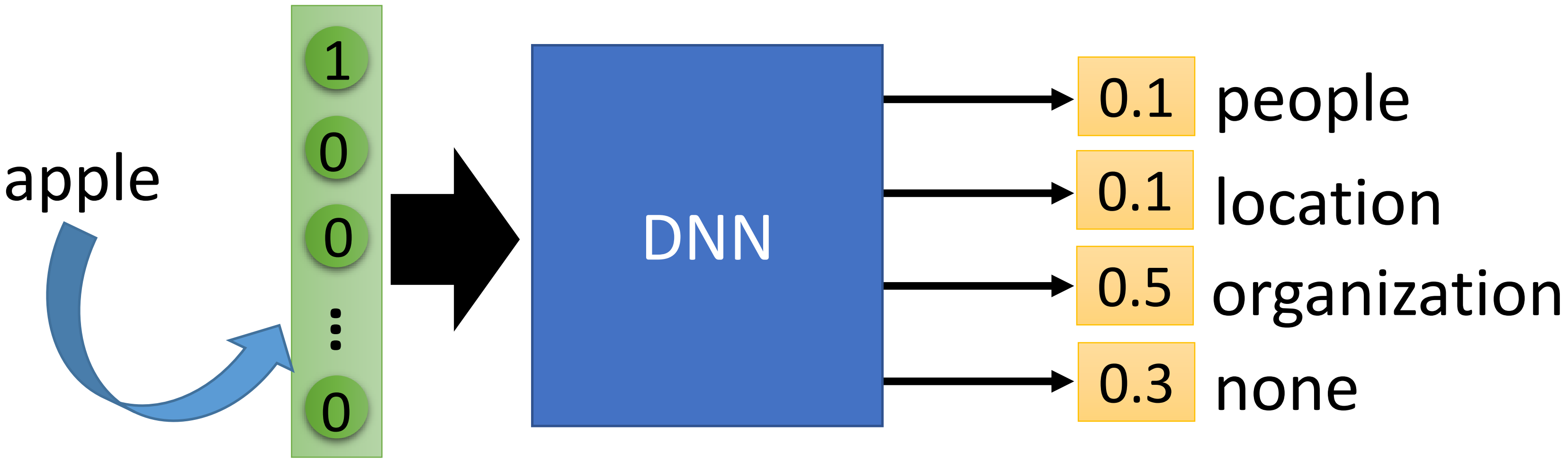
Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM)

Instructor: Dr. Mohammad Rashedur Rahman

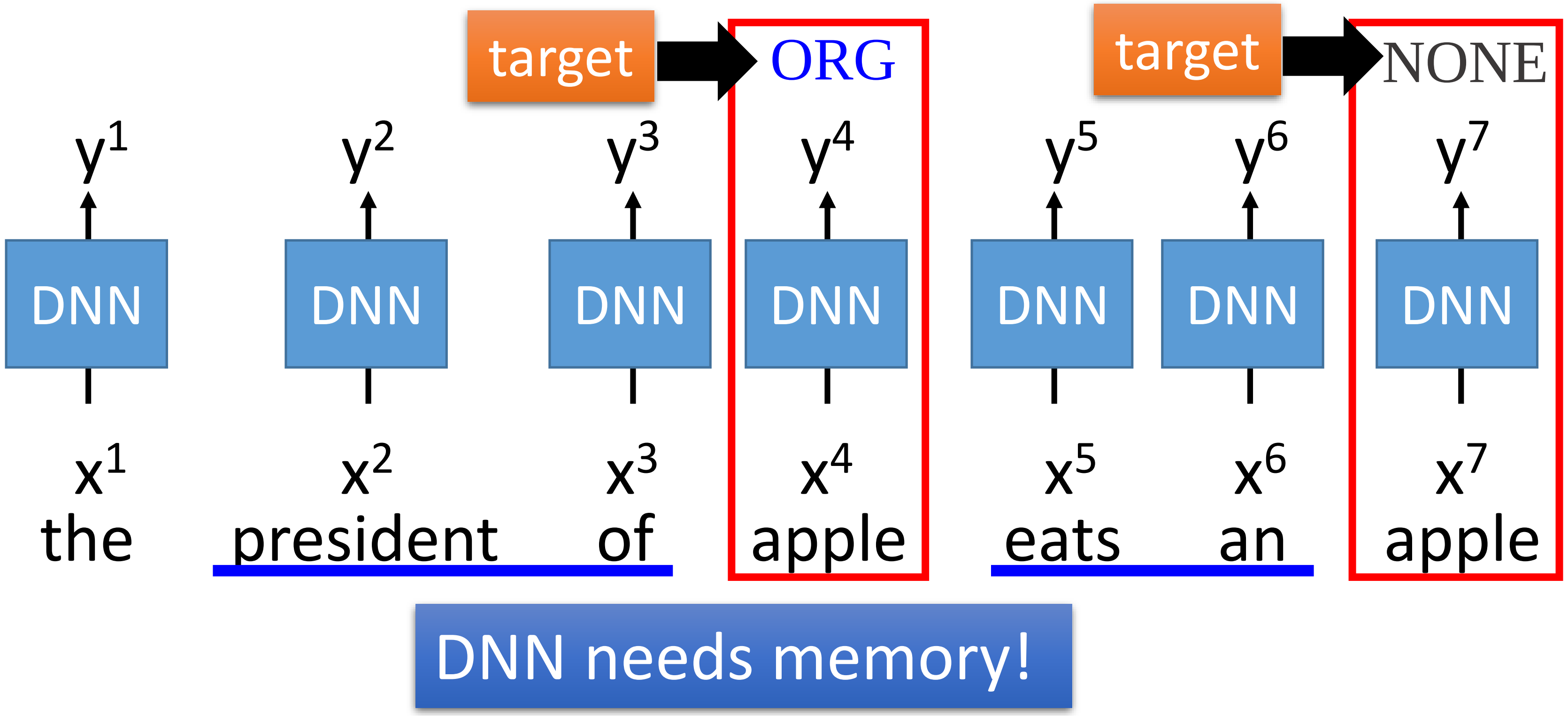
Neural Network needs Memory

- Name Entity Recognition
  - Detecting named entities like name of people, locations, organization, etc. in a sentence.



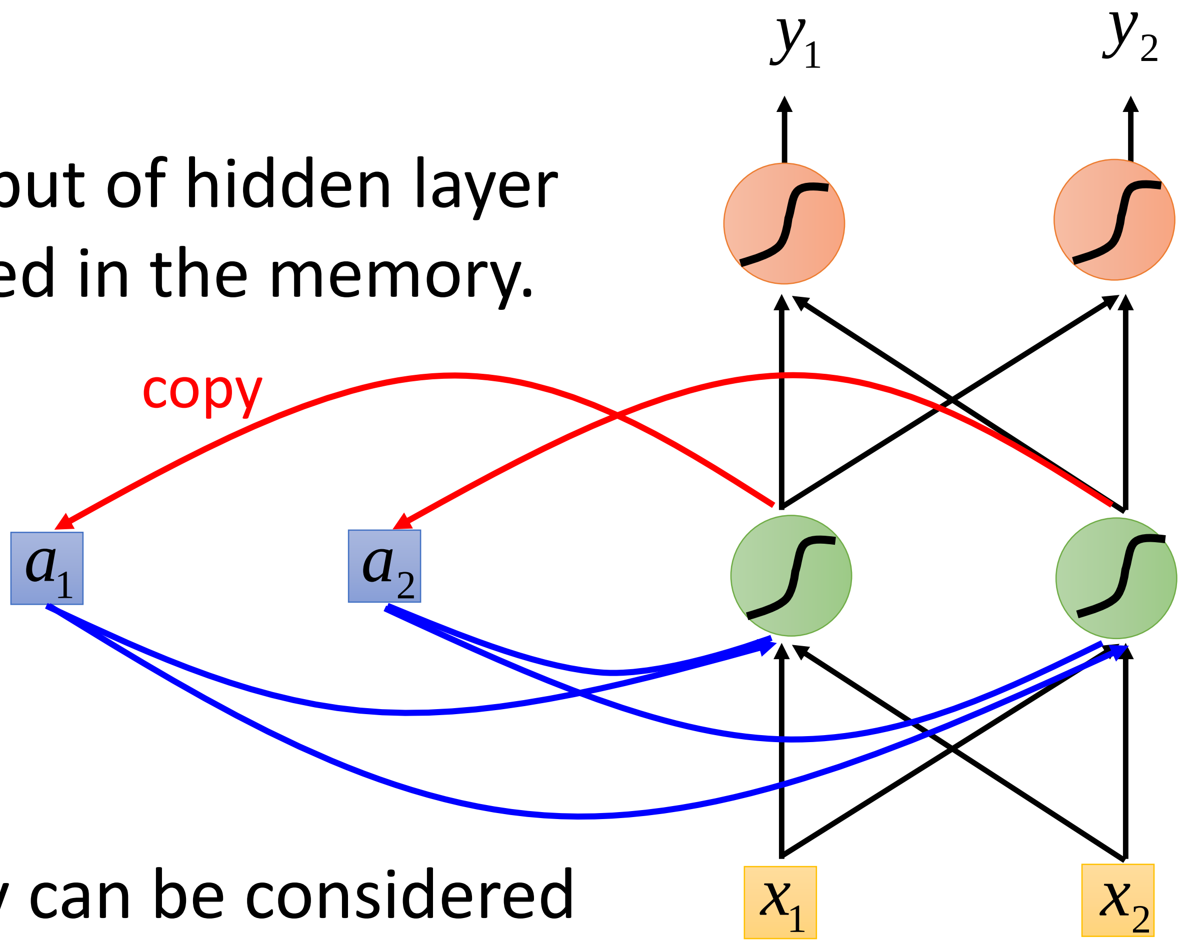
Neural Network needs Memory

- Name Entity Recognition
  - Detecting named entities like name of people, locations, organization, etc. in a sentence.



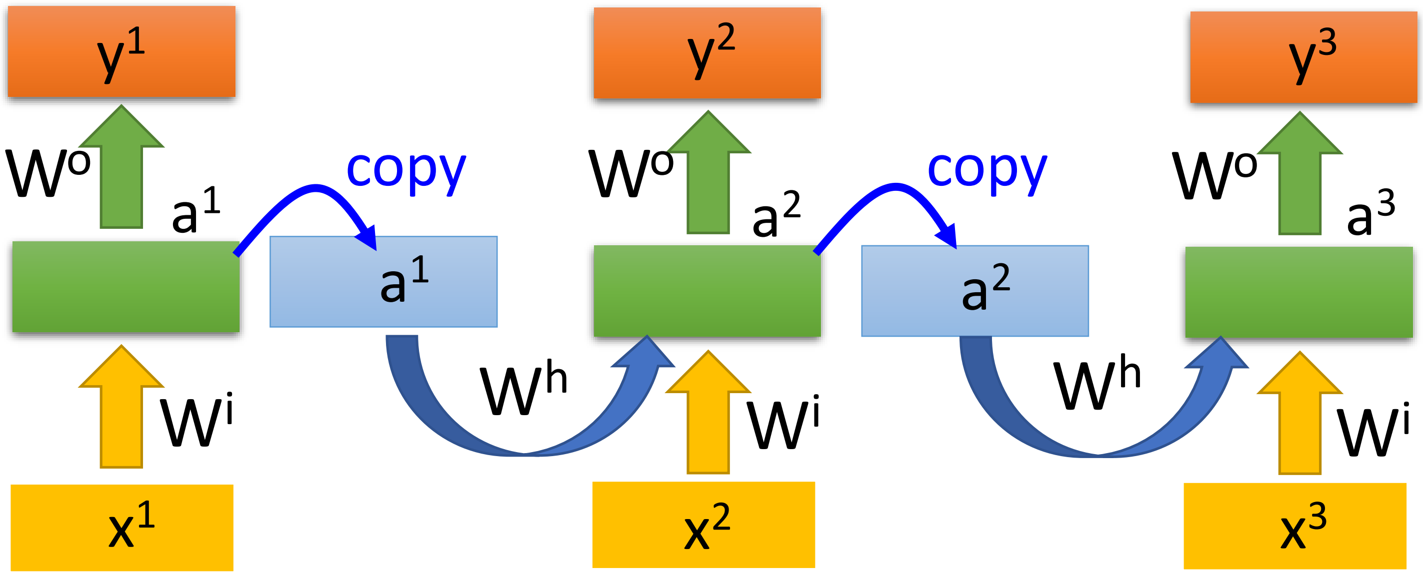
Recurrent Neural Network (RNN)

The output of hidden layer are stored in the memory.



Memory can be considered as another input.

RNN

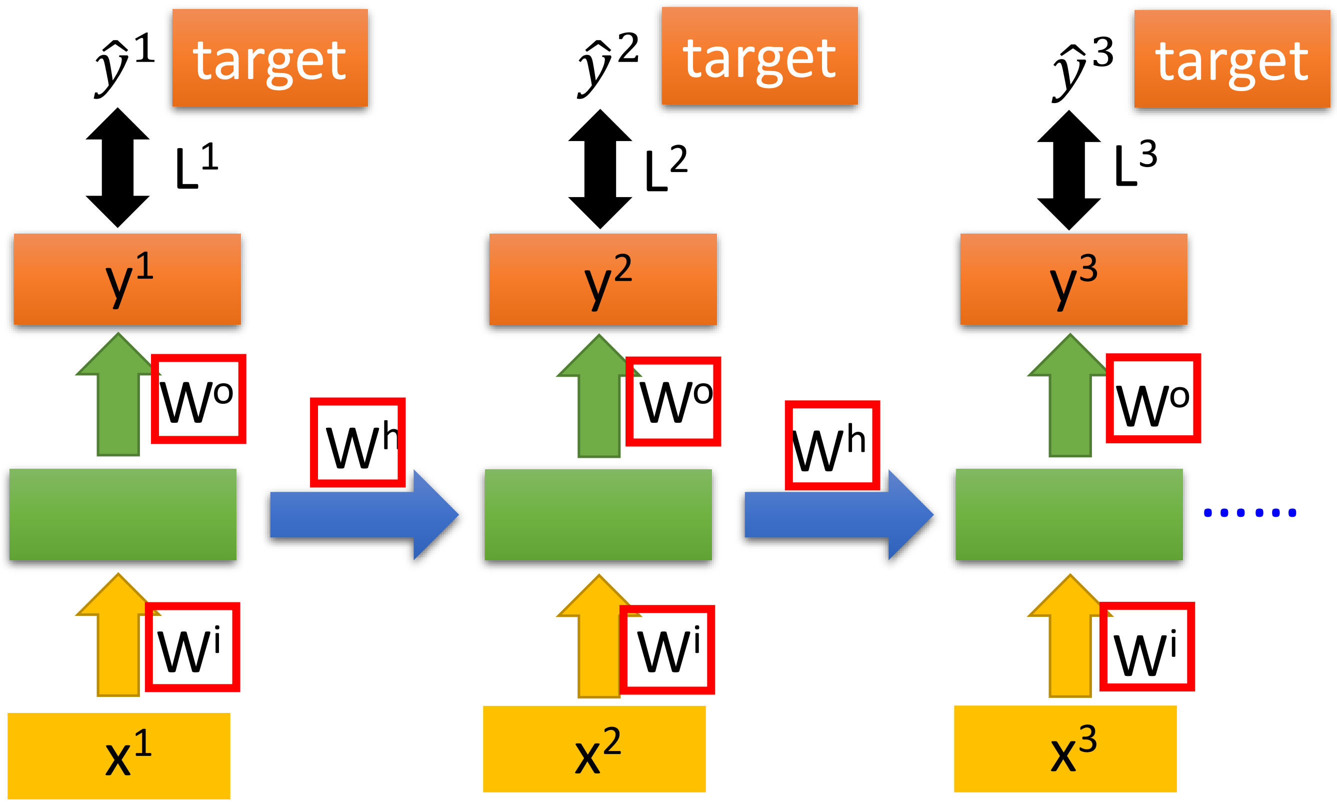


The same network is used again and again.

Output y^i depends on x^1, x^2, …… x^i

RNN

How to train?



Find the network parameters to minimize the total cost:

Backpropagation through time (BPTT)

RNN

A vanilla RNN to predict sequences from input

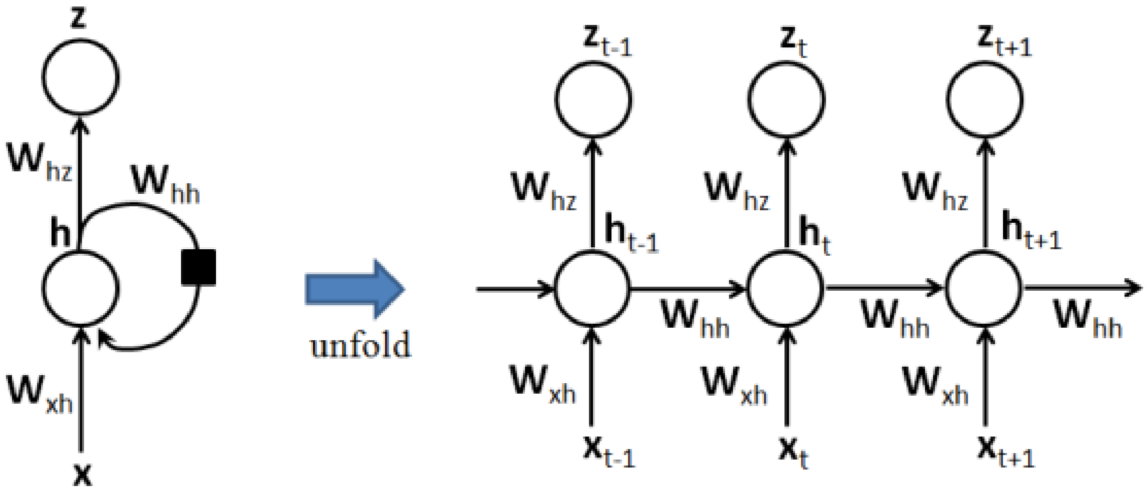
$$P(y_1, \ldots, y_T | \mathbf{x}_1, \ldots, \mathbf{x}_T)$$

- Forward propagation equations, assuming that hyperbolic tangent non-linearities are used in the hidden units and softmax is used in output for classification problems

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

$$\mathbf{z}_t = \mathrm{softmax}(\mathbf{W}_{hz}\mathbf{h}_t + \mathbf{b}_z)$$

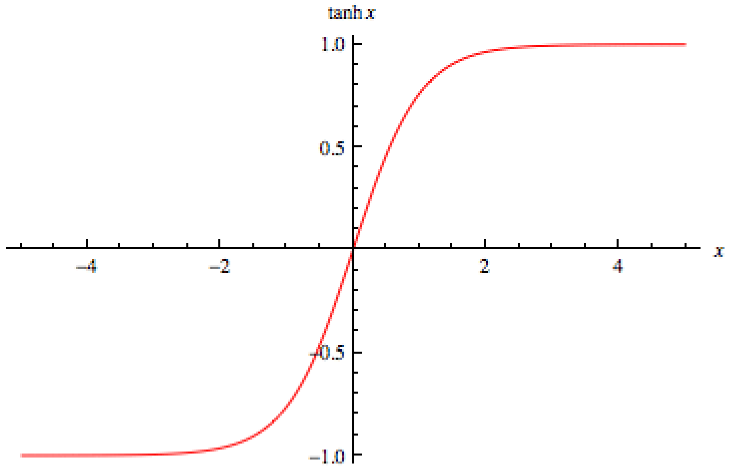
$$p(y_t = c) = z_{t,c}$$



RNN

- Tanh function (shift the center of Sigmoid to the origin)

$$f(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$



- Softmax: mostly used as output non-linearrity for predicting discrete probabilities

$$f(s_k) = \frac{e^{s_k}}{\sum_{k'=1}^{C} e^{s_{k'}}}$$

BPTT (Back Propagation Through Time)

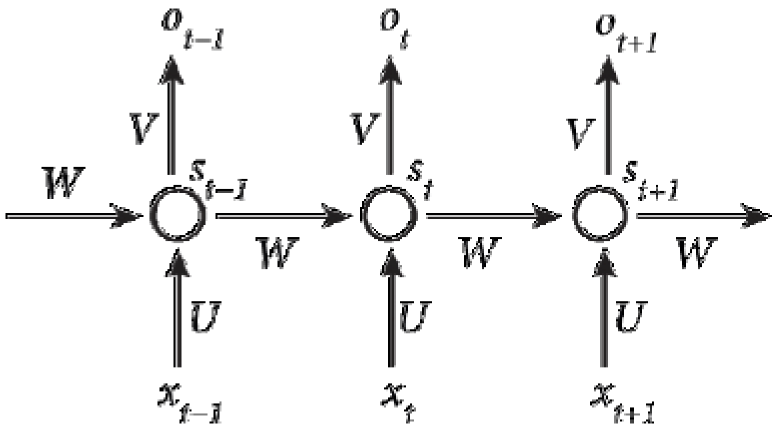
Recall

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

Loss Function

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$
$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$
$$= -\sum_t y_t \log \hat{y}_t$$

Recall:



BPTT (cont..)

Goal

- Calculate error gradients w.r.t. U, V and W
- Learn weights using SGD

- Just like we sum up errors, we also sum up gradients at each step for one training example

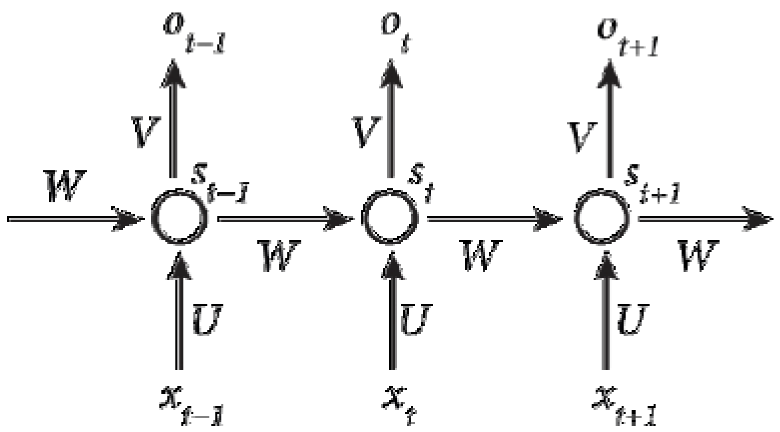
$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

BPTT (cont..)

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

$$\begin{aligned}\frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}\\ &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}\\ &= (\hat{y}_3 - y_3) \otimes s_3\end{aligned}$$

Recall:



Where z3 = Vs3 and is X is outer product

- What do you think about gradient of E3 w.r.t. W?

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

- Is that complete?

$$s_3 = \tanh(Ux_t + Ws_2)$$

Chain rule needs to be applied again

BPTT (cont..)

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

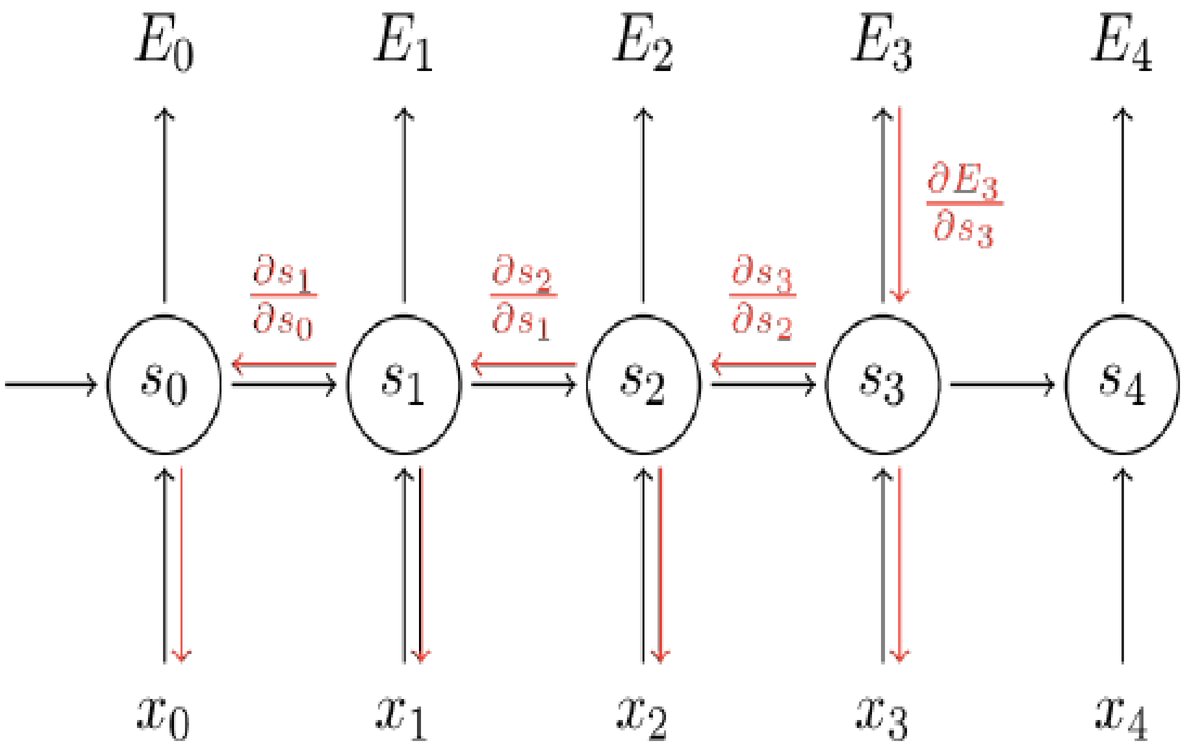
We would have:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

Similar to backprop, you can define:

$$\delta_2^{(3)} = \frac{\partial E_3}{\partial z_2} = \frac{\partial E_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial z_2}$$

with $z_2 = Ux_2 + Ws_1$

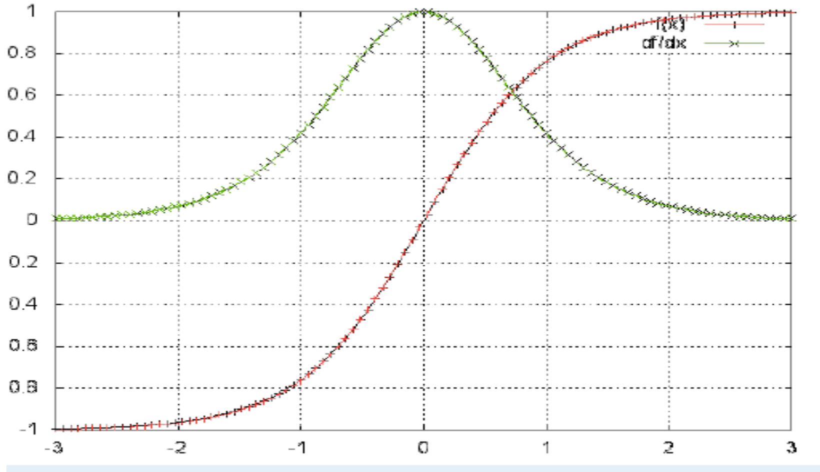


Vanishing Gradient Problem

- Sequences (sentences) can be quite long, perhaps 20 words or more - need to back-propagate through many layers!
- **Vanishing gradient problem**

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}$$

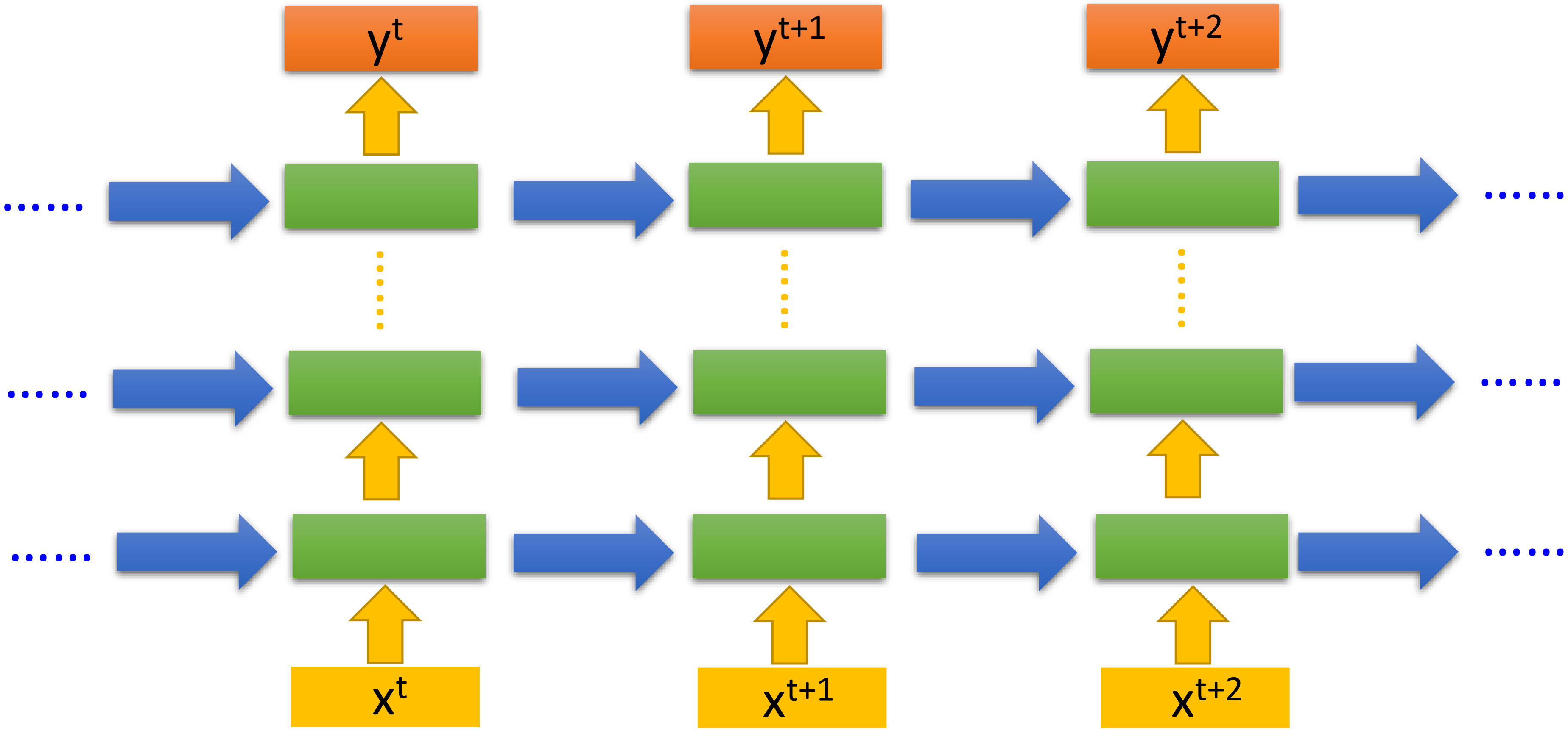


- For sigmoid activations ->gradient is upper-bounded by 1
- What does this tell you?
- Gradients will vanish over time , and long-range dependencies will only worsen learning

Vanishing Gradient Problem (Cont..)

- Derivative of a vector w.r.t a vector is a matrix called jacobian
- 2-norm of the above Jacobian matrix has an upper bound of 1
- tanh maps all values into a range between -1 and 1, and the derivative is bounded by 1
- With multiple matrix multiplications, gradient values shrink exponentially
- Gradient contributions from “far away” steps become zero
- Depending on activation functions and network parameters, gradients could explode instead of vanishing

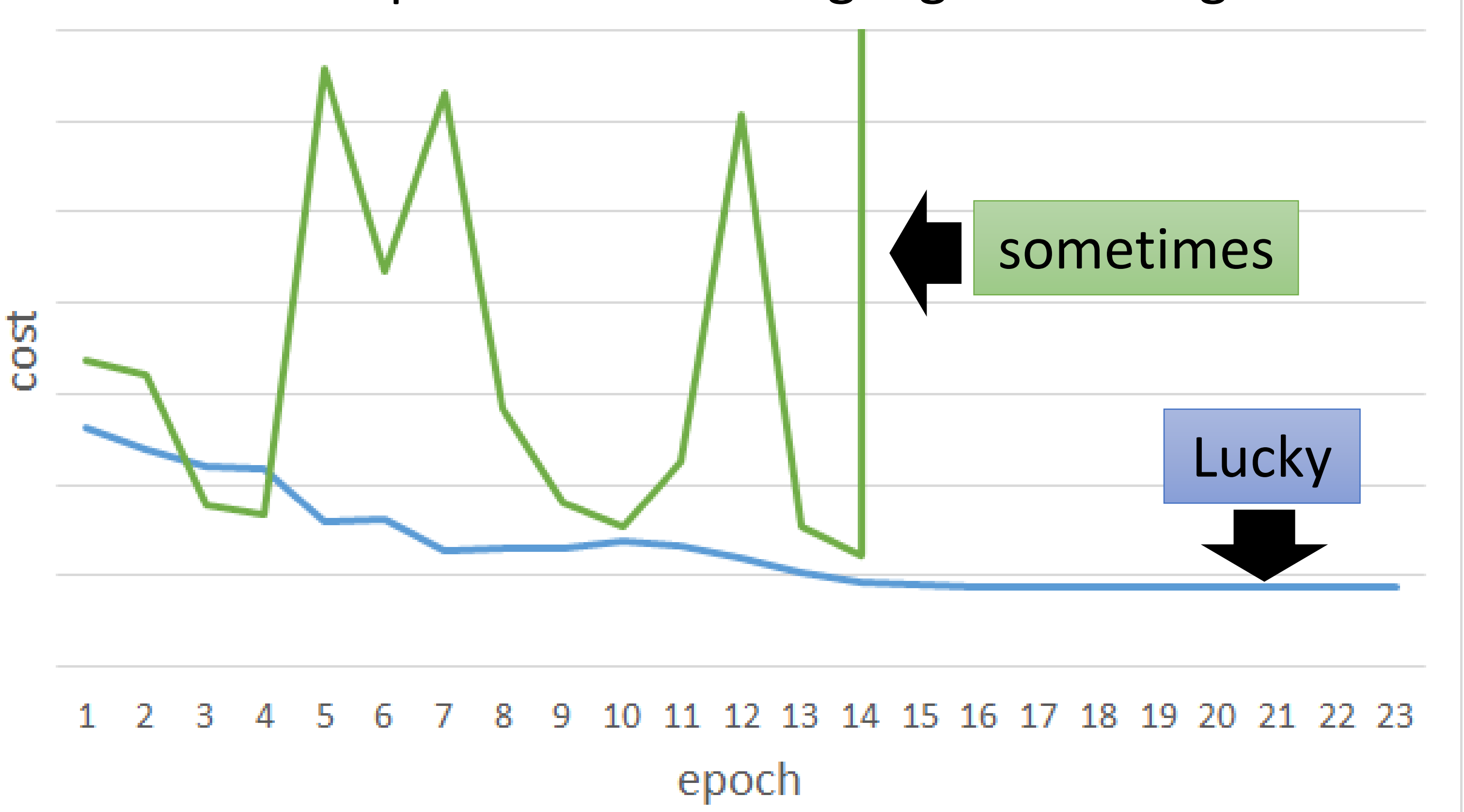
RNN can be deep also…



Unfortunately ……

- RNN-based network is not always easy to learn

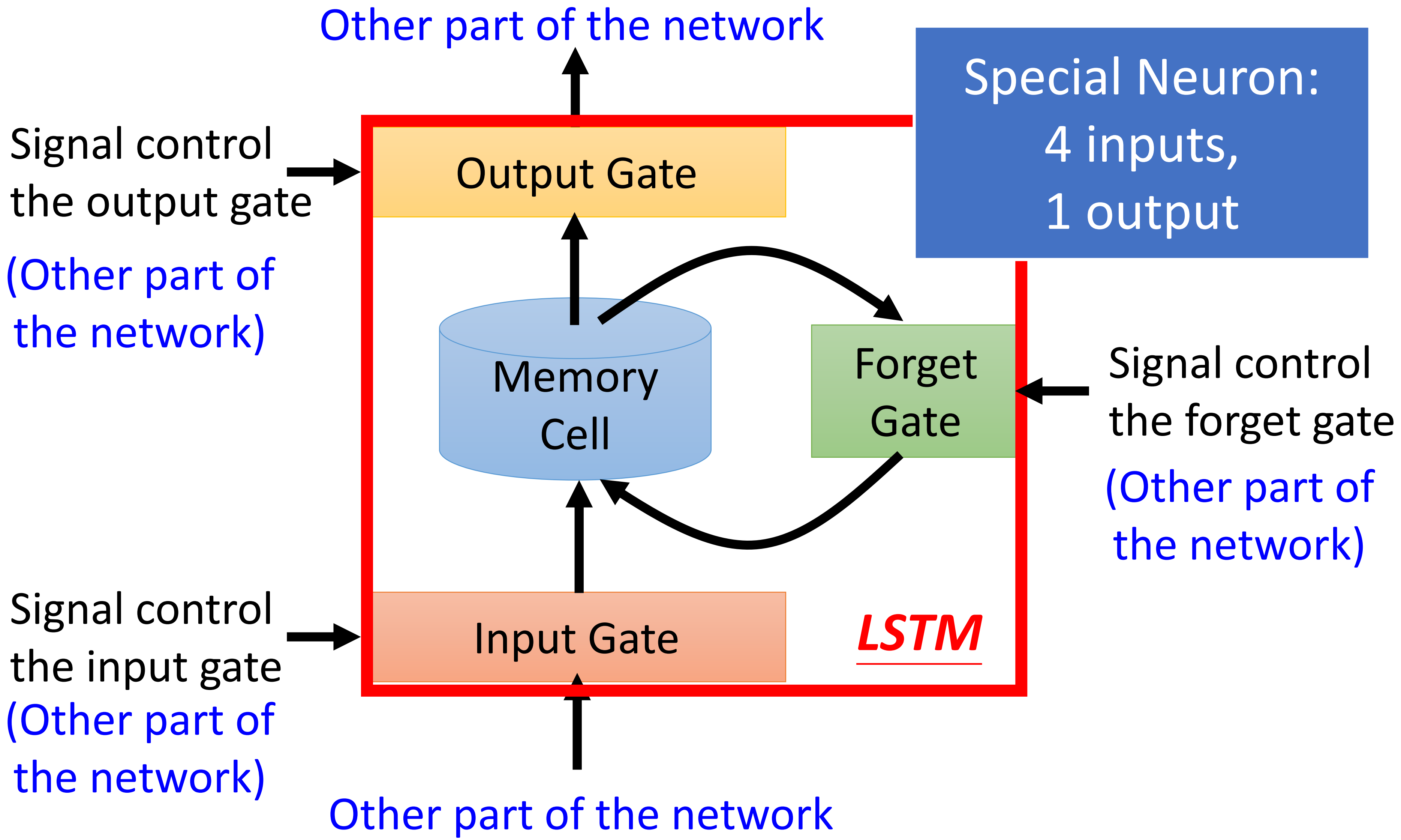
Real experiments on Language modeling



LSTM Solution

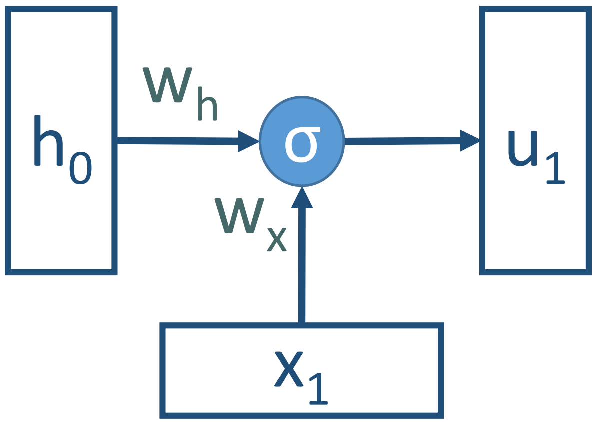
- Use memory cell to store information at each time step.
- Use “gates” to control the flow of information through the network.
  - Input gate: protect the current step from irrelevant inputs
  - Output gate: prevent the current step from passing irrelevant outputs to later steps
  - Forget gate: limit information passed from one cell to the next

Long Short-term Memory (LSTM)



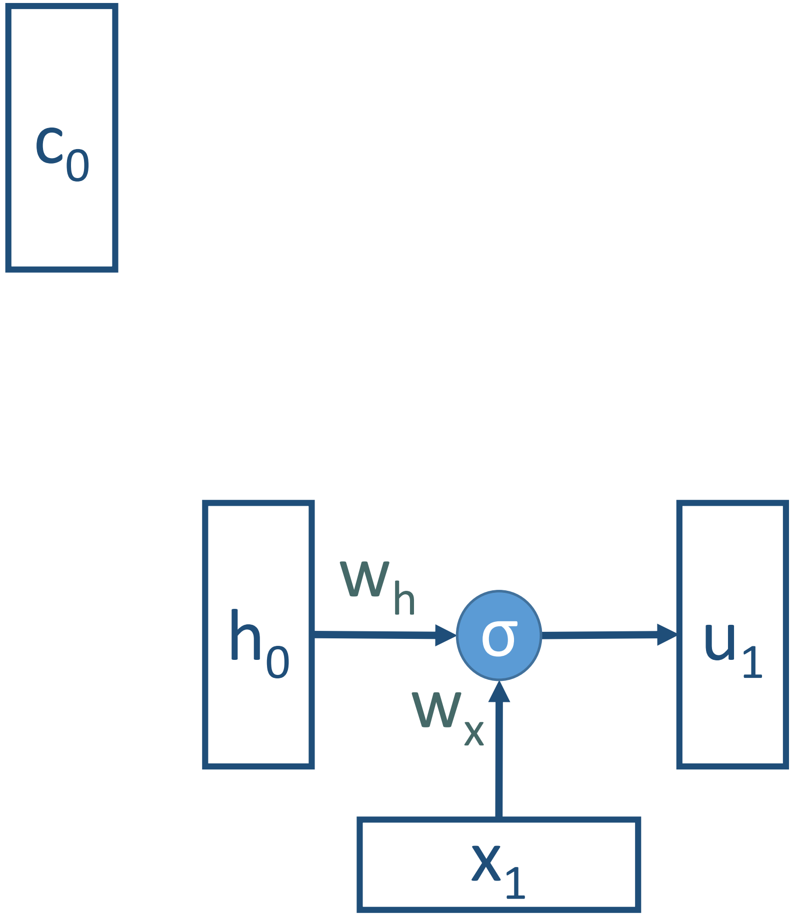
Transforming RNN to LSTM

$$u_t = \sigma(W_h h_{t-1} + W_x x_t)$$



Start with the same basic structure as our RNN, but call the output vector u1 instead of h1.

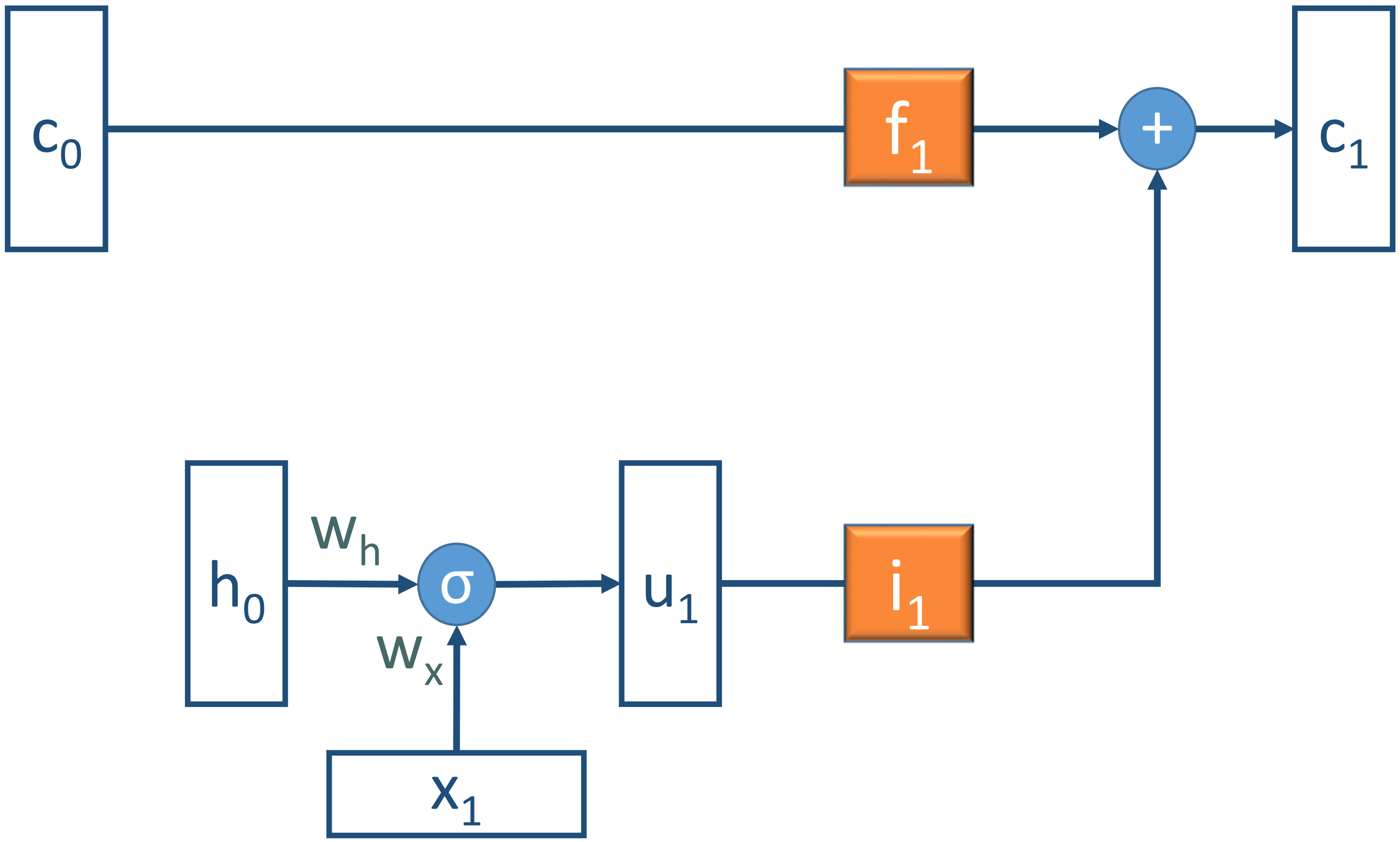
Transforming RNN to LSTM



Now we're going to add another vector, c, which will be our memory cell. C0 is usually initialized to all 0s, and we'll see how c is calculated for each time-step in a moment.

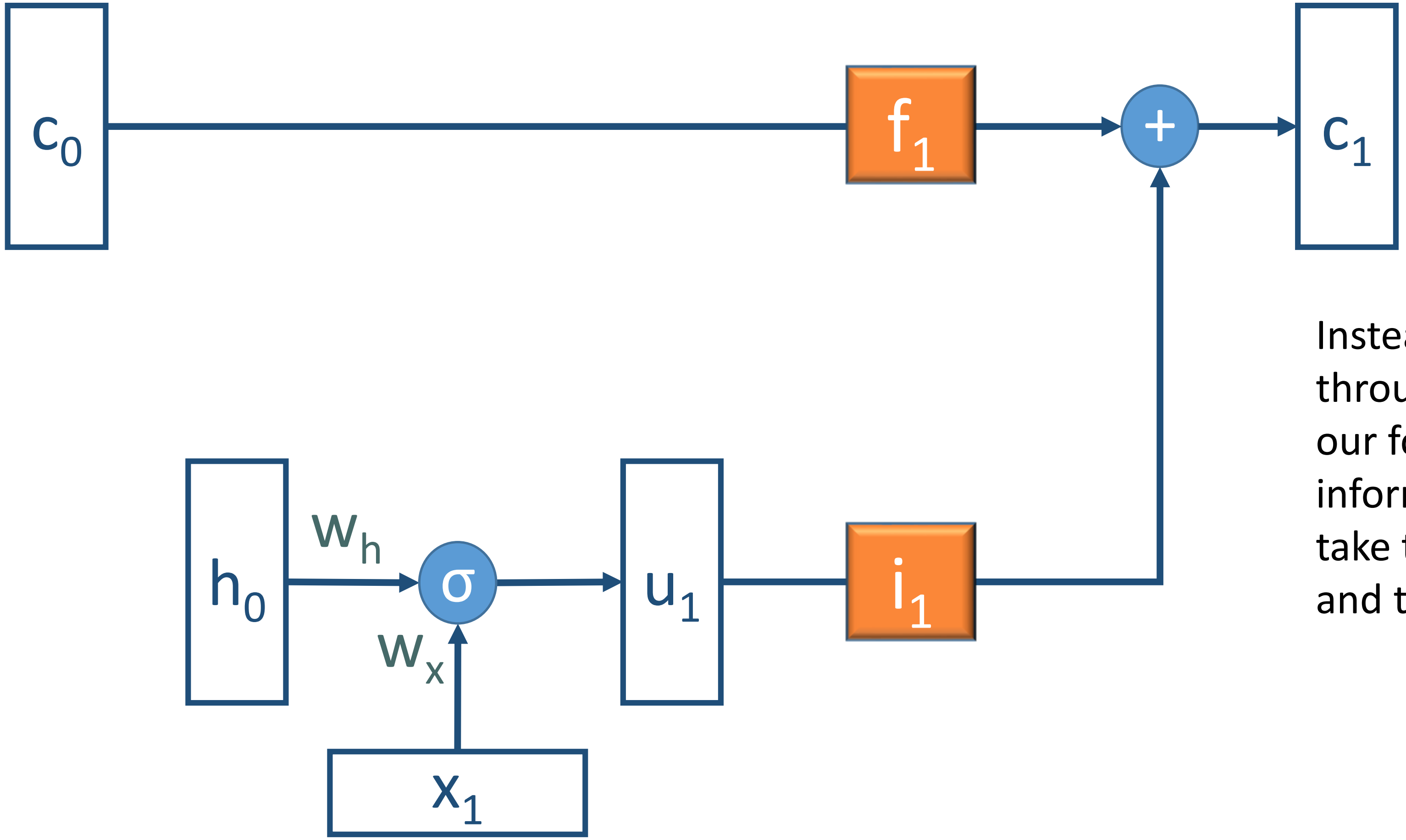
Basically, we're going to want to somehow combine c0 with u1 to get c1. There's a temptation to apply a set of weights to each and then apply some non-linear function..

Transforming RNN to LSTM



$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$

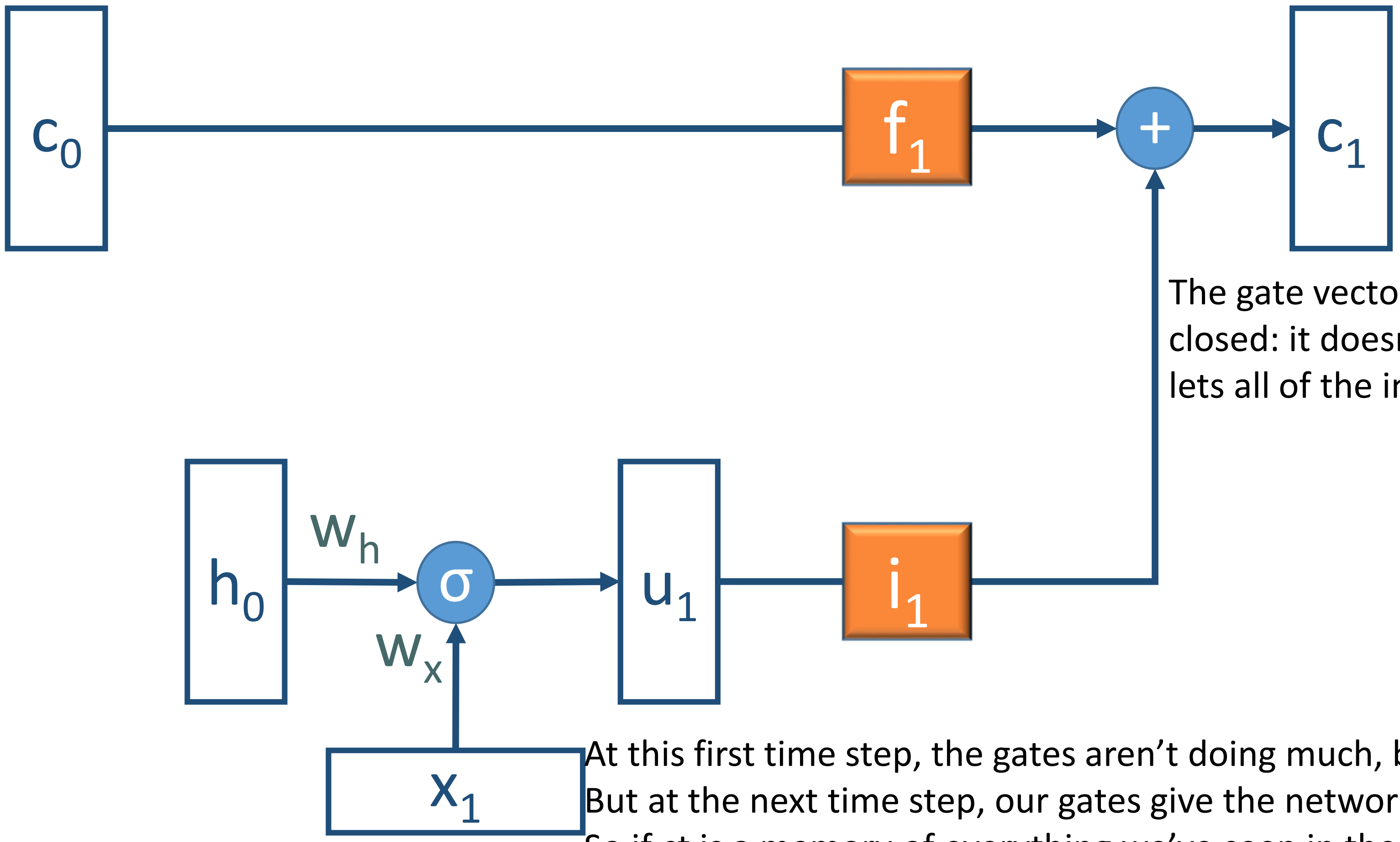
Transforming RNN to LSTM



$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$

Instead, what we're going to do is pass u1 through our input gate i1, and pass c0 through our forget gate f1, and take their sum. When I say information is "passed through" a gate, I mean take the elementwise product of the input vector and the gate vector

Transforming RNN to LSTM

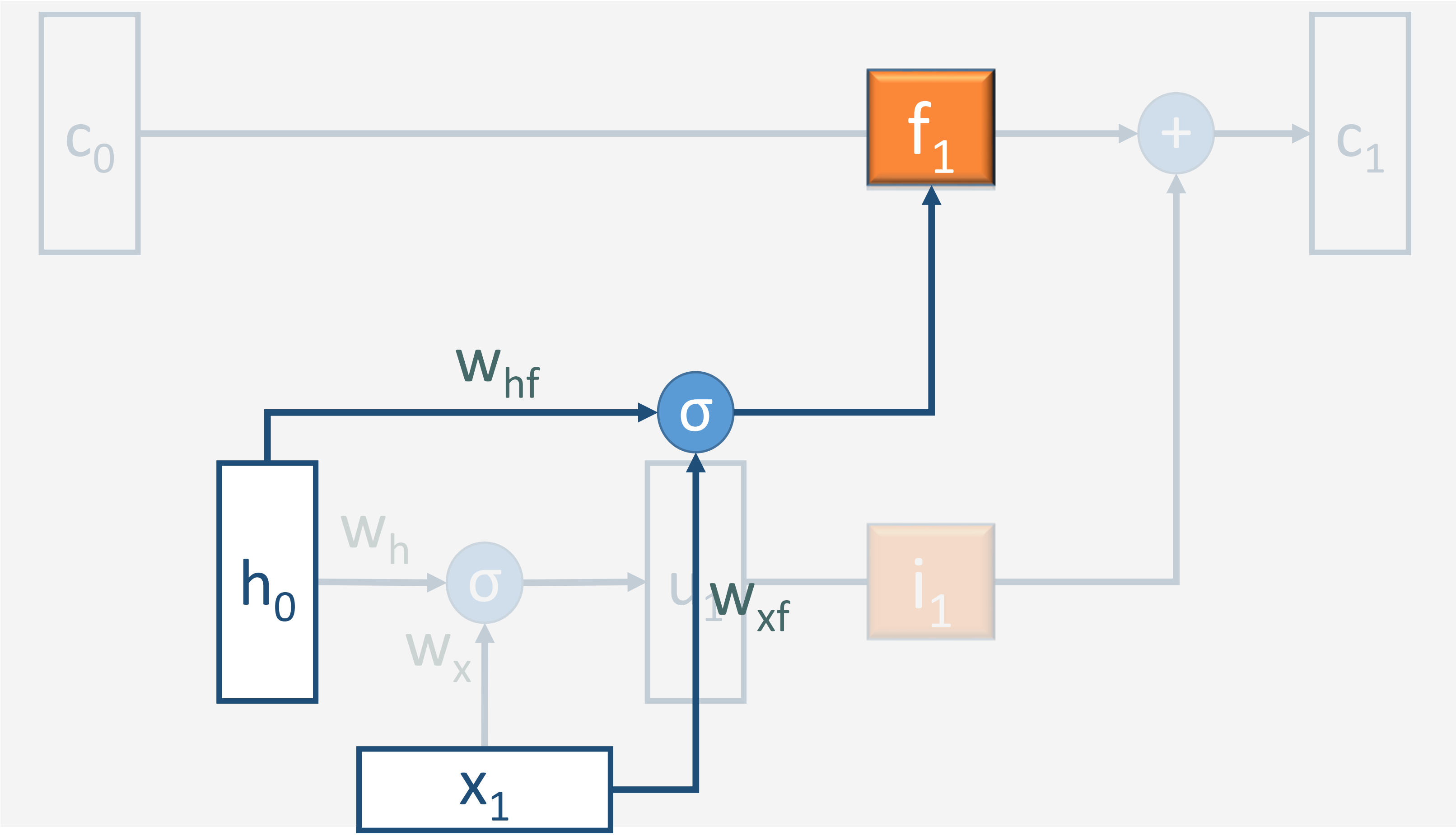


$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$

The gate vector will have values from 0 to 1. A gate value of 0 is closed: it doesn't let information pass through it. A gate value of 1 lets all of the information pass through it.

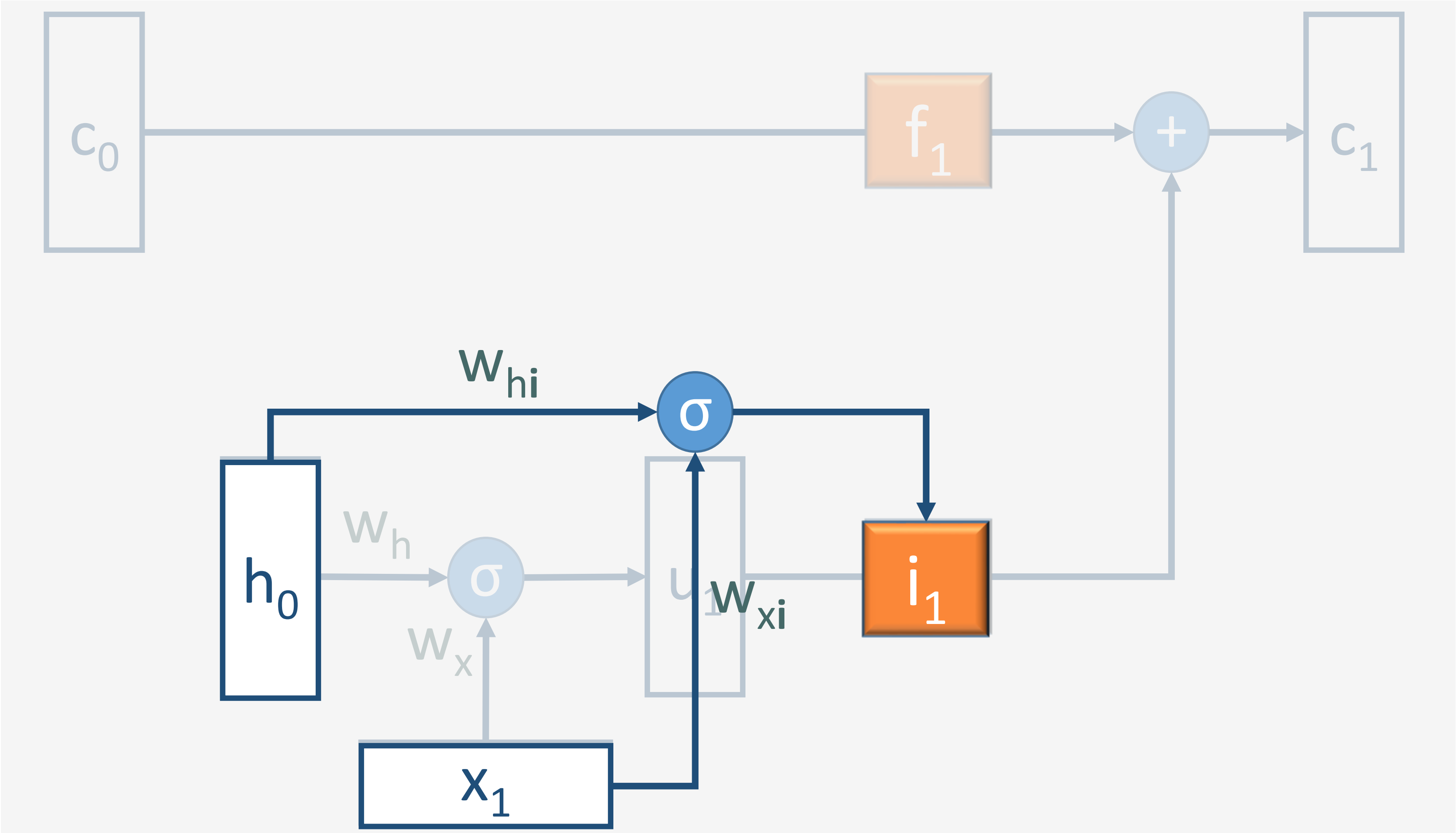
At this first time step, the gates aren't doing much, because we initialized c0 and h0 to vectors of zeros. But at the next time step, our gates give the network the option of treating the input c as a vector of zeros. So if ct is a memory of everything we've seen in the network up to time t, we can totally close our gate and use all 0s for ct at the next step – effectively forgetting everything and going back to our starting value for c.

Transforming RNN to LSTM



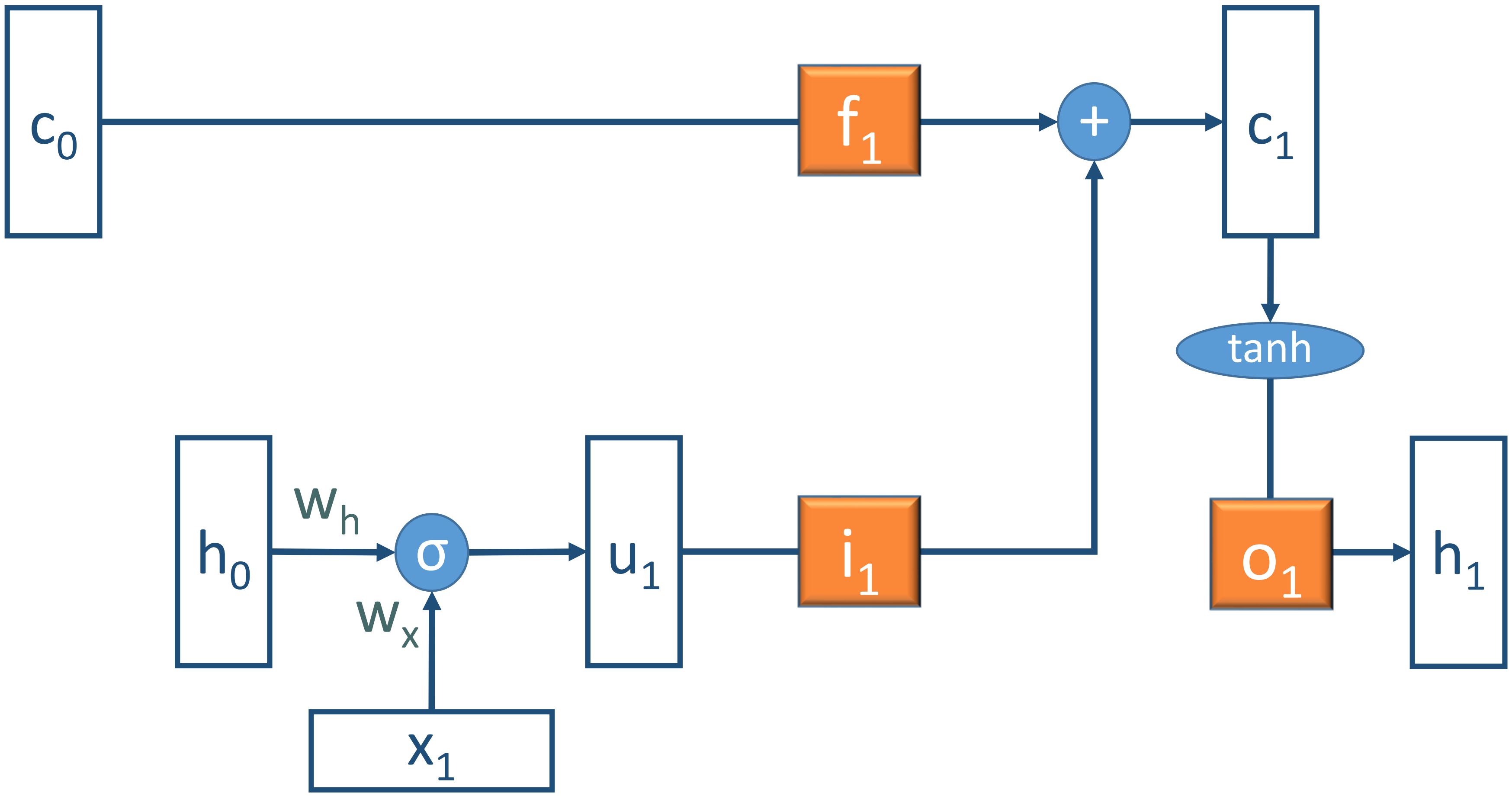
$$f_t = \sigma(W_{hf}h_{t-1} + W_{xf}x_t)$$

Transforming RNN to LSTM



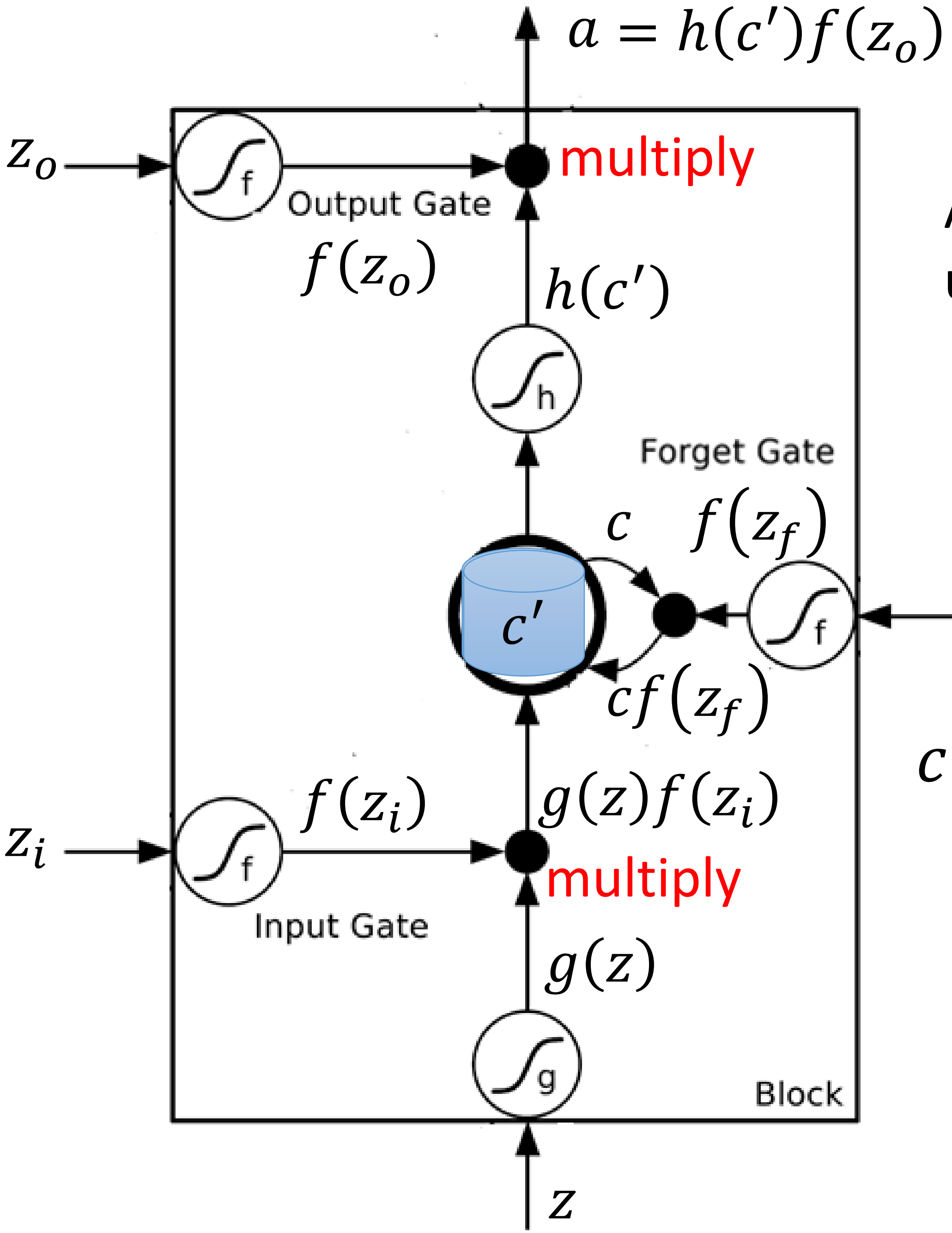
$$i_t = \sigma(W_{hi}h_{t-1} + W_{xi}x_t)$$

Transforming RNN to LSTM



$$h_t = o_t \odot \tanh c_t$$

Finally, we apply a non-linearity to our memory cell, c1, and pass that through the output gate to give us h1



Activation function f is usually a sigmoid function

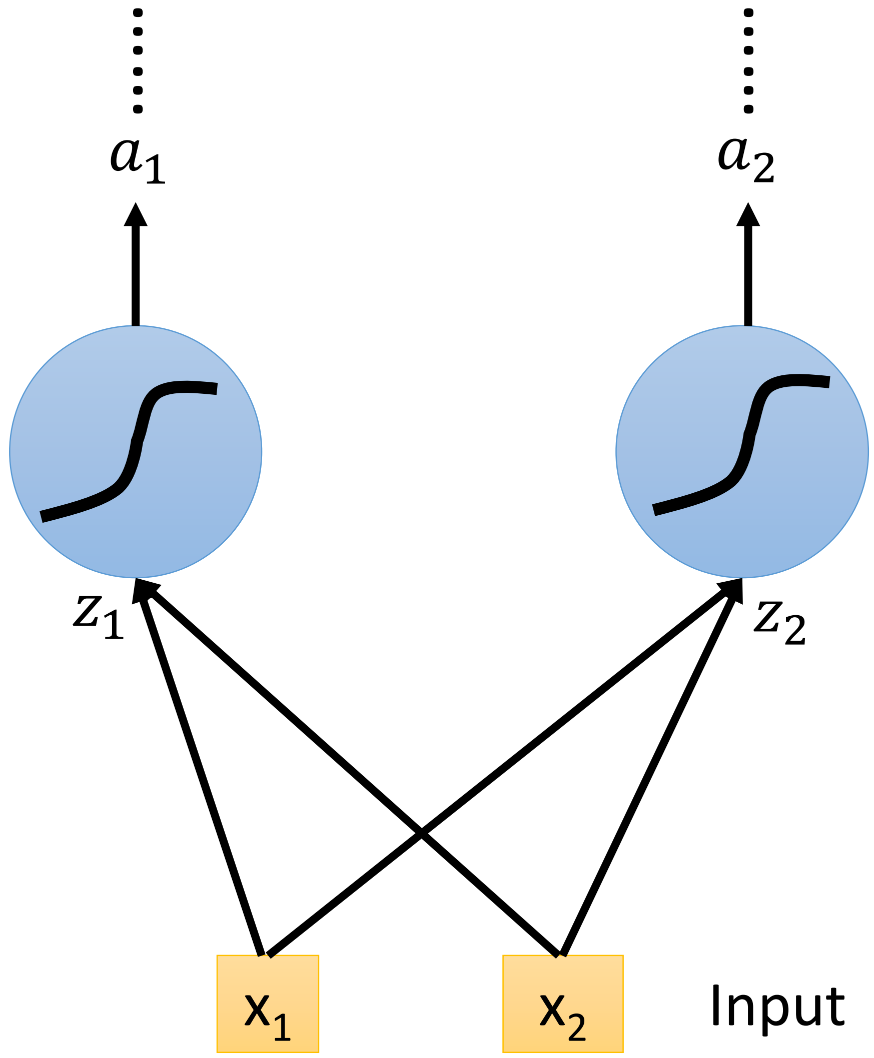
Between 0 and 1

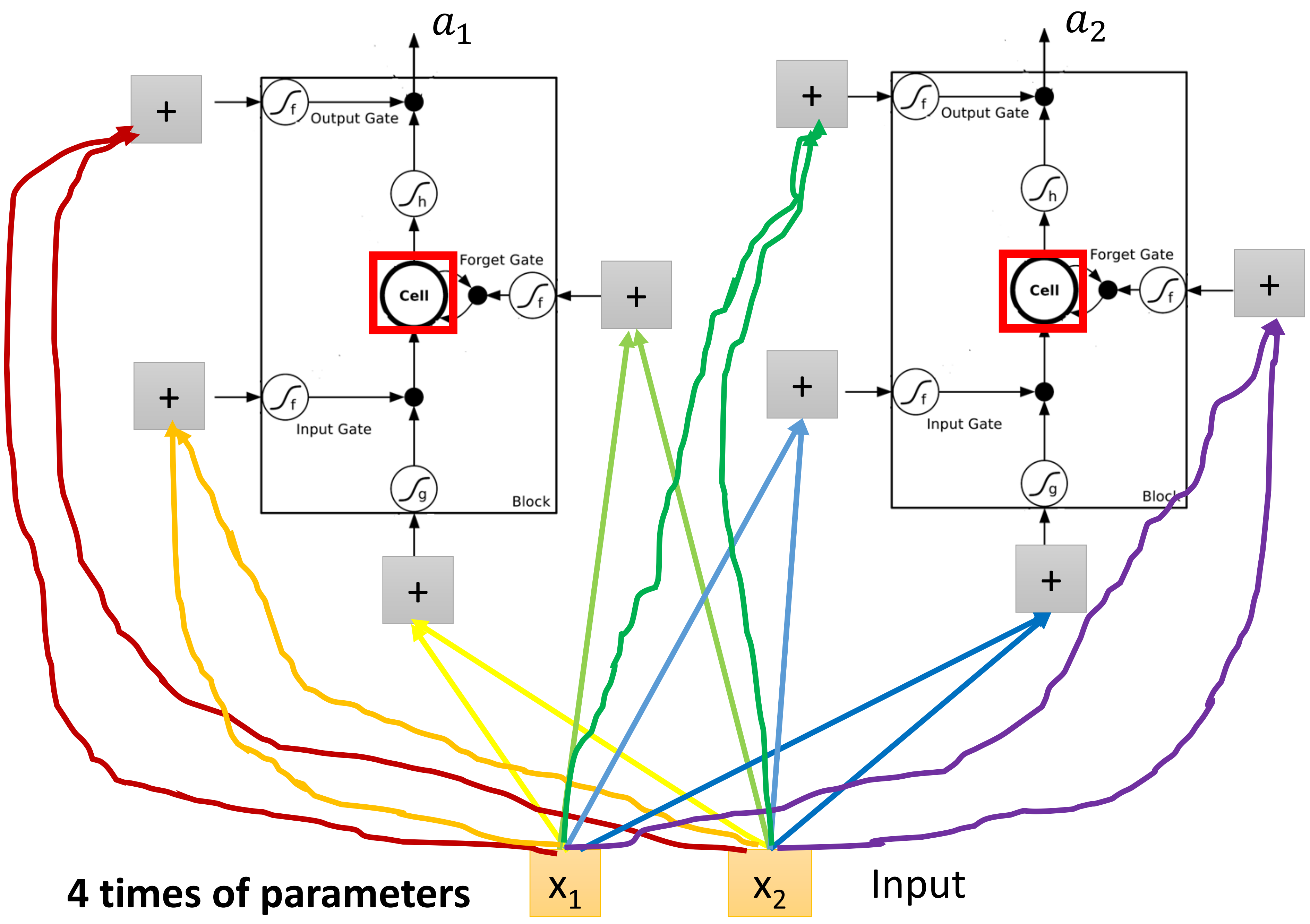
Mimic open and close gate

$$c' = g(z)f(z_i) + cf(z_f)$$

Original Network:

➢Simply replace the neurons with LSTM





a_1
a_2
+
Output Gate
Forget Gate
Cell
Input Gate
Block
x_1
x_2
Input
4 times of parameters

LSTM

Extension: "peephole"