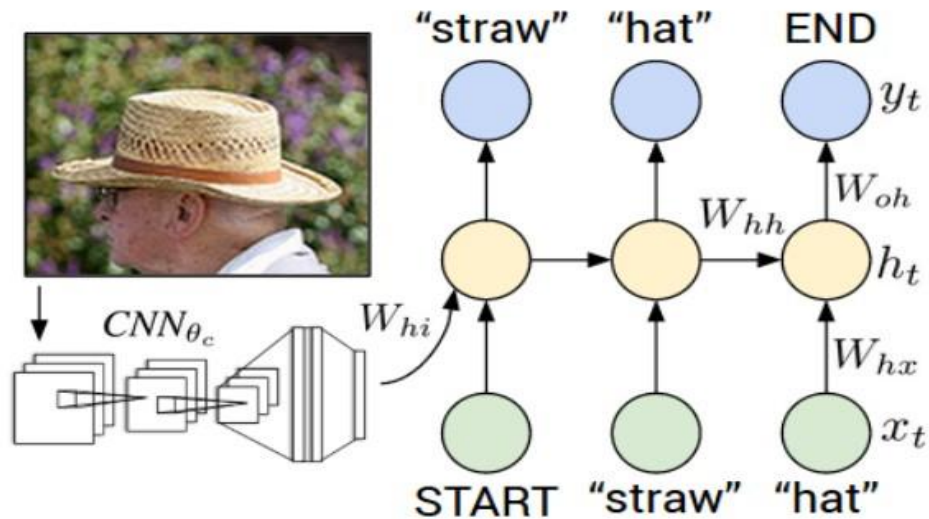
(Image by author)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \mathbf{s} - \mathbf{y}$$

In order to kick off the backpropagation process, as described in this [post](#), we have to calculate the derivative of the loss w.r.t to *weighted input* z of the output layer, see figure above:

$$\frac{\partial \mathcal{L}}{\partial z_j} = -\frac{\partial}{\partial z_j} \sum_{i=1}^c y_i \cdot \log(s_i) = -\sum_{i=1}^c y_i \cdot \frac{\partial}{\partial z_j} \log(s_i) = -\sum_{i=1}^c \frac{y_i}{s_i} \cdot \frac{\partial s_i}{\partial z_j}$$

Image Captioning



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

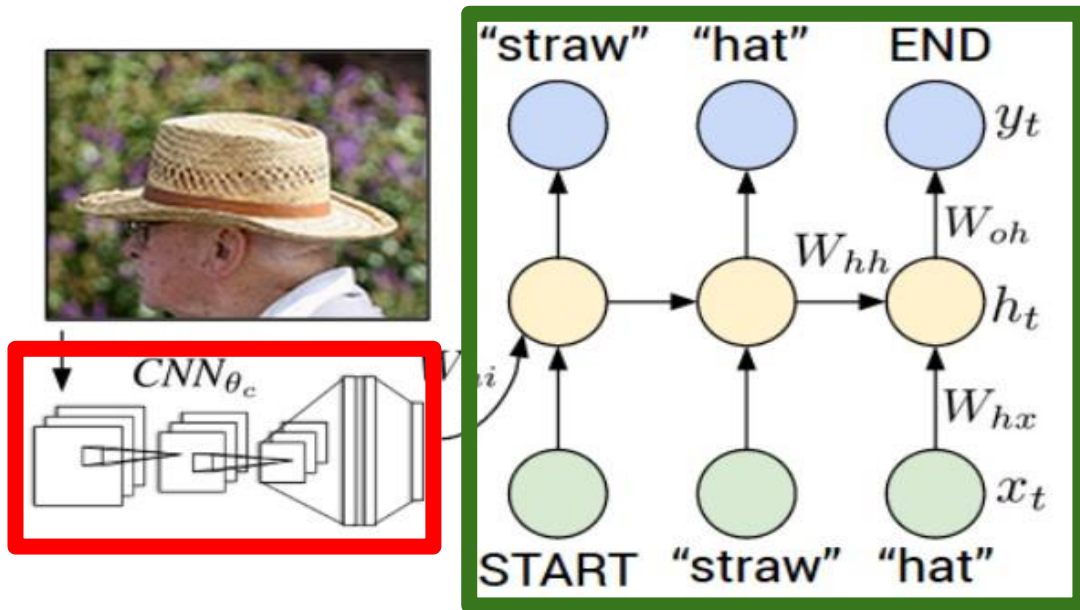
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

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

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

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

Recurrent Neural Network



Convolutional Neural Network



test image

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax



test image

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

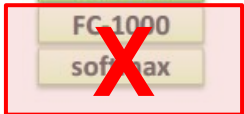
FC-4096

FC-1000

softmax



test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096



test image

x0
<STA
RT>

<START>

image

conv-64
conv-64
maxpool

conv-128
conv-128
maxpool

conv-256
conv-256
maxpool

conv-512
conv-512
maxpool

conv-512
conv-512
maxpool

FC-4096
FC-4096

V



test image

y0

h0

x0
<START
RT>

<START>

Wih



before:

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

now:

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

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

y0

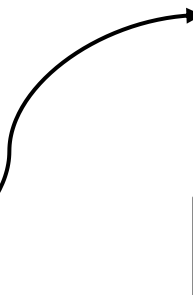
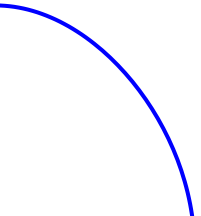
h0

x0
<START
RT>

straw

sample!

<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

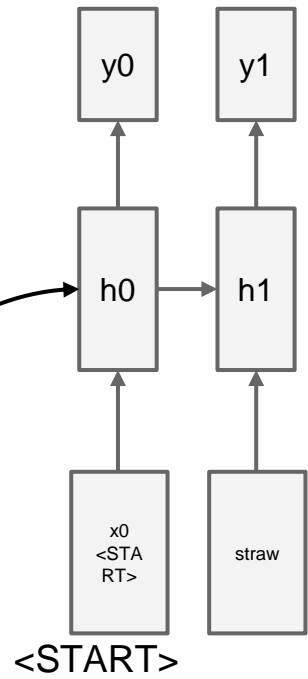
maxpool

FC-4096

FC-4096



test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096



test image

y0

y1

h0

h1

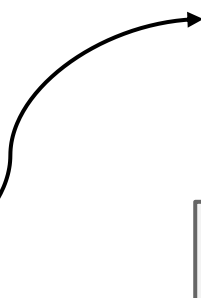
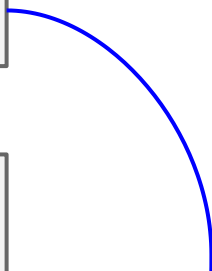
x0
<START
RT>

straw

hat

sample!

<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

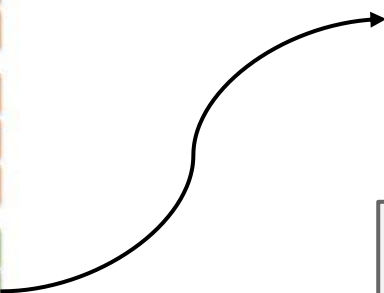
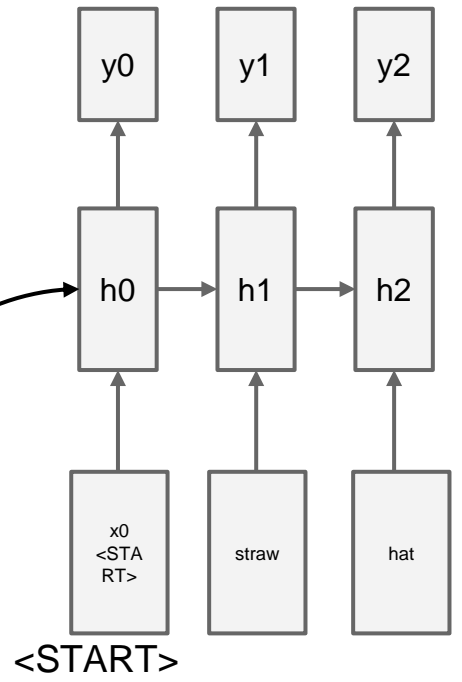
maxpool

FC-4096

FC-4096



test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

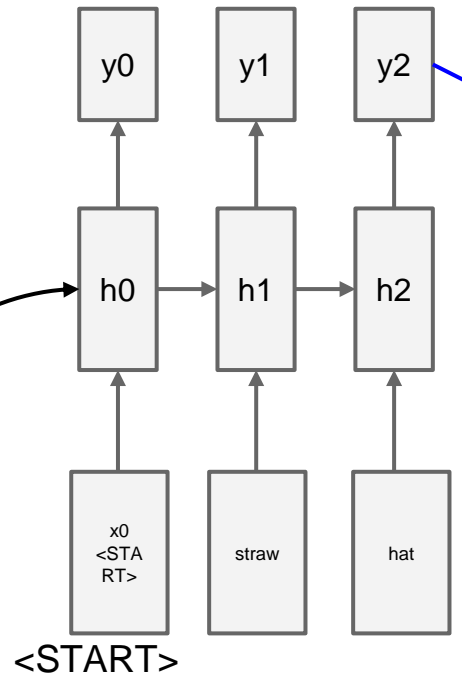
maxpool

FC-4096

FC-4096



test image



sample
 $\langle \text{END} \rangle$ token
 \Rightarrow finish.

Image Sentence Datasets

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



Microsoft COCO

[Tsung-Yi Lin et al. 2014]

mscoco.org

currently:

~120K images

~5 sentences each



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



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



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



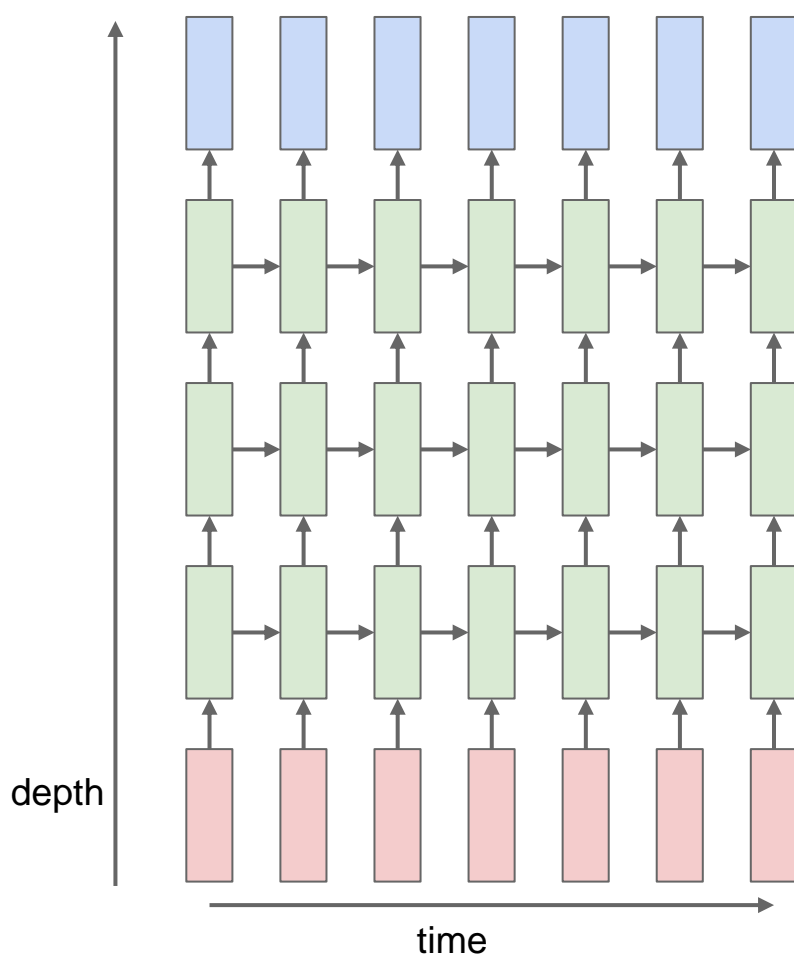
"a horse is standing in the middle of a road."

RNN:

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

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

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



Recurrent Neural Networks have loops

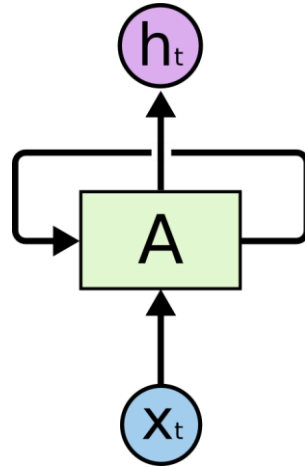


Figure credit: [Understanding LSTM Networks](#) on colah's blog

An unrolled recurrent neural network

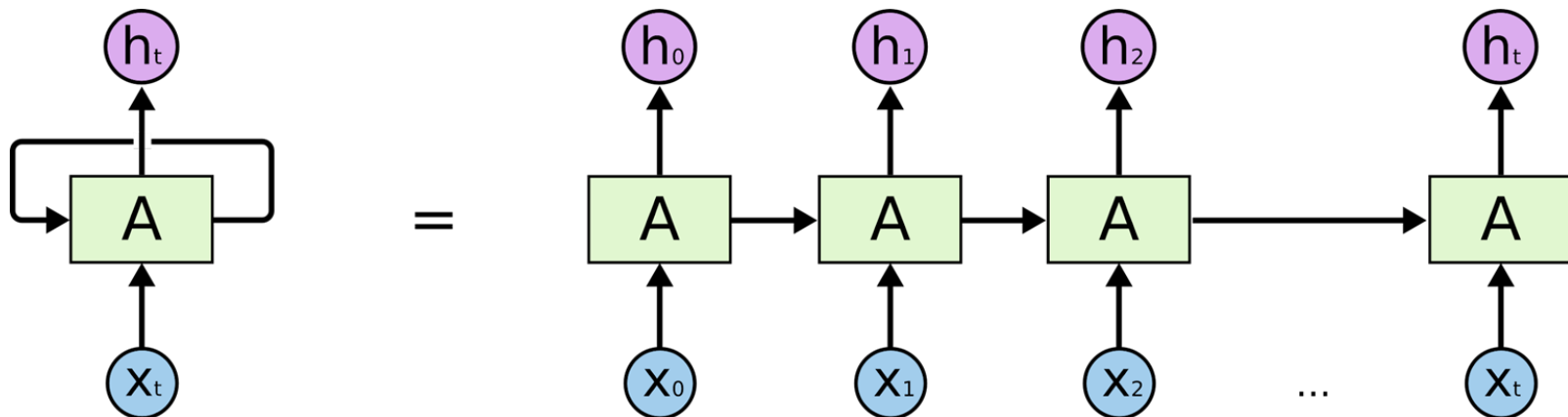


Figure credit: [Understanding LSTM Networks](#) on colah's blog

Problem of Long-Term Dependencies

“the clouds are in the *sky*”

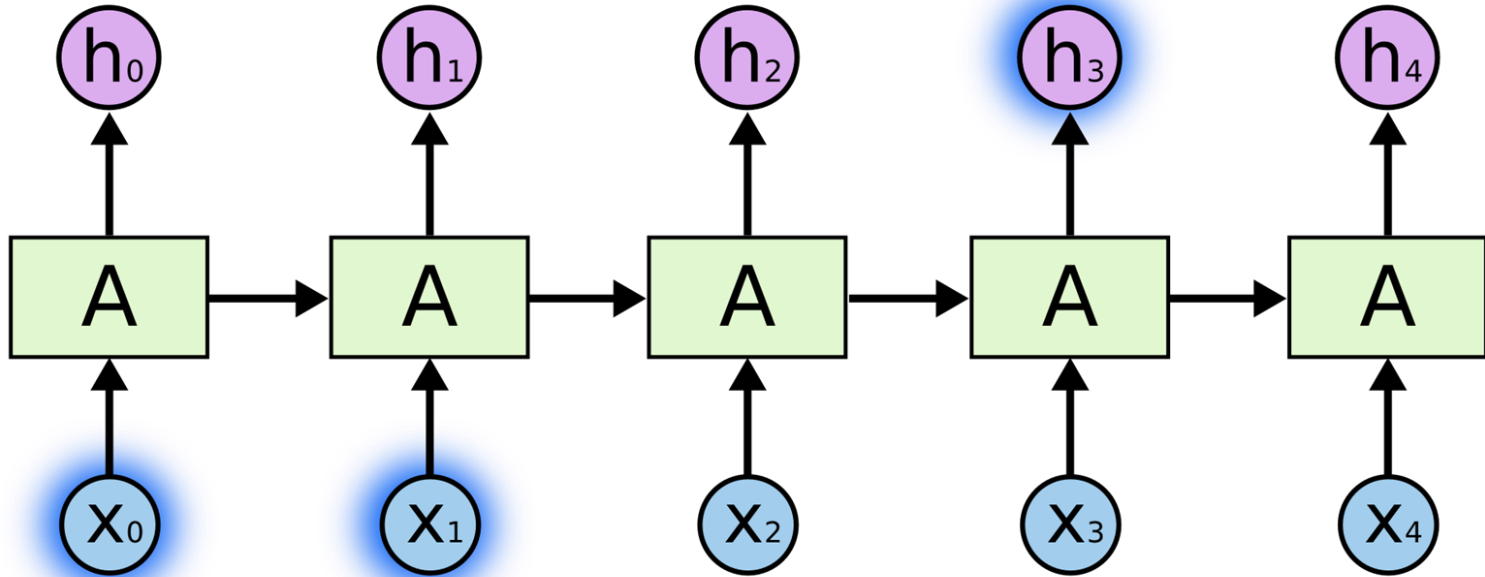


Figure credit: [Understanding LSTM Networks](#) on colah's blog

Problem of Long-Term Dependencies

“I grew up in France... I speak fluent *French*.”

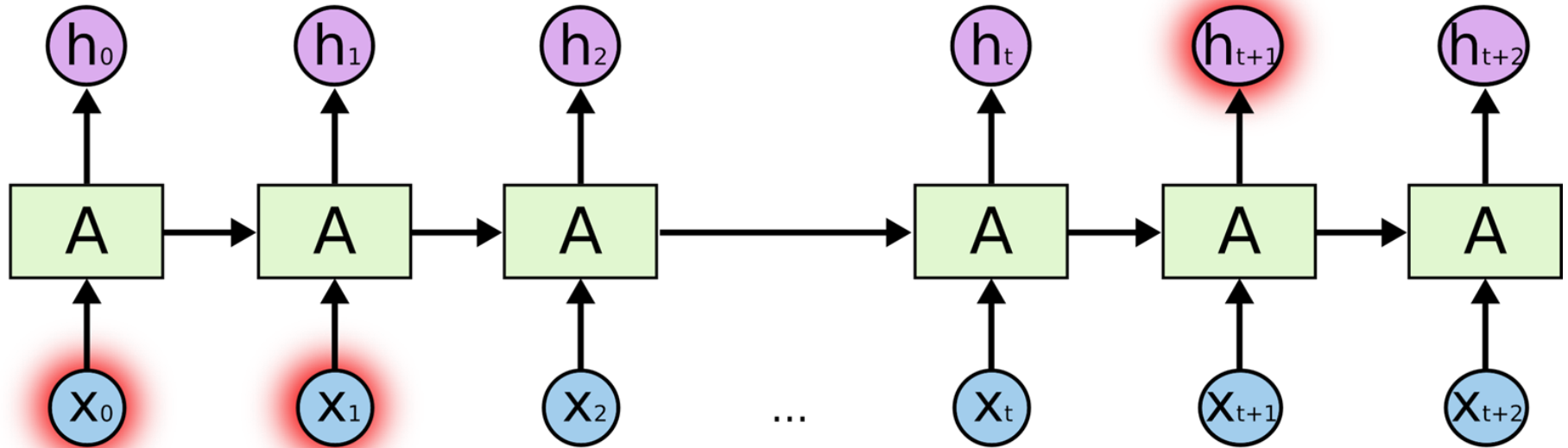


Figure credit: [Understanding LSTM Networks](#) on colah's blog

RNN:

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

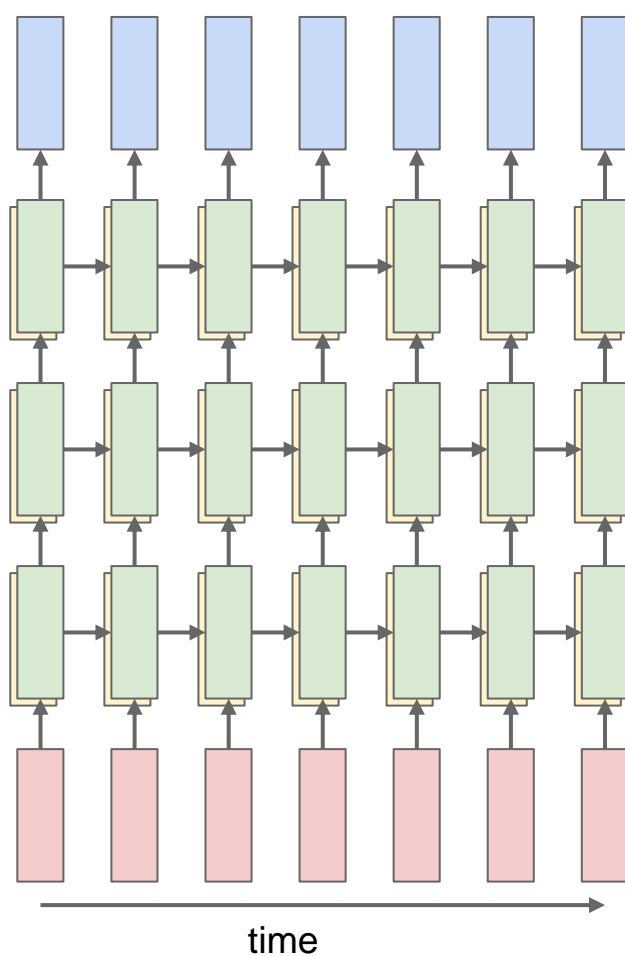
$$h \in \mathbb{R}^n. \quad W^l [n \times 2n]$$

LSTM:

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

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

depth



END