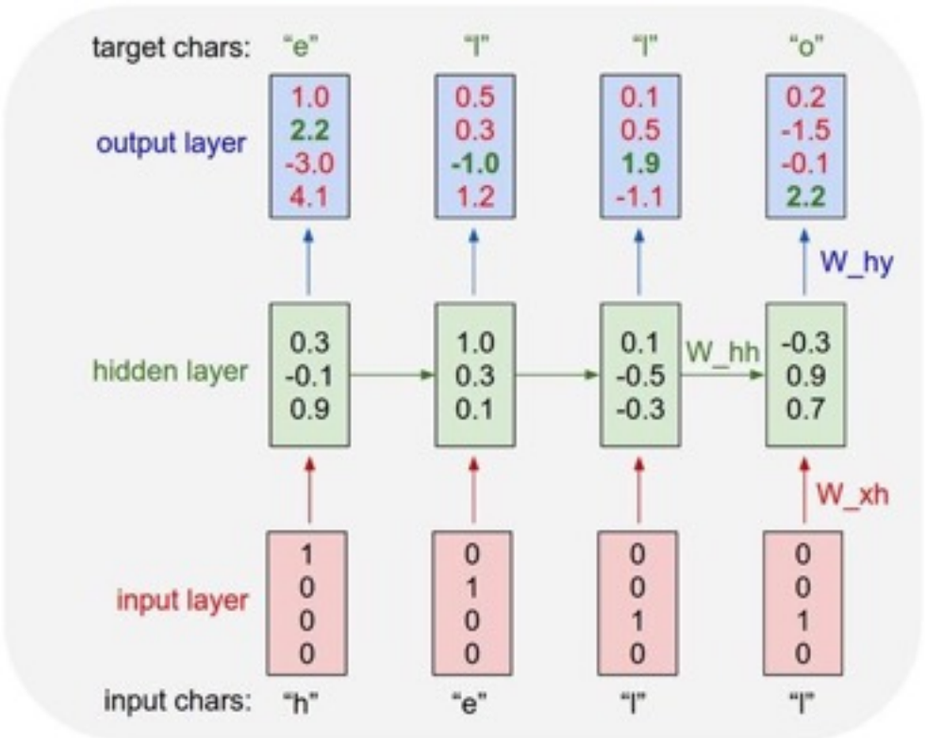
Sequence to Sequence learning with encoder decoder Model

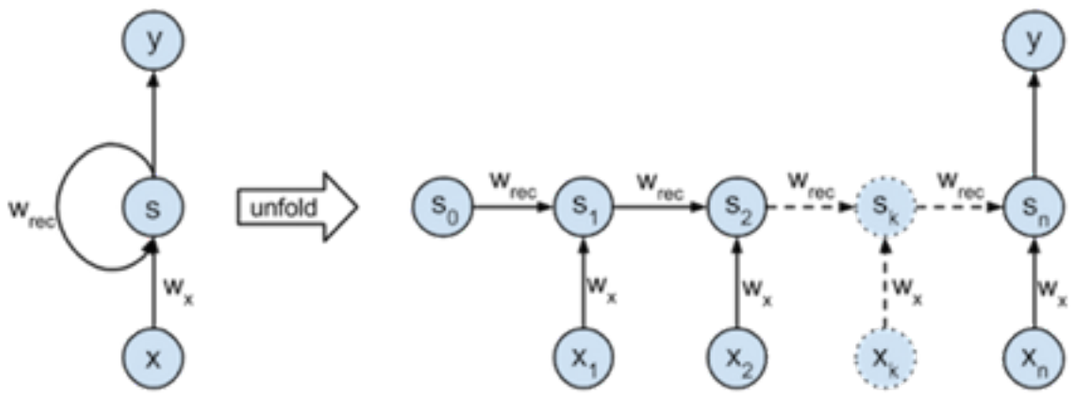
Recurrent Neural Networks

应用背景：使用机器学习方法，做 输入序列 到另外一个领域的输出单位 / 序列 的转换

tasks：

language modelling　　sentiment analysis





Feedforward:

$$
\begin{aligned}
h_t &= \text{sigm}\left(W^{\text{hx}} x_t + W^{\text{hh}} h_{t-1}\right) \\
y_t &= W^{\text{yh}} h_t
\end{aligned}
$$

To constrain: $w_1 = w_2$

we need: $\Delta w_1 = \Delta w_2$

compute: $\frac{\partial E}{\partial w_1}$ *and* $\frac{\partial E}{\partial w_2}$

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ *for* w_1 *and* w_2

```
class RNN:
    def __init__(self, in_dim, hidden_dim, model, include_h0=True, prefix='rnn'):
        self.Wih = model.add_parameters((hidden_dim, in_dim), prefix + '_Wih')
        self.Whh = model.add_parameters((hidden_dim, hidden_dim), prefix + '_Whh')
        self.bh = model.add_parameters(hidden_dim, prefix + '_bh')
        if include_h0:
            self.h0 = model.add_parameters(hidden_dim, prefix + '_h0')
        else:
            self.h0 = theano.shared(np.zeros(hidden_dim, theano.config.floatX), prefix + '_h0')

    def initial_hidden(self):
        return [self.h0]

    def recurrence(self, x_curr, h_prev):
        h_t = T.tanh(T.dot(self.Wih, x_curr) + T.dot(self.Whh, h_prev) + self.bh)
        return [h_t]
```

“linear” RNN

output：

$$\frac{\partial_E}{\partial W_{jk}} = O_j \delta k$$

sigmoid actfunc

$$\delta_k = O_k(1-O_k)(O_k-t_k)$$

hidden：

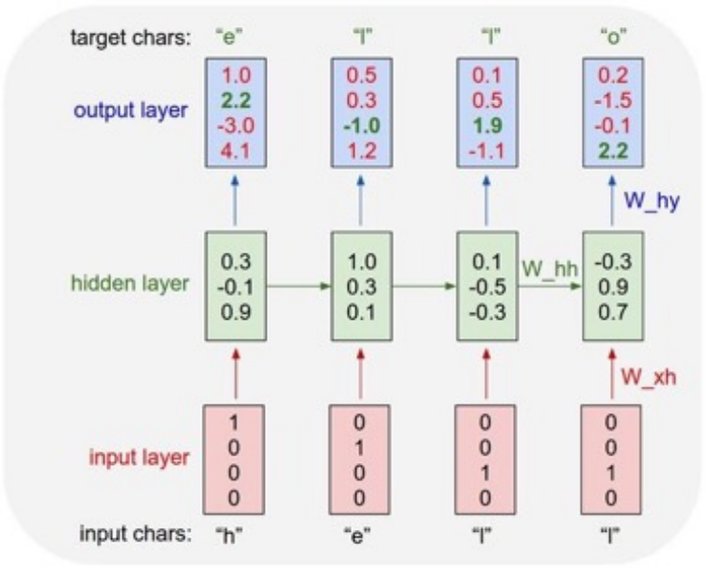
$$\frac{\partial_E}{\partial W_{ij}} = O_i \delta j$$

$$\delta j = O_j(1-O_j)\sum_{k\in k}\delta k W_{jk}$$

$$\Delta W = -a\delta_l O_{l-1}$$

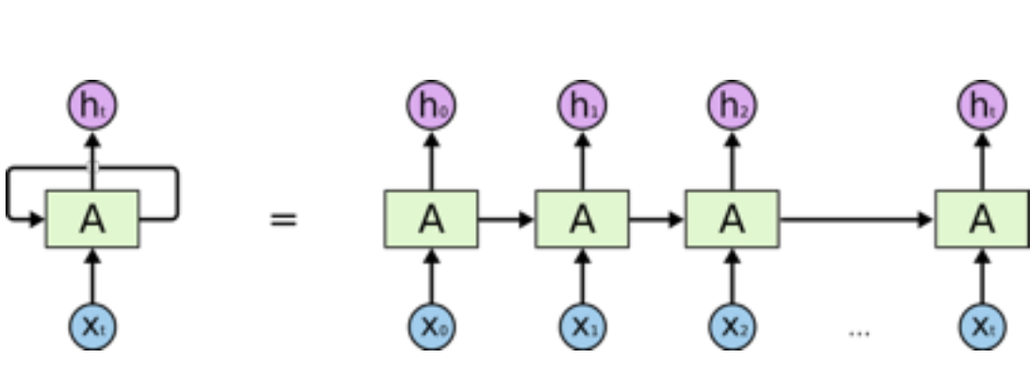
$$\Delta W = \Delta W_{t1} + \Delta W_{t2} + \ldots$$

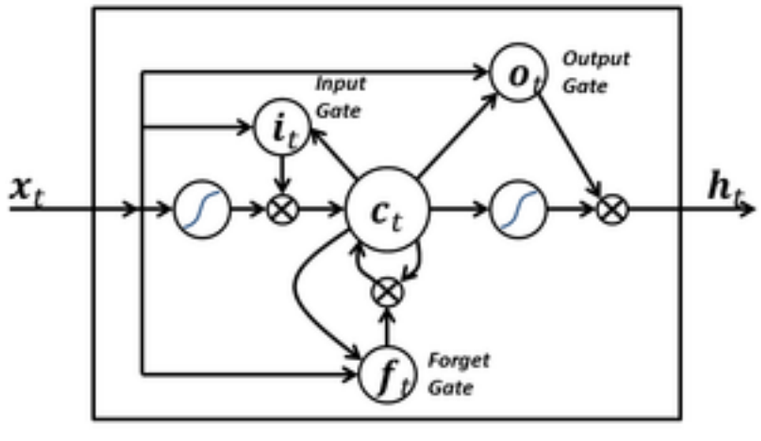
“sequence” RNN

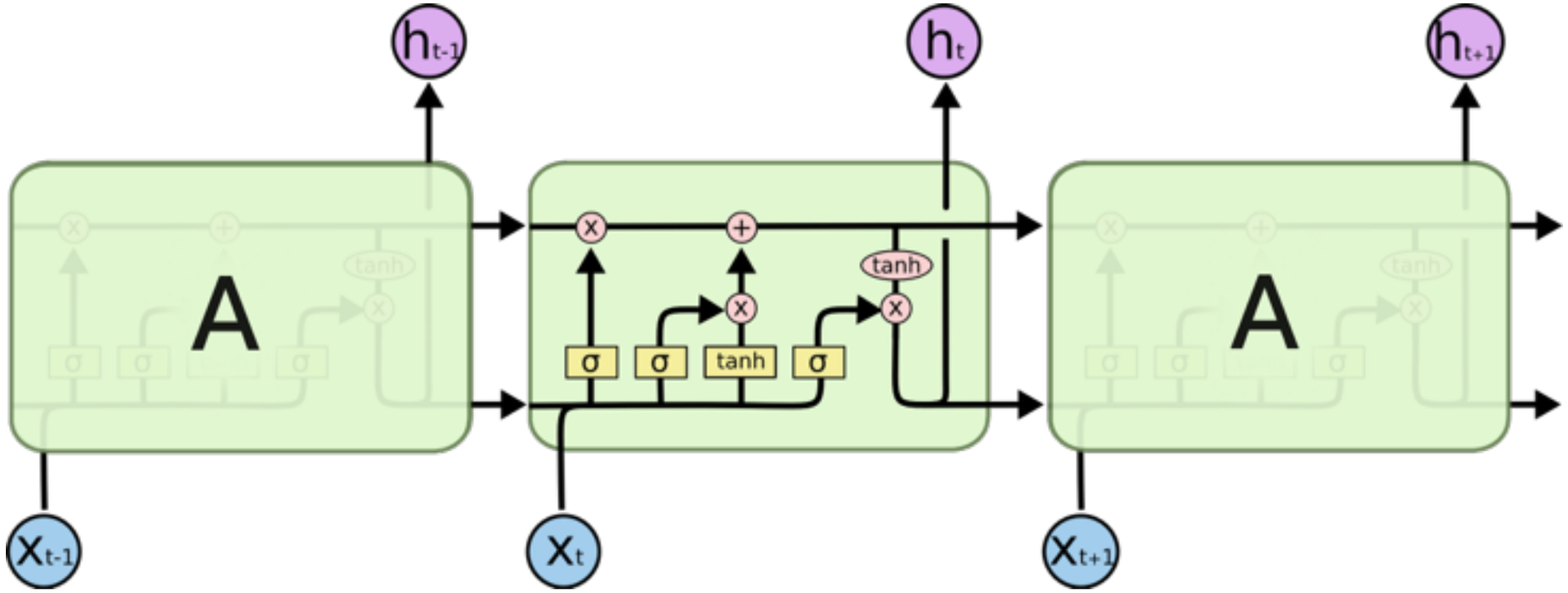


$$\Delta W = -aO_{l-1}(\delta_l^1 + \delta_l^2 + \ldots)$$

LSTM


A
=
A
A
A
A
...


Input Gate
Output Gate
Forget Gate


A
A
tanh
σ
σ
tanh
σ
tanh

```lua
require 'nn'
require 'nngraph'

function LSTM.create(input_size, rnn_size)
  ------------------------ input structure ------------------------
  local inputs = {}
  table.insert(inputs, nn.Identity()())   -- network input
  table.insert(inputs, nn.Identity()())   -- c at time t-1
  table.insert(inputs, nn.Identity()())   -- h at time t-1
  local input = inputs[1]
  local prev_c = inputs[2]
  local prev_h = inputs[3]

  ------------------------ preactivations -------------------------
  local i2h = nn.Linear(input_size, 4 * rnn_size)(input)    -- input to hidden
  local h2h = nn.Linear(rnn_size, 4 * rnn_size)(prev_h)     -- hidden to hidden
  local preactivations = nn.CAddTable()({i2h, h2h})         -- i2h + h2h

  ------------------- non-linear transforms -------------------
  -- gates
  local pre_sigmoid_chunk = nn.Narrow(2, 1, 3 * rnn_size)(preactivations)
  local all_gates = nn.Sigmoid()(pre_sigmoid_chunk)

  -- input
  local in_chunk = nn.Narrow(2, 3 * rnn_size + 1, rnn_size)(preactivations)
  local in_transform = nn.Tanh()(in_chunk)

  ---------------------- gate narrows -----------------------
  local in_gate = nn.Narrow(2, 1, rnn_size)(all_gates)
  local forget_gate = nn.Narrow(2, rnn_size + 1, rnn_size)(all_gates)
  local out_gate = nn.Narrow(2, 2 * rnn_size + 1, rnn_size)(all_gates)

  --------------------- next cell state ---------------------
  local c_forget = nn.CMulTable()({forget_gate, prev_c})  -- previous cell state contribution
  local c_input = nn.CMulTable()({in_gate, in_transform}) -- input contribution
  local next_c = nn.CAddTable()({
    c_forget,
    c_input
  })
```

$$i_t = g(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = g(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = g(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$c\_in_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{c\_in})$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot c\_in_t$$

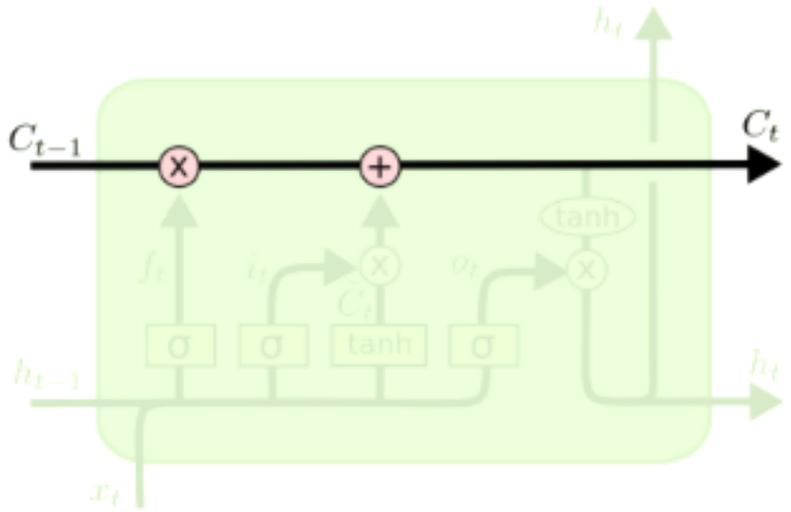
$$h_t = o_t \cdot tanh(c_t)$$

Rnn的

$$\begin{aligned} h_t &= \text{sigm}\left(W^{\text{hx}}x_t + W^{\text{hh}}h_{t-1}\right) \\ y_t &= W^{\text{yh}}h_t \end{aligned}$$

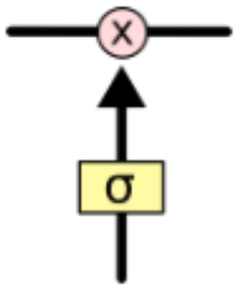
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

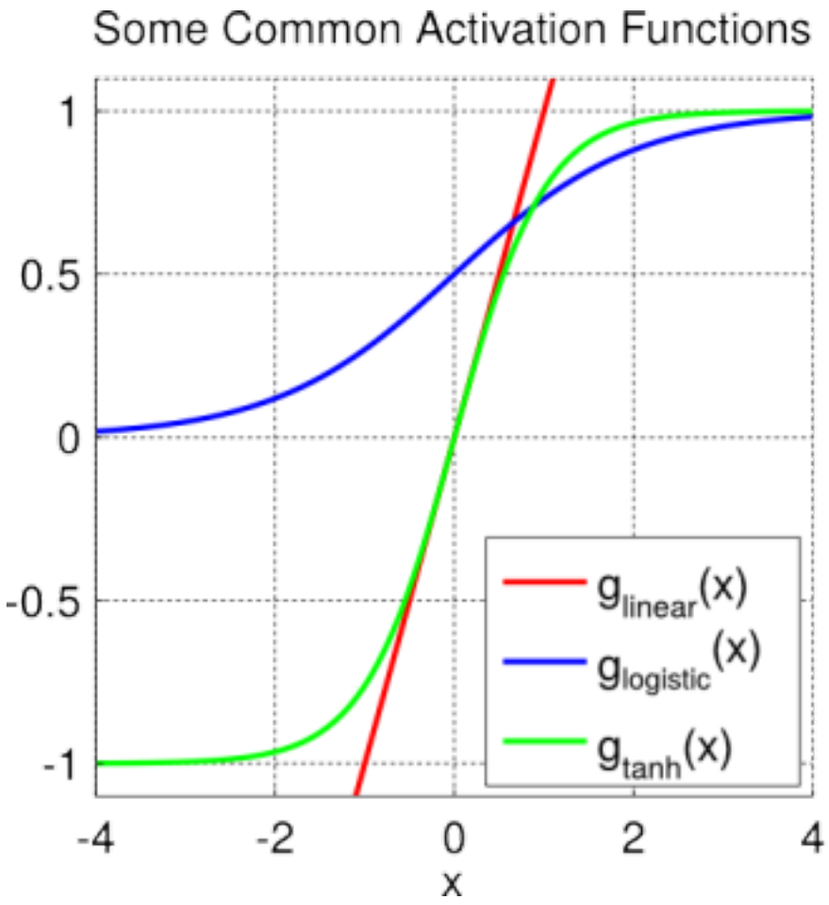
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

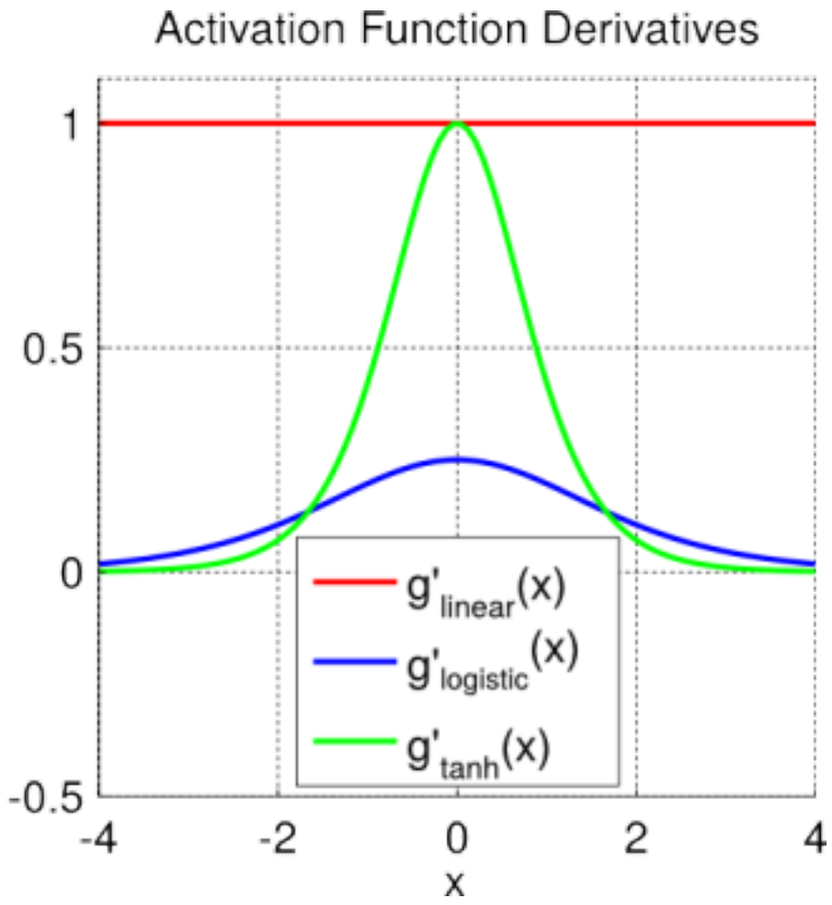


The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



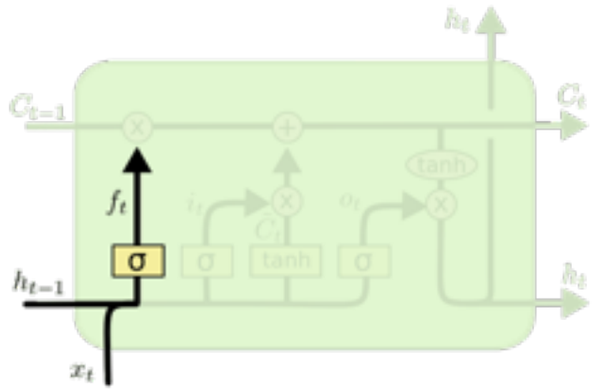




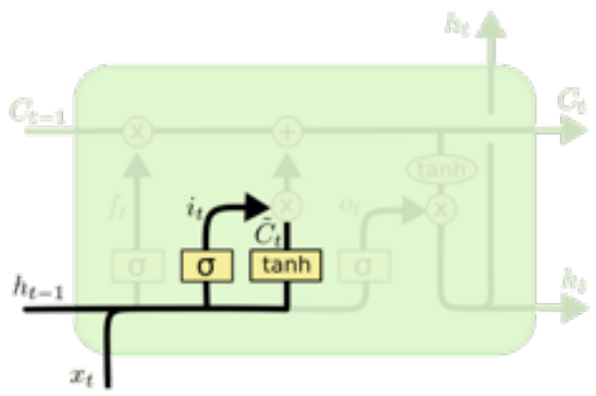
$$\delta_k = O_k(1-O_k)(O_k-t_k)$$

$$\delta j = O_j(1-O_j)\sum_{k\in k}\delta k W_{jk}$$

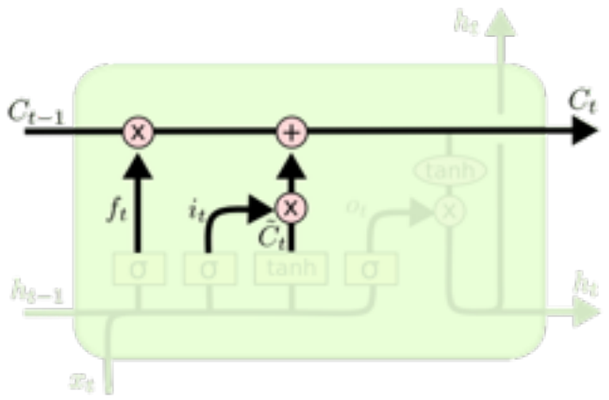
In the recurrence of the LSTM the activation function is the identity function with a derivative of 1.0. So, the backpropagated gradient neither vanishes or explodes when passing through, but remains constant.



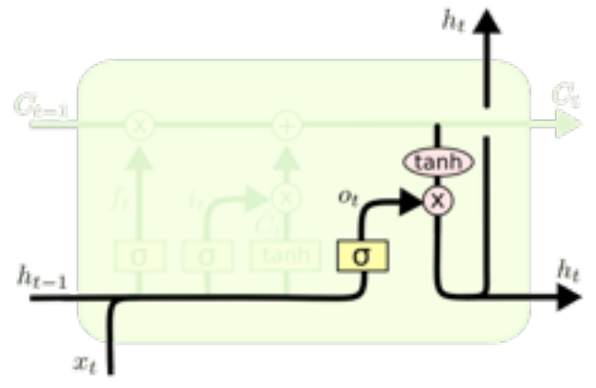
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



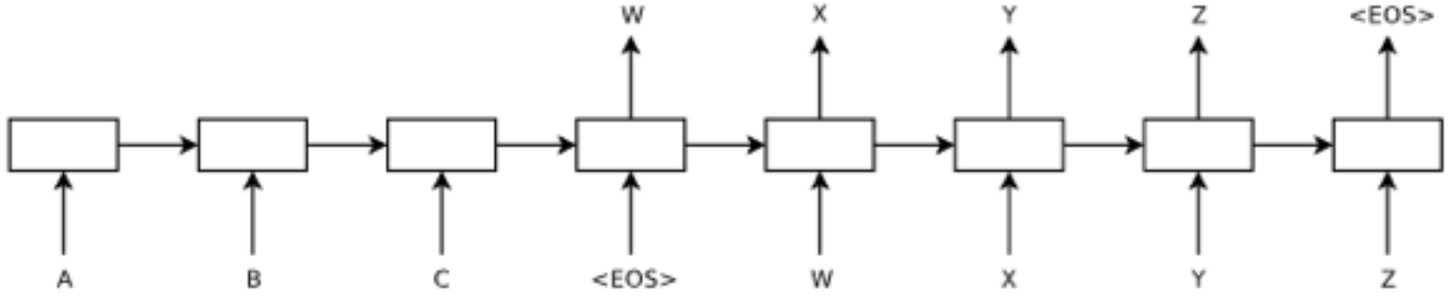
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever
Google
ilyasu@google.com

Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com



$$P(Y|X) = \prod_{t \in [1, n_y]} p(y_t | x_1, x_2, ..., x_t, y_1, y_2, ..., y_{t-1}) = \prod_{t \in [1, n_y]} \frac{\exp(f(h_{t-1}, e_{y_t}))}{\sum_{y'} \exp(f(h_{t-1}, e_{y'}))}$$

```
self.encoder_rnn = rnn_factory(nes, nh, self, prefix='encoder')
self.decoder_rnn = rnn_factory(net, nh, self, include_h0=False, prefix='decoder')

self.add_parameters((nwt, nh), 'Woh')
self.add_parameters(nwt, 'bo')

self.src = T.ivector('source')
self.tgt = T.ivector('target')
self.inputs = [self.src, self.tgt]

# step 1: encode the source
xs = self.Es[self.src[:-1]]

encoded, _ = theano.scan(fn=self.encoder_rnn.recurrence, sequences=xs,
        outputs_info=self.encoder_rnn.initial_hidden(), n_steps=xs.shape[0])

encoding = []

encoding = [encoded[0][-1]]

# step 2: decode the target
def recurrence(x_tm1, *h_tm1):
    hiddens_t = self.decoder_rnn.recurrence(x_tm1, *h_tm1)
    y_t = T.nnet.softmax(T.dot(self.Woh, hiddens_t[0]) + self.bo)

    return hiddens_t + [y_t]

xt = self.Et[self.src[::-1]]

outputs, _ = theano.scan(fn=recurrence, sequences=xt,
        outputs_info=encoding + [np.zeros(64, theano.config.floatX)] + [None], n_steps=xt.shape[0])
```